ScreenIO

COBOL Screen Manager

Personal Computer COBOL Software Products by:

NORCOM
2217 North Jordan Avenue
Juneau, Alaska 99801

http://www.screenio.com
support@screenio.com

(907) 790-3386 (voice)
(907) 790-2780 (fax)

This manual accompanies ScreenIO, a screen manager for use with the COBOL language on personal computers running Microsoft Windows 3.1 and above.

Getting Started

About ScreenIO

ScreenIO is a screen handler for COBOL that was originally implemented as a character-mode application under MS-DOS. The Windows version of ScreenIO uses the same interface and panel library structure, which simplifies your transition from the legacy MS-DOS environment to the Microsoft Windows environment.

ScreenIO runs as a native Windows application, using either 16-bit or 32-bit COBOL compilers. The 16-bit version runs in all Windows environments from Windows 3.1 through Windows NT. The 32-bit version requires a 32-bit version of Windows[1]; at least Windows 95 or Windows NT.

You do not need a Windows Software Developers Kit (SDK) in order to develop Windows applications with ScreenIO. All necessary Windows support is provided with this package.

You don't have to know how to program for Microsoft Windows to use ScreenIO in a Windows environment. We take care of all relevant Windows programming issues within ScreenIO. It's easy to get your COBOL program running under Windows with ScreenIO!

ScreenIO was developed using CA-Realia II Workbench versions 2 and 3. The 32-bit ScreenIO runtime, SCRWIN32.DLL, is a CA-Realia Workbench Version 3 executable implemented as a Windows dynamic link library.

ScreenIO is Year 2000 compliant.

ScreenIO was entirely written in COBOL.

[1] The 32-bit version of ScreenIO will not run on 16-bit Windows under the Win32s 32-bit subsystem.

Product Description

ScreenIO is a COBOL subroutine that manages your user interface under Microsoft Windows so that a COBOL programmer can easily perform use screen input and output.

It consists of the following components:

- A panel editor, which is used to design your panels.
- The ScreenIO runtime library, which is called by your program to display your panels and accept input from the user.

This facility provides the following major functions to the programmer using COBOL:

- A simple, interactive panel design facility specifically oriented toward the COBOL programmer.
- The ability to display a screen image and acquire user entered data with a single COBOL subroutine call.
- Automatic handling of caret positioning, tabbing, field colors and other display functions.
- Data validation (e.g., editing) and masking external to the user program.

ScreenIO is linked with your application just as any other COBOL subroutine would be. A 16-bit application that uses ScreenIO may be distributed as a single .EXE module like any other COBOL program. The 32-bit version will require that you distribute a couple of runtime DLL files (supplied with this package) with your executable; we do not charge royalties or runtime fees for distributing these files.

The panel editor was developed using COBOL, itself, and ScreenIO.

Operating Requirements

Compiler

You must have a compatible COBOL compiler.

The 16-bit version of ScreenIO requires CA-Realia COBOL 5.0; CA-Realia II COBOL Plus, or the CA-Realia COBOL II Workbench version 2.x.

The 32-bit version of ScreenIO requires the CA-Realia II COBOL Workbench version 3.x, IBM Visual Age COBOL, Fujitsu COBOL, Micro Focus Object COBOL 4.x, Micro Focus Net Express, or virtually any other 32-bit compilers that can generate CALLs to .DLLs using standard Windows linkage conventions.

Hardware

ScreenIO is intended for operation on personal computers running the Intel i386 (or higher) processor, and true compatibles running Microsoft Windows. If your computer will run Microsoft Windows, it will run your ScreenIO applications.

Operating Systems

The 16-bit Windows version of ScreenIO requires Windows 3.1 or higher, including Windows 95 and Windows NT.

The 32-bit Windows version of ScreenIO requires a 32-bit Microsoft Windows-compatible operating system; at least Windows 95 or Windows NT.

Disk Requirements

There are no significant disk requirements other than space for your executables and the files that they use. ScreenIO does not use external screen files; everything that ScreenIO needs at runtime is contained within your executable programs.

Support

NORCOM provides high quality technical support, as well as periodic upgrades for bug fixes, compiler upgrades, and enhancements. If you have a problem with your ScreenIO application, or if you have a suggestion for an enhancement, please contact us to discuss it.

Our telephone number, mailing address, email address, and Internet URL can be found at the front of the manual.

Maintenance period

Your purchase of a ScreenIO license includes one year of maintenance and upgrades. We will automatically issue system upgrades as they become available. Maintenance after the first year is available for a modest annual fee; we will bill you.

Version Differences

This Windows version of ScreenIO is presently limited to the OEM character set for compatibility with MS-DOS legacy ScreenIO applications. The Windows ANSI fonts do not support line draw and other special characters commonly used in MS-DOS applications.

16-bit ScreenIO runs as a native Windows application with CA-Realia COBOL version 5.x. It provides a number of graphical Windows functions, such as scrollbars, Windows-style pulldown menus, and message boxes.

32-bit ScreenIO runs as a native 32-bit Windows application with any 32-bit COBOL compiler (though it is callable from *any* 32-bit language; not just COBOL). It has all of the features of the 16-bit version, in addition to some features we implemented only in the 32-bit Windows environment.

Acknowledgments

CA-Realia COBOL, CA-Realia II COBOL Plus, and CA-Realia II Workbench are trademarks belonging to Computer Associates, Inc.

Intel and i386 are trademarks belonging to Intel Corporation.

Microsoft, Windows, Windows 95, Windows NT, and MS-DOS are trademarks belonging to Microsoft Corporation.

Micro Focus COBOL, Object COBOL, and NetExpress are trademarks belonging to Micro Focus, Inc.

Programming Overview

A ScreenIO application consists of screen images (panels) and COBOL program(s) that manage them. Here's how you use ScreenIO:

- First, design your panels using the panel editor. Type text on the screen, draw lines and boxes, and paint things the color you want. Mark the locations of data fields on the panel and define the characteristics of those fields in COBOL terms. It's very easy.

- Next, save the panel and generate the panel copybook[2]. The panel copybook contains everything ScreenIO will need to display the panel, plus the field definitions you need to load data to the panel and to receive data from the panel.

- Write your program. Copy the panel copybook from the panel editor into your WORKING-STORAGE SECTION. To display your panel, just call ScreenIO and pass it the data names provided to you in the panel copybook.

- ScreenIO is generally used in screen-at-a-time mode. In this mode, control will return to your program when the user presses a function key or selects a menu item using the mouse. All of the data the user typed into the fields on the panel will be sitting in the fields you defined in the copybook. It's much like using record-oriented I/O to the screen and keyboard.

At this point, what you do with the data is up to you. That's ScreenIO in a nutshell. Now, let's take a closer look at how you create your panels…

[2] Copybooks are known as "copy members" in some environments.

ScreenIO

The Panel Editor

ScreenIO's panel editor allows you to visually design your panels. Define menus, draw lines and boxes, type text, paint colors, and define data fields in COBOL terms.

Panels are stored in a file called a panel library. You may store as many panels in a library as you like, and you may have more than one panel library. Generally, you will find it convenient to create a panel library for each discrete application.

Keyboard Conventions

End - Exit Current Panel

We use the End key to exit a panel. In some cases, you will be prompted to supply an override key, (usually Alt-Z) to exit in spite of a potential data loss.

Tab - Activate Next/Previous Field

The Tab and Shift-Tab keys are used to navigate from field to field. You may also use the up and down arrow keys for this.

Esc - Restore Original Contents of Field

If you type something unintentionally, and want to restore what was there previously, press Esc to restore the original data before you press a function key or exit the field.

Backspace - Destructive Backspace

Pressing the backspace key will delete the character immediately to the left of the cursor and shift the cursor and the data following it one character to the left.

Ctrl-End - Erase to End of Field

The Ctrl-End key combination erases all characters from the cursor position to the rightmost end of the current field.

Starting the Panel Editor

Should I use the 16-bit or 32-bit panel editor?

We distribute two versions of the panel editor; PEF16.EXE (16-bit), and PEF32.EXE (32-bit). PEF16 runs on any version of Windows; PEF32 only runs on 32-bit versions of Windows.

Which version should you use? If you're running a 32-bit version of Windows, use PEF32. You only need to use PEF16 if you need to run the panel editor under Windows 3.x or if you want to be able to edit your panel libraries with older versions of the panel editor.

Both versions of the panel editor produce identical panel copybooks which may be used with earlier versions of ScreenIO. There is only one difference you should consider.

PEF16 uses the same 16-bit file system for your source libraries as the DOS versions of the panel editor. PEF32 uses a 32-bit file system which must convert 16-bit source libraries before it can edit them. Conversion is automatic. Once converted to 32-bit, you can't convert it back to the 16-bit file system.

ScreenIO Source Library Conversion (32-bit ONLY)

If you are running PEF32, the 32-bit panel editor, and you specified a 16-bit panel library, the panel editor will automatically perform the conversion to the 32-bit file system used by the 32-bit panel editor. Your original source library will not be modified.

The conversion process copies panels from the original panel library to a library named REORG.SOR, then renames it *original-name*.PNL. We recommend that you use the .PNL extension to signify a 32-bit source library. If a file of the name *original-name.*PNL already exists, the converted library will be named *original-name*.TMP

Hit F4=Options to verify that the panel library name has been changed to *original-name*.PNL, and to correct your .INI file if necessary for subsequent editing sessions. Keep reading.

Specifying the Source Library Name and File Locations

The panel editor stores processing options, file names, and file locations in an .INI file which it writes whenever you close the panel editor[1]. The .INI file is passed to the panel editor as a command tail; that is, a string following the name of the executable:

[1] If you do not specify a .INI filename in the command tail of the panel editor command, the panel editor will create the file using the default filename PEF.INI in your Windows directory. This is, incidentally, the only directory where you are guaranteed to have read/write permission if you are running a shared copy of Windows on a network. That's why Microsoft recommends you store .INI files in the Windows directory.

For the 16-bit panel editor:

> **PEF16.EXE** *drive:\path\myapp*.**INI**

For the 32-bit panel editor:

> **PEF32.EXE** *drive:\path\myapp*.**INI**

If you are maintaining more than one application, it is very convenient to create a shortcut for each application on your Windows 95 or Windows NT desktop (or an icon for Windows 3.x), and then specify a different .INI file within each application.

The panel editor will accept files specified using UNC (Universal naming convention) format if your operating environment allows it.

The first time you fire up the panel editor, you will be presented with an options panel. This panel displays the file names and file locations for your chosen application. You can change these file names and locations at any time. The panel editor will update the .INI file specified when you press Enter and when you close the panel editor.

Here’s how to fill in these fields:

.INI file:

> Specify the full path and filename to your .INI file. If you didn’t specify a command tail when you invoked the panel editor, it looks in your Windows directory for PEF.INI . If not found, the panel editor will create it when you press Enter.

Panel Library:

> Specify the of your panel library in 8.3 format. For example, PANEL.PNL[2].
>
> If you have specified a library which does not exist, or if the panel editor can't find a panel library with the name you entered, you will be asked if you want to create the library. Press F1 if you want to create the library[3]. If you made a mistake entering the name or location of the panel library, just correct it and press Enter.

Panel Library Directory:

> The directory containing the panel library (don’t include the library name itself).

EDITMASK.SEQ Directory:

[2] We recommend you use the .PNL extension for libraries created by PEF32 (which uses the 32-bit file system). Use the extension .SOR for libraries using the 16-bit file system of earlier versions of ScreenIO and PEF16.

[3] If the subdirectory specified for the Source Library location does not exist, the panel editor will not create it; the operation will fail. Create the directory and try again.

The location of your EDITMASK.SEQ file. This is the file that contains your edit masks. We suggest that you keep this in the same subdirectory as your panel libraries.

Copybooks:

The panel editor creates a copybook for each of your panels. This is where the panel editor will put the copybooks. It is convenient to place them in a separate subdirectory rather than mixing them with other files.

The panel editor will save all of these items in your .INI file when you terminate your editing session. If you have many applications to maintain, you will find it convenient to have a separate .INI file with appropriate parameters for each of your applications.

The panel editor also stores its own Windows state information, including the size and location of the panel editor window, the font you are using, and several other items. The next time you fire up the panel editor, it will look the same as it did when you quit your last session.

Press Enter when you have finished entering these items. The panel editor will edit the filenames and subdirectories for validity. Press End to return to the Main Menu.

Main Menu

The Main Menu shows a little version and registration information. You can get more information by selecting Help/About.

Press F1 (or click on F1=Editor on the bottom line) to create or edit a panel. Enter the name[4] of a panel you want to create or edit:

If you know the name of the panel you intend to edit, or if you want to create a new panel, type it in the panel to edit field. Otherwise, select List to select an existing panel from a list. You can also use an existing panel as a template by entering its name in the Panel Template field.

Panel Specifications

Here's where you specify the general characteristics of your panel.

Panel Title

The title field is a comment which is displayed only on the panel listing (not on the panel itself). The date updated is the last time you modified this panel.

[4] A suggestion: We usually name the panel the same as the program that drives it, with a "P" suffix. Because we rarely have more than one panel in a program or subroutine (more on that later), it's easy to know the name of the program driving the panel. Plus, if you put the name of the panel in the upper corner, it's easy to get coherent problem reports from your users.

Panel Size

Specify how tall the panel is; 25, 43, or 50 lines. Panels are always 80 columns wide.

Home Field

Specify the number of the HOME field, which controls which field is active when the panel is first displayed or when the user hits the Home key.

Copybook Options

These fields direct the panel editor how to generate your copybook. The default values are usually appropriate.

COBOL Quote symbol

The style of quote mark you prefer to use when you write your COBOL programs. This way, the copybook that the editor generates will use a quote mark that conforms to your coding standards.

Named Attributes

The panel generates attributes for your fields that are the name of your field plus a suffix of -C, -P, or -O. The panel editor also generates a table of attributes that allow you to refer to the attributes by field number, but in most cases it's easiest to refer to them by name. That way, if you rearrange your fields on the panel, you won't have to change the field numbers you coded in your program!

Panel -name prefix (instead of the word "PANEL")

The panel editor creates a number of control fields for your panel when it creates the panel copybook. The names of these fields will be prefaced with the name of the panel: *panel*-EXIT-KEY, *panel*-MENU-MSG, *panel*-DISPLAY-OPTION, etc.

This avoids the need to qualify references to these fields when you have more than one panel in one program: e.g., PANEL-EXIT-KEY OF *panel*-PANEL. Most programmers find it inconvenient (and confusing) to use qualifiers.

If you specify N for this option, your control fields will be named PANEL-.

Window Panels

A window is a less-than-full-size panel; a small panel which pops up on top of another panel.

To design a window panel, you paint the part that you want to show just as you normally would, and then mark the rest of the panel with the transparent marker character. When ScreenIO displays your panel, it will only display the part of the image you painted. The transparent portion will be unchanged. It looks as though the window popped up on top of the existing panel image.

You make areas transparent by filling them with a "transparent marker" character. This character may be any character of your choice, except that alphabetic characters are not allowed.

To define a window panel

- Put a Y in the Window panel field.
- Enter a character in the transparent marker field, usually "@".

Panel Colors

Select the colors that will be used on the various parts of the panel. Colors are specified by entering the color number from the color bar at the right edge into each of the color number fields on this panel. The title of that field will change color to reflect your choice.

You can also specify the color of the Static Text. This is the default color; individual areas may be painted different colors later.

The active field is the field containing the caret[5]. It is usually desirable to make the active field a different color than the other fields to make it easy to find the caret.

The error color is used for fields that are determined to contain errors. If you mark a field in error, ScreenIO will paint the field the error color.

The Menu

Fill in the text of the menu line that will appear on the bottom line of your panel.

If you define your menu items using the form *function-key=description*, ScreenIO will automatically translate them if the user releases the left mouse button ("single click") while pointing anywhere within the phrase. For example, if a menu item is F1=Help, ScreenIO returns an F1 when it's clicked on. ScreenIO's mouse handling is automatic.

ScreenIO can also translate your menus to Windows-style menus, if you select that option when you make your initialization call to ScreenIO. This will be discussed in the Initialization topic in the Programming section.

User Exit Names

See the section on Advanced Topics for information regarding user exits.

Menu line Indicators

F10 will step through each of the possible menu status indicators. The status indicator will overlay menu text residing in the last 7 to 9 positions of the menu line.

[5] Windows terminology is different from DOS. In Windows, the cursor refers to the mouse pointer, and the caret is the insertion point; where you are about to type.

- Row-Column indicator, which shows where the caret is.
- Row only indicator.
- Column only indicator.
- Field and position indicator, which displays the field number and the position of the caret within the field.
- Shift states indicator which shows the status of the SHIFT keys (left and right), the Num-Lock, the Scroll-Lock, and the Insert key.
- 12 hour time of day.
- 24 hour time of day.
- ASCII/Hex representation of the character at the caret position.

When you're finished…

Press F2 to continue to the Full Screen Editor, or F5 to save your changes. Press End to abandon your changes (you will have to confirm this by pressing Alt-Z).

The Full Screen Editor

The Full Screen Editor is basically a blank canvas used to lay out your panel. You can type or draw anywhere on the panel except for the menu line.

Functions Available:

F1 will toggle through the four available menus. You can also invoke most of the options by using Shift-, Alt-, or Ctrl- key combinations. When you press the Alt or Ctrl keys, the menu will change to show the available functions.

Painting

Painting large areas is best done using block operations. Painting smaller sections of text, for highlight or emphasis is easy and accomplished as follows:

F9	Toggle painting on/off.

F3	Toggle the background color. When painting is on, the menu line is displayed using the color combination you selected.
Shift-F3	Toggle the background color in the reverse order.
F4	Toggle the foreground (character) color.
Shift-F4	Toggle the foreground color in the reverse order.
F2	Toggle the background between normal and high intensity.

To paint, just move the caret over the area you want painted or drag it by holding the left mouse button down while you move the mouse. The caret will paint the position that it leaves, similar to typing.

If you paint the first character of a data field that you'd defined, the field will assume the color that you painted it, overriding the colors you selected in the field editor.

Drawing lines

You can draw lines and erase in any direction with this function.

F7	Toggles drawing on/off.
F8	Toggles the drawing style.

Move the caret using the arrow keys or drag it with the mouse to draw the lines and boxes you desire. The proper corner and intersection characters are inserted automatically as you change direction.

Marking data fields

F5	Mark the first character of a data field. The panel editor will mark the location with a "tire track" block character. The field has been marked but no definition has yet been entered. Moving or deleting this character does not move or delete the field!
Shift-F5	Delete an undefined field if the caret is on the field mark.
F6	Goes to the Field Editor to define fields; if caret is on a field, it will start with that field.

Note:

You can't type over a data field. Once you define a data field, it is fixed. It can be moved using block operations, or deleted, but you cannot type or draw lines over it.[6]

Block and Line Operations

The panel editor provides block operations to move text and fields, paint large areas, and fill large areas with a single character (as for marking windows).

Alt-B	Mark one corner of a block at the caret position. If you move the caret and press Alt-B again, it will mark the diagonal corner of a rectangular block. Double clicking the left mouse button also marks block corners.
Alt-U	Unmark block. If on a block mark, only unmark that corner; otherwise, both corners are unmarked. Double clicking the right mouse button also unmarks block corners.
Alt-F	Fill a block with a specified character.
Alt-K	Copy a block to the caret position.
Alt-M	Move a block to the caret position.
Alt-P	Paint a block the currently selected color.
Alt-T	Tab the caret from one block mark to the other.
Alt-D	Delete the line the caret is in.
Alt-I	Insert a line in front of the line the caret is in.
Alt-C	Center text horizontally within a line or a block.

Note:

For the block copy *any fields that were in the marked block are not copied to the new block, but rather new fields are marked with the field mark character (but not fully defined) in the new block.*

[6] A limitation of the panel editor allows you to move data field marks using the Insert, Delete, and Backspace keys; this does not, however move the data field. <u>You can only move data fields using the block functions!</u>

IMPORTANT: Data fields cannot be lost or wiped out by a move or copy operation. If you try to copy or move a block to another area of the panel in such a way that data fields would be overlaid the following two[7] *rules apply:*

If **text** *from the source area would overlay data fields in the target area, the move/copy will be executed, but the fields will not be overlaid.*

If **fields** *from the source area would overlay fields in the target area or be moved off the panel, the move/copy request will not be performed.*

The exception to the above rules is if the move *is such that the source and target areas overlap, and if the collision of fields would have occurred inside the source area, the panel editor will allow the move.*

Ctrl-Key functions

Use the Ctrl key in conjunction with the arrow keys to move quickly around the panel. The combinations are:

Ctrl-Up	Position caret to top line.
Ctrl-Down	Position caret to bottom line.
Ctrl-Left	Position caret to beginning of line.
Ctrl-Right	Position caret to end of line.
Ctrl-End	Erase from caret to end of line.

Menu 4 - Miscellaneous functions

Press F1 to cycle through the menus to get to menu 4, which allows you to set up a custom tabbing route.

F5 - Custom Tabbing

The F5 key will take you to the Tabbing Editor, where you may specify the order in tabbing will proceed through your fields.

[7] There is in fact a third rule which states that a field included in a source block must be fully within the block. You cannot move half of a field.

The Field Editor

You have to define the fields that you marked in the full screen editor. The definition of a data field consists of the COBOL name, PICTURE clause, and VALUE clause, the field's color, type, and processing options.

How It Works

The field editor shows you a monochrome image of your panel in the upper portion of the screen. This image is used to show you which field you are defining at any given time. Only a portion of your panel is shown. The Field Editor will scroll this image as necessary to show you the rest of your panel image.

The field you are about to define is marked by a yellow field mark character.

After you have supplied the complete and correct definition for a field, the field will be marked on your panel with a diamond character in the first position of the field, and a small solid square for the remaining portion.

Note:

Once defined, a field preempts all other use of its position on the screen. You can move a field only by using block operations, and you can modify or delete a field only by using the Field Editor.

After you define fields, you can invoke the Field Editor at any time to display or modify the characteristics of any field.

The field editor will begin with the first field on the panel unless the caret is on the first character of a field when you press F6. In that case, the field editor will begin with that field definition.

Defining data fields

COBOL data name

Enter any legal COBOL data name for the field. It may be up to 30 characters long, although it must be shorter if you intend to use Named Attributes or field replication.

The field editor will not allow you to use a COBOL reserved word for a data name, except that the word FILLER may be used.

PICTURE clause

The PICTURE clause field must be a legal PICTURE clause. It is 18 characters long, which supports any legal PICTURE clause that you could use.

ScreenIO supports only display data types. COMPUTATIONAL data types or EDITED data types are not supported.

The only PICTURE types supported are:

X	ALPHANUMERIC data, no restrictions on content.
A	ALPHABETIC data, characters must pass a standard COBOL ALPHABETIC test to be entered in this field type[8].
9	NUMERIC data, field will only accept NUMERIC digits.[9]

The longest field supported is 80 bytes. The field editor will not allow you to define a field that would overlap any other field. Fields may not wrap from one line to another.

VALUE clause

There are two 40-byte value clause fields. The second one is a continuation of the first, since a field may be up to 80 bytes in length and may have a VALUE clause up to 80 bytes in length.

If you do not enter a value, the field will contain the value generated by your COBOL compiler for uninitialized data fields. ***CAUTION*****:** Some compilers initialize fields to SPACE or LOW-VALUE (even numeric fields) which could cause problems!

The value you enter must agree with the PICTURE clause. Figurative constants such as SPACES, ZEROS, LOW-VALUES, etc., are allowed. If entered in lower case, the field editor will force them uppercase automatically.

If you specify an alphanumeric literal, the first character must be a quote. Do not enter the trailing quote; it will be supplied by the panel editor when the copybook is generated. The type of quote must agree with what was specified previously.

Field editing and validation

In addition to the COBOL specifications of a field, ScreenIO uses the field type to determine what characters may be entered in a field, edit masks to specify how data for the field should be displayed, and validation data to determine how values keyed into the field should be edited.

[8] COBOL only accepts characters in the English as ALPHABETIC.

[9] There are exceptions to this for decimals, signs, and currency characters. See Edit Masks.

The following items are used by ScreenIO to manage the editing and validation of your data fields.[10] These items will not be accessible to your program at run time.

Field Type

The field type tells ScreenIO how to handle data entry into this field. Field types are cross-edited with the PICTURE clause, and only legitimate combinations are allowed.

A list of field types which are legal for any given PICTURE clause may be seen by pressing F1 when the Type field is active.

As you enter a value in this field, the field type description will be immediately displayed to the right of the value you entered.

The NUMERIC field types are:

1 **Integer numeric:** Accepts numeric digits and sign characters. An edit mask with a sign is required if the PICTURE is signed.

2 *Decimal numeric*: ***Rarely used***. Assumed Decimal. Accepts numeric digits, signs, currency symbols, thousands separator, and decimal points. *Last digits entered are assumed to fall to the right of decimal, unless an explicit decimal is entered.* Similar to a fixed place adding machine.

3 **Decimal numeric:** Explicit Decimal. Accepts numeric digits, signs, currency symbols, thousands separator, and decimal points. *The last digits entered are presumed to fall to the left of the decimal, unless an explicit decimal is entered.* Similar to an electronic calculator mode. This is generally preferable to type 2 for entering numeric data.

4 *Date field (Numeric): (Date field type 5 is more versatile)*. Accepts numeric digits, /, and - characters. The number entered must be a legal six-digit date. The format of the date is MMDDYY in North America, YYMMDD in Japan and DDMMYY in Europe.[11]

[10] ScreenIO actually has no way of knowing the PICTURE clause of your data field. The PICTURE clause is for the benefit of your COBOL program. The ScreenIO subroutine actually bases its handling of each data field on the field type, edit mask, validation data, and processing options specified when fields were defined.

[11] The format of the date and the characters used for the currency symbol, decimal separator, and thousands separator are determined by the country specific information defined to Windows via the [country] section of the WIN.INI file (Windows 3.x) or via the Regional Settings (Windows 95 and NT).

If you develop an application in one country for use in another, be sure to test your application after configuring your computer for the target country!

5	**YMD-translated Date field (Numeric):** Accepts numeric digits, /, and - characters. The number entered must be a legal six or eight digit date (depends on the PICTURE clause). This field always passes the date to your program in YYMMDD format (or CCYYMMDD format if picture 9(8)). The date is displayed by ScreenIO according to the country date format. Only valid with the type D edit mask. Works with either 6 or 8 digit dates, depending upon the picture clause used to define the field. This is an extremely useful date field because any rational programmer will store dates in YMD format, so this field eliminates the need for reformatting dates before and after displays. Plus, you don't have to do anything special to have your dates displayed correctly in other countries.

The Non-NUMERIC field types are:

Blank	**Any character:** Accepts any character.[12]
A	**Alphanumeric uppercase**[13] Accepts alphanumeric characters only, letters (including non-English) will be forced uppercase.
B	**Alphanumeric lowercase**: Accepts alphanumeric characters only, all letters (including non-English) will be forced lowercase.
C	**Alphanumeric mixed case:** Accepts alphanumeric characters only, case will be respected.
D	**Alphabetic uppercase**[14] Accepts alphabetic characters only, all letters will be forced uppercase.
E	**Alphabetic lowercase:** Accepts alphabetic characters only, all letters will be forced lowercase.
F	**Alphabetic mixed case:** Accepts alphabetic characters only, case will be respected.

[12] Except for LOW-VALUES and the ASCII control characters 08 (Backspace), 09 (Tab), 13 (Enter), and 27 (Escape).

[13] By alphanumeric we mean those characters which will pass a COBOL ALPHANUMERIC class test. This includes only the characters A-Z, space, and the numeric digits 0-9. It specifically excludes special characters and non-English alphabets.

[14] Again, the definition used for alphabetic characters is the COBOL ALPHABETIC class definition, which includes only the English alphabet; letters A-Z and space.

G	**Any character uppercase:** Accepts any character.[15] Letters (including non-English) will be forced uppercase. This is not the same as A above, as some characters are neither alphabetic nor numeric and therefore will not pass the COBOL class test.
H	**Hexadecimal:** Accepts and displays hexadecimal characters (0-9, A-F) only, two characters displayed per picture clause byte.
I	**Any character, first forced uppercase (if it is a letter):** Accepts any character[16], case respected on all but first character. Works for non-English alphabets, too!

Non-data entry fields:

P	**Pointer Field:** Data entry not permitted, has visible caret, and arrow keys operate to allow movement within the field. Useful for allowing user to point at characters within a field. See the Tabbing Editor for an example of this field type in use.
R	**Return First Character Selector:** Has no caret. Works like a Selector field (type S), but in addition, if the user presses any character key ScreenIO immediately returns to the calling program with the character in the first byte of *panel*-EXIT-KEY.[17] Useful for menus.
S	**Selector:** Has no caret, data entry not permitted: Used to make menu fields or for selecting items from a list by tabbing to the item.
X	**Protected:** Output only, user cannot tab into this type of field.

Edit Masks

The edit mask advises ScreenIO how to format this field when it is displayed. It allows you to specify insertion characters such as currency[18] symbols, commas, slashes, dashes, blanks and what have you.

For instance, if the number:

1234.56

was displayed with an edit mask of:

[15] Except for LOW-VALUES and the ASCII control characters 08 (Backspace), 09 (Tab), 13 (Enter), and 27 (Escape).

[16] Excluding LOW-VALUES and ASCII control characters.

[17] Move *panel*-EXIT-KEY to a numeric data item PIC S9(4) COMP-5. Redefine this item as two one byte PIC X fields. The first of these contains the character entered.

[18] The currency symbol used by the edit mask facility depends on the country you are in. The country is specified in your Windows configuration.

$$$,$$$,$$$,$$$.99

the result would look like:

$1,234.56

Press F1 to display a list of masks. The mask you select must agree with the PICTURE clause and field type:

- If the PICTURE is NUMERIC, the mask must be a numeric mask.
- If the PICTURE has a decimal (V), the number of digits following the decimal in the mask must match the PICTURE clause.
- If the PICTURE has a sign, the mask must have a sign.

Data Validation

Validation relieves your program of some editing tasks by allowing ScreenIO to handle them for you. When the user fills or tabs out of a field, ScreenIO will compare the data in the field against a list and/or range of legal values that you specify when you define the field. If the entry does not match the validation specifications, ScreenIO will issue an error message without intervention from you.

You can specify individual values and ranges in your validation list. In this example, the field will accept the letters A, G, and Z, as well any letter in the range R-X.

A,G,R-X,Z

Value Separator

Specify the character that will be used to separate individual entries in a list. The character selected must not be one of the legal values for this field.

Range Indicator

Specify the character that will indicate a range. The beginning and ending values of a range are separated by the character you enter in this field. For example, if the letters A through M were legal values you might choose to enter a dash in this field. Then, the Values field could simply be coded as A-M.

Note:

You should not use a dash as a separator for signed numeric fields, since it precludes the entry of negative values. An underscore character is usually a good substitute.

Don't make the mistake of using ranges of alphabetic values more than one character long without a little thought. This can allow unwanted values to slip through ScreenIO's validation. For example, if you specify the range AA-ZZ on a two-byte alphabetic field, intending to accept only alphabetic entries, you may be surprised to find that B(space) will be accepted; B(space) is greater than AA and less than ZZ. If you intended that all entries be two alphabetic characters, you'll have to perform the edit yourself.

Values (for validation)

The list of legal values is entered in this field with the previously specified delimiters separating one value from the next. You may combine individual values and ranges in the same string, as:

1,23,43,99-157,200,655

You do not have to allow trailing spaces for alphanumeric values, or leading zeros for numeric items. If the field is masked, you may, if you wish, enter the data in its masked form:

$193,471.23_$265,333.45

where the "_" character was specified as a range character.

If you wish to include a blank as a legal value, put it anywhere except as the last argument in your validation string.

Note:

It is possible to specify a validation string that will reject any entry made by the user. (High end of range before the low end of the range, a list separator where you intended a range indicator). Check your validation values as part of your testing process!

The validation section of the ScreenIO subroutine performs a test for equality (on individual items) and a range test (for range items) using variables of the appropriate data type, e.g., numeric fields for numeric items and character fields for character items. Range tests are done assuming the first value is the lower end of the range. Range tests are inclusive; equivalence to the first or last value of a range is accepted as valid.

Field Processing Options

Field processing options are additional parameters specifying how the field should be handled.

Field processing options are selected by typing an X next to the option(s) you desire. You can generally combine many field processing options to achieve the effect you want. Illegal combinations are edited out by the field editor.

Field processing options may not be modified at run time.

Must Valid

Must Validate - ScreenIO will not allow a user to exit a panel until all Must Validate fields contain data that will pass your validation specifications.

Note:

Normally ScreenIO will validate a field only if you tab from the field. The field may contain invalid data if the user never fills up or tabs through the field.. Must Validate eliminates all possibility of a field containing data that does not pass your validation specifications when control returns to your program.[19]

Required

Required fields must contain other than LOW-VALUES or SPACE before ScreenIO will return to your program. Any value will do, including pre-loaded values.[20]

No Autoskip

Normally, when the user types the last character in a field, ScreenIO tabs to the next field. You may suppress this by selecting No Autoskip. You must press Tab to go to the next field.

Clear First

Strongly recommended for numeric fields, this option causes ScreenIO to clear the field when the user enters anything in the first position of the field. If the user moves the caret to other than the first position and types a character/number, the field will not be cleared.

Clear Buffer

Selecting the clear buffer option will cause ScreenIO to clear the keyboard buffer of any remaining keystrokes when the field becomes active. This will prevent a user typing ahead of the application.

Blank if ZERO

For use with numeric fields, this option causes a field to be blank if the value is zero.[21] This will prevent the field from containing any digits, edit mask characters, signs, etc., if the value is zero.

---

[19] Although this sounds great, it has a significant and frequently undesirable side effect. See the section on Programming. Consider: What happens if the user doesn't know the correct value to enter in the field? They can't get out unless they fake it (or you can use Function 11 to allow certain keys to bypass the Must Validate logic).

[20] Like the Must Validate option, this one also has the possibly negative effect of prohibiting a user leaving a panel until all Required fields contain something.

Beep Full

The Beep full option will play the Windows asterisk sound when a character is entered in the last position of a field; that is, when the field is filled.

HOT Field

HOT field allows your program to regain control when the user either fills a field or tabs out of the field. A HOT field which has No Autoskip selected will return to your program as soon as the field is filled, as well as when the user presses Tab.

ScreenIO returns control to your program and indicates a hot field was exercised in two ways:

- The control field *panel*-EXIT-KEY will contain a value of 1000 plus the field number instead of a function key value.
- The paint attribute *field-name*-P will contain H (for HOT return).

Hint:

HOT fields allow you to perform inline edits that are more complex than ScreenIO's internal validation. You can also initiate actions without requiring the user to select a menu option.

HOT Modify

This option is identical to HOT field with one exception; ScreenIO will only return to your program if the user has modified the field. The field is considered to be modified if the user types a character, clears the field with Ctrl-End, or presses Backspace.

This option is often desirable in panels with lots of HOT fields that are primarily used for editing; it improves performance when tabbing from field-to-field since unnecessary processing is suppressed.

Return Left-Right Arrows

ScreenIO normally reserves the left and right arrow keys for moving within a field. This option will cause ScreenIO to treat the left and right arrow keys as function keys and return to your program.

Return Up-Down Arrow

This is similar to the previous option, except it applies to the up and down arrow keys. These keys are normally treated as Shift-Tab and Tab.

21 Uninitialized fields, regardless of data type, are set to LOW-VALUES by the some compilers. Blank if zero treats such fields as zero, and they will be blanked.

Hint:

This function is most often used to allow caret movement up and down in screens formatted as columns. Your program may then determine the handling of the arrow key just like any other function key.

It's also very useful when browsing a file; you enable this option only on the top and bottom fields of the file list, thereby causing control to return to your program only when it's time to read the next (or previous) record and to scroll the list a record at a time.

Caret Off

Causes the caret to be turned off when this field becomes active.

Caret 1/2

Causes the caret to be set to ½ character wide when this field becomes active.

Caret 3/4

Causes the caret to be set to ¾ character wide when the field becomes active.

The Field Editor Function Keys

The field editor panel function keys are defined as follows:

F1 - Codes

Depending upon the active field, F1 will display a list of Field types or edit masks.

F2, F3, F4 - Field Color Selection

Function keys F2, F3, and F4 act exactly as the color selection keys in the full screen editor. They toggle through the various color combinations, and the combination last selected will be the color assigned to this field.

The menu will be displayed using the color you select, and the color will be displayed in text form in the color and location section of this panel.

F6 - Next Field

The F6 key advances to the next field. Shift-F6 selects the prior field.

F7 - Delete Field

This is the only way you can delete a field that has been defined in the Field Editor.

F9 - Replicating Fields

This function is used to copy a field (across or down), except for the field name.

The field editor will append a suffix of the form -01, -02, -03, etc., to the name of the original field (unless it is named FILLER.

The Tabbing Editor

It is sometimes handy to move through fields in some other than the default order of left to right, top to bottom. The tabbing editor allows you to change the default tabbing order. You can specify backward tabbing, up and down tabbing, or any tabbing order that suits your application.

Using the tabbing editor

While at the full screen editor, press F1 to toggle the menu until you see F5=Custom Tabbing. Then, press F5. You will see a monochrome image of the panel you were designing. You may not type on this image or change any text.

The functions are Demo Tabbing, with which you can see the tab order in action, and Redefine tab order, which you use if you would like to change the tabbing order.

F3 - Backward

This function allows you to simulate tabbing backward through the fields. Each time you press F3, the caret will jump to the prior field in the current tabbing sequence.

F4 - Forward

Similar to the above option, this jumps forward to the next field each time you press it.

F9 - Redefine Tab Order

When you press the F9 key one of two things will happen.

- If you previously defined a customized tabbing order, you will be advised that the tabbing order will be reset to the default order, or,
- The directions for designing your custom tabbing order will appear in the menu.

If you did not want to continue, press Enter.

Specifying the tabbing order

The tabbing order of the fields is established by moving the caret through the first character of each field, using the caret movement keys.

As you move the caret, little arrow symbols will trace your course. DON'T PANIC! These arrows are not destroying your panel text, they just show where you've been. Your PC will play the Windows asterisk sound each time you record a field in the tabbing order.

You may "drive" over any of the little arrows, as well as any previously recorded field. No harm is done by this; only the first time you cross a field determines the order of that field.

You don't need to drive over all of the fields. Those that you don't drive over will be appended to the end of the tabbing sequence, using the normal left to right, top to bottom tabbing order. It doesn't matter whether you pass through protected fields.

When you are finished, press F9.

F1 - Removing Custom Tabbing

Press F1 to restore the original default tabbing sequence of left to right, top to bottom.

When to apply custom tabbing

We recommend you define the tabbing order after you have finalized the layout and content of your panel. There are two reasons for waiting:

- If you define new fields after you have applied the custom tabbing order, the new fields will be added at the END of the tabbing sequence.
- When you move fields using Block operations, the tabbing order is retained. This may not be appropriate, considering the new physical location of the field on the panel.

Leaving the tabbing editor

To leave the tabbing editor, just press the End key when you are finished trying out your custom tabbing order. If you specify normal tabs (F1 instead of driving the caret) you will be returned to the full screen editor automatically.

Generating the Panel Copybook

After you create your panel, you must generate the panel copybook that your program will use for passing control information and field data to and from ScreenIO.

The Panel Copybook contains control fields[22], your data fields, and the panel runtime data. The runtime data describes the static portion and the field specifications of your panel, and must not be modified by your program.

The panel copybook is a standard ASCII text file with tab compression. It must be included in the WORKING-STORAGE section of your COBOL program with a simple copy statement:

COPY *panel***.**

Note:

Never, never, never change the panel copybook using your programming editor! Each time you generate a panel, its copybook is rebuilt from scratch, and any changes you would have made will be lost. You should modify values of the panel copybook only by using the panel editor and regenerating the copybook.

Copybook Destination

The panel editor will place copybooks in the directory that you specified in your options. We recommend you keep them separate from other source code; it's more convenient that way.

The copybook generator is selected from the main menu by pressing F2. Press F1 to select a panel from a list. Press Enter to create the copybook.

Options Available

The options are specified by placing a single letter in one of several fields on the screen. The single letter is the first letter of the options to the right of the field.

Generate Quantity

The default is to generate a single panel. If you changed several panels in your source library, you can generate copybooks for all of the revised panels, or ALL panels in the library.

[22] Explained in Programming.

Named Attributes

This option specifies whether the field attributes are to be individually named. If you disabled this option when you created the panel, you can force named attributes using this option.

The field attribute names will be of the form: *field-name*-P, -C, and -O.

Copybook Comments

There are some comments in the copybook which will be unnecessary after you are familiar with using ScreenIO. These can be suppressed.

Panel-name Prefix

The control fields in the panel copybook are, by default, prefaced with the name of the panel. If you overrode this option when you created or edited the panel (the control fields are prefaced with PANEL-) you can still impose the panel name prefix when you create the panel copybook.

Source Library Manager

The source library manager is used to copy panels from one library to another, delete panels from a source library, or to compress the source library. The libraries are simply ISAM files. Like any ISAM files, they must be reorganized occasionally to improve performance and recover disk space.

Hint:

We recommend you use a separate library for each project or application. This simplifies library management and organization, as well as reducing risk of loss.

The source library manager is invoked from the main menu. Three functions are available:

- Library reorganization.
- Copy panels from one library to another.
- Delete panels from the library.

The main panel of the library manager displays only two data fields. The first advises you of the name of the current source library. The second field is used to specify the output library for copying panels. Panels are copied from the current library to the output library.

Library Reorganization

Reorganization is used to recover waste space in the library ISAM file. Reorganization can improve performance and conserve disk space, particularly in large, active source libraries. An ISAM file is reorganized, or compressed, by copying it sequentially to a new file.

This is what happens when we reorganize your source library:

- Your source library is renamed with an extension of BAK.
- Your source library is then copied to REORG.SOR in the panel library subdirectory.
- REORG.SOR is renamed to your original source library name.

As you can see, your old source library has not been changed; it has merely been renamed with an extension of .BAK.

The .BAK file

When you have finished reorganizing your Source Library, there will be a file with an extension of .BAK. This is the old copy of your library. This .BAK file will be replaced the next time you reorganize your source library.

The panels in the library will be listed as they are copied to the new library. If the panel editor is unable to rename REORG.SOR to the name of the original library, you will be advised to do it manually. It’s no big deal.

Copying panels from one source library to another

To move panels from one library into another, use the copy feature on the source library main menu. You will first have to fill in the name of the target library, which may include the full path. If the target library does not exist, it will be created.

Press F2, which will take you to a panel where you may select panels to copy.

Selecting panels to copy

The selection panel allows you to mark (with any character) the panel or panels you wish to copy to the target library. After you have marked your choices, press Enter to start the copy process. After the copy is completed, the selection screen will page forward in the panel list if there is more than one page.

Deleting Panels

Deleting panels is almost exactly like copying panels. Press F3 for the delete selection panel.

It operates the same as for copy; mark your selections with any character and press Enter. The panel will be deleted[23] from this library, and the word DELTED will show in the panel date field.

[23] Actually, being rather timid about physically deleting panels, we have designed this facility to COPY the panels to a special library called DELETED.PAN; before we delete it from your library. If you accidentally delete a panel, it can be recovered by making DELETED.PAN your current Source Library and copying the panel back into your working library.

Programming

After you have created your panel with the panel editor, you must write the COBOL program that will drive the panel. You will find that using ScreenIO is, in many respects, similar to doing file I/O; only much simpler.

The programs you write using ScreenIO will all have two things in common:

- They will contain one copybook for each panel to be displayed.
- They will call the ScreenIO subroutine to display the panels.

Data Division

The DATA DIVISION code required to use ScreenIO is limited to several COPY statements:

- SCRWINIT.COB; the standard copybook we provide to initialize ScreenIO.
- KEYVALUE.COB; the copybook containing a table of function key values; so you can determine what caused ScreenIO to return to your program.
- *panel*.COB; the copybook generated for each panel (where *panel* is the name you chose for the panel).

Procedure Division

The PROCEDURE DIVISION coding is very straightforward for most applications. First, you move the data you want to display into the data fields you defined in the panel editor. Then, you simply call ScreenIO using the four arguments provided in the panel copybook.

The ScreenIO subroutine will manage all aspects of the display and keyboard such as tabbing, data type checking, error detection (if validation is specified), etc. Control will return to your calling program when the user presses a function key, selects a menu option using the mouse, or tabs out of a HOT field.

When ScreenIO returns, the fields in the copybook will contain the data that was displayed on the panel, whether it was initially placed there by your program or keyed by the user.

Other fields in the copybook will tell you where the caret was when control was returned, what caused ScreenIO to return to your program, and many other items of interest (although most of the time, you won't need to pay attention to most of them).

Your program then evaluates *panel*-EXIT-KEY to see why ScreenIO returned to your program.

It then performs the requested action, which may include editing the data, writing it to a file, performing calculations, or whatever COBOL guys do when they get their hands on new data.

When your application is finished, you *must* call ScreenIO using function 15 to shut down your application's window. Do this only once, just before you STOP RUN.[1]

Sample Application

Here's a sample program that uses ScreenIO. As you will see, it's very easy to implement a simple ScreenIO application.

The sample program calculates the monthly payment and total amount paid on a mortgage loan. We don't do any file I/O, table searches, elaborate editing, or printing in this example since we presume you already know COBOL.

The Sample Panel

The panel SAMPLEP has three unprotected fields for entering the loan amount, number of months, and the interest rate. We utilize validation on these fields to make sure the numbers entered are positive and within the limits the program is designed to handle.

The panel also has two fields that are protected (available for output only), which are used to show the monthly payment and the total amount of the payments.

[1] It is imperative that you shut down your ScreenIO application properly.

The Sample Program

The program name is SAMPLE. The COBOL source code is provided with the package as the file SAMPLE.COB.

Let's briefly go over what this sample program does, statement by statement:

Call ScreenIO to initialize

We provide a copybook, SCRWINIT.COB, that contains the initialization parameters for ScreenIO. You can generally take the defaults, although you will probably want to set the title of your application's window, and the text for the About box.

For 16-bit applications, this is also where you must pass along a couple of parameters that are provided when your program is invoked by Windows.

Call ScreenIO to display your panel

The program displays the panel by calling ScreenIO using the arguments provided by the panel editor in the copybook SAMPLEP.COB.

ScreenIO will handle all screen and keyboard operations. Control will return to the SAMPLE program when the user presses a function key, such as Enter.

Check for a request to quit

We have decided that our program will treat the Alt-F4 key combination as a request to terminate the application. If the operator presses Alt-F4 (or selects File/Exit), we will fall through our PERFORM loop, call ScreenIO with function 15 to shut down, and then STOP RUN.

Note that testing for which function key was pressed is simply a matter of using the appropriate data name from the KEYVALUE.COB copybook. You could check for the numerical value instead, but the program would be less readable.

Edit the data

We will edit the input data if the user pressed Enter. We have arbitrarily decided that we will consider a value of 999,999.99 illegal, for the express purpose of demonstrating the recommended way of indicating errors[2] to a user.

If we find invalid data (loan amount of 999,999.99), we will mark this field in error by moving an **E** to the *field-name*-P attribute for the field.

[2] We could have done these edits solely with the validation data, but chose to do some of the editing ourselves to show you how it is done.

We also will issue a message telling the user what was wrong with the data by moving the error text to *panel*-MENU-MSG, then setting *panel*-DISPLAY-OPTION to 1 to tell ScreenIO that we have found an error and we want a message to be displayed. The message will remain in the in the status bar (32-bit) or message box (16-bit) until the user hits a key or clicks the mouse.

Then, loop around and call ScreenIO. That's all it takes to flag errors and issue messages!

Calculate the answer

We calculate the monthly payments and total amount repaid if no errors were detected. (We use WORKING-STORAGE fields with more significant digits to reduce the rounding error.)

Move the results to the ScreenIO data fields

To display the result of the calculations, we move them into the output fields, then loop back to call ScreenIO again. This will display the data and possibly accept a new set of input values for computing another loan.

Disregard unsupported function keys

If the operator pressed a function key other than Enter or Alt-F4, we simply loop and redisplay the panel. This effectively ignores the unsupported key(s), and because we changed nothing, it will look to the user as though nothing happened. This is a fail soft way of handling unsupported function keys.

You could also elect to issue an error message informing the user that the key they pressed is unsupported. You may do that by using *panel*-MENU-MSG and *panel*-DISPLAY-OPTION to place the message in the menu line, as we did above when indicating an error.

```
       IDENTIFICATION DIVISION.
      * -----------------------: Sample 32-bit program using
ScreenIO.
       PROGRAM-ID. SAMPLE.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-PC.
       OBJECT-COMPUTER. IBM-PC.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

      *---:Define working fields for improved accuracy

       01  WS-PAYMENT         PIC 9(6)V9(9).
       01  WS-INTEREST        PIC 9(2)V9(9).
       01  FUNCTION-SHUTDOWN  PIC S9(4) COMP-5 VALUE 15.

      *---:Copy initialization, function key, and panel copybooks.
      *
           COPY KEYVALUE.
           COPY SCRWINIT.
           COPY SAMPLEP.
      *
       PROCEDURE DIVISION.

      *---:Initialize ScreenIO options in SCRWINIT.COB copybook.
      *
           MOVE ALT-F4                       TO SCREENIO-CLOSE-
```

```
VALUE.
           MOVE 'Loan Calculator!'            TO SCREENIO-TITLE.
           MOVE 'About My Test Program'       TO SCREENIO-ABOUT-
TITLE.
           MOVE ‘Loan Calculator version 1.0' TO SCREENIO-ABOUT-
TEXT.
      *
           CALL 'SCREENIO' USING WINDOWS-INITIALIZATION
                                 WINDOWS-INITIALIZATION
                                 WINDOWS-INITIALIZATION
                                 WINDOWS-INITIALIZATION.
      *
      *----Perform loop until ScreenIO returns Alt-F4
      *      Display the panel
      *      If user hit Enter,
      *        If amount is 999,999.99,
      *          issue a message as follows:
      *          move ‘E’ to paint attribute of the field in error
      *          move an error message to panel-MENU-MSG
      *          move 1 to panel-DISPLAY-OPTION to display the
message
      *        else
      *          calculate payment
      *          move/compute results into panel fields
      *          position to S-LOAN-AMOUNT field
      *        end-if
      *      end-if
           End-perform.
      *
           PERFORM WITH TEST AFTER
                 UNTIL SAMPLEP-EXIT-KEY = ALT-F4
             CALL 'SCREENIO' USING SAMPLEP-PANEL
                                   SAMPLEP-PASS-TO-EXIT
                                   SAMPLEP-WORK-S
                                   SAMPLEP-WORK-D
             EVALUATE SAMPLEP-EXIT-KEY
               WHEN ENTER-KEY
                 IF S-LOAN-AMOUNT = 999999.99
                   MOVE 'E' TO S-LOAN-AMOUNT-P
                   MOVE 'Value not supported.' TO SAMPLEP-MENU-MSG
                   MOVE 1 TO SAMPLEP-DISPLAY-OPTION
                 ELSE
                   MOVE S-LOAN-RATE TO WS-INTEREST
                   DIVIDE 100 INTO WS-INTEREST
                   DIVIDE 12 INTO WS-INTEREST
                   COMPUTE WS-PAYMENT ROUNDED = S-LOAN-AMOUNT /
                    ((1 - (1 + WS-INTEREST) ** ( -1 * S-LOAN-
MONTHS))
                      / WS-INTEREST)
                   COMPUTE S-LOAN-TOTAL = WS-PAYMENT * S-LOAN-
MONTHS
                   COMPUTE S-LOAN-PAYMENT ROUNDED = WS-PAYMENT
                   MOVE 'C' TO S-LOAN-AMOUNT-P
                 END-IF
               WHEN OTHER
                 CONTINUE
             END-EVALUATE
           END-PERFORM.
      *
      *----Tell ScreenIO to shut down before STOP RUN (IMPORTANT!)
      *
           CALL 'SCREENIO' USING FUNCTION-SHUTDOWN
                                 FUNCTION-SHUTDOWN
                                 FUNCTION-SHUTDOWN
                                 FUNCTION-SHUTDOWN.
       STOP RUN.
```

Initialization

You have to make an initialization call to ScreenIO, and you can specify ScreenIO's behavior by setting the options in the (provided) copybook SCRWINIT.COB, shown below.

```
            01  WINDOWS-INITIALIZATION.
           *                          :---------------------------------
      -----
           * -----------------------: This sets various ScreenIO
      runtime
           *                          : options in the Windows
      environment;
           *                          :
           *                          : It is also used on an INFO call
      to
           *                          : obtain current state information,
           *                          : which can be saved and used to
      restore
           *                          : the state of your ScreenIO
      application
           *                          : the next time it is fired up.
           *                          :---------------------------------
      -----

             05  FILLER                  PIC S9(4) COMP-5 VALUE -1.

           * -----------------------: Name of your application's .INI
      file.
           *                          :
           *                          : If no .INI file is found, or if
      it
           *                          : contains no [ScreenIO] section,
           *                          : ScreenIO will initialize using
      the
           *                          : values in this copybook.
           *                          :
           *                          : NOTE:  You may specify a complete
           *                          : path to your .INI file.  If you
           *                          : specify only the filename,
      ScreenIO
           *                          : places it in the Windows
      directory.
           *                          :
           *                          : When you finish up and call
      ScreenIO
           *                          : with function 15 to close down,
           *                          : ScreenIO will save font, size,
      and
           *                          : location information in a
      [ScreenIO]
           *                          : section.  If the .INI file isn't
      found
           *                          : ScreenIO will create it for you.
             05  SCREENIO-INI-FILE       PIC X(255).

           * -----------------------: Returns ScreenIO handles
           *                          : to your application (needed if
      you
           *                          : call some Windows APIs directly
      in
           *                          : your application); ignored by
           *                          : ScreenIO at initialization time.
```

```
        05  SCREENIO-HWND-32         PIC 9(9) COMP-5.
        05  FILLER REDEFINES SCREENIO-HWND-32.
          10  SCREENIO-HWND-16       PIC 9(4) COMP-5.
          10  FILLER                 PIC XX.

        05  SCREENIO-HINST-32        PIC 9(9) COMP-5.
        05  FILLER REDEFINES SCREENIO-HINST-32.
          10  SCREENIO-HINST-16      PIC 9(4) COMP-5.
          10  FILLER                 PIC XX.

      * ------------------------: Displayed in your application's
title

        05  SCREENIO-TITLE           PIC X(255).

      * ------------------------: Windows About box contents

        05  SCREENIO-ABOUT-TITLE     PIC X(255).
        05  SCREENIO-ABOUT-TEXT      PIC X(255).

      * ------------------------: ScreenIO Windows options:

        05  SCREENIO-OPTIONS.
      *
      * ------------------------: Font Definition.  This is
documented
      *                         : in the Windows SDK, but the
easiest
      *                         : way is to select the font you
like
      *                         : within your ScreenIO application,
      *                         : make a ScreenIO function 17 call
      *                         : to obtain this data, then save
it.
      *
          10  SCREENIO-FONT.
            15  SF-HEIGHT            PIC S9(4) COMP-5 VALUE -12.
            15  SF-WIDTH             PIC S9(4) COMP-5 VALUE   0.
            15  SF-ESCAPEMENT        PIC S9(4) COMP-5.
            15  SF-ORIENTATION       PIC S9(4) COMP-5.
            15  SF-WEIGHT            PIC S9(4) COMP-5.
            15  SF-ITALIC            PIC X.
            15  SF-UNDERLINE         PIC X.
            15  SF-STRIKEOUT         PIC X.
            15  SF-CHARSET           PIC X VALUE X'FF'.
            15  SF-OUTPRECISION      PIC X VALUE X'01'.
            15  SF-CLIPPRECISION     PIC X VALUE X'02'.
            15  SF-QUALITY           PIC X VALUE X'01'.
            15  SF-PITCHANDFAMILY        PIC X VALUE X'31'.
            15  SF-FACENAME          PIC X(32) VALUE 'Terminal' &
x'00'.
          10  FILLER                 PIC X(32).

      * ------------------------: Menu translation;
      *                         :   WIN translates them to Windows
menu
      *                         : NOWIN does not; but default menu
remains

          10  SCREENIO-OPTIONS-MENU  PIC X VALUE '0'.
            88  SCREENIO-MENU-WIN          VALUE '0'.
            88  SCREENIO-MENU-NOWIN        VALUE '1'.
            88  SCREENIO-MENU-NOMENU       VALUE '2'.

      * ------------------------: ALT- key handling;
      *                         :   WIN is Windows conventions
      *                         : NOWIN is DOS convention (Most
ALT- key
```

```
      *                         :          combinations are trapped
by
      *                         :          ScreenIO).

          10  SCREENIO-OPTIONS-ALT   PIC X VALUE '0'.
            88  SCREENIO-ALT-WIN           VALUE '0'.
            88  SCREENIO-ALT-NOWIN         VALUE '1'.

      * ------------------------: SYSMENU
      *                         : (X box in upper right and box in
      *                         :  upper left with
Move/Size/Minimize/
      *                         :  Maximize/Close options)
      *                         :
      *                         :    ON enables system menu.
      *                         :   OFF disables system menu.
      *                         :
      *                         : See also SCREENIO-CLOSE-VALUE
below

          10  SCREENIO-SYSMENU       PIC X VALUE '0'.
            88  SCREENIO-SYSMENU-ON        VALUE '0'.
            88  SCREENIO-SYSMENU-OFF       VALUE '1'.

      * ------------------------: panel-EXIT-KEY value ScreenIO
returns
      *                         : when user selects FILE/EXIT or
CLOSE
      *                         :
      *                         : Note: If left 0, the system menu
CLOSE
      *                         : options will be grayed and
disabled,
      *                         : and may be enabled on a panel-by-
panel
      *                         : basis...
      *                         :
      *                         : If set to, say, 79 (END-key),
then
      *                         : ScreenIO will ALWAYS return 79 to
you
      *                         : if the user attempts to close
your
      *                         : application (systemwide).  If
your
      *                         : program calls ScreenIO
dynamically,
      *                         : you could handle this in your
SCRNIO.

          10  SCREENIO-CLOSE-VALUE    PIC S9(4) COMP-5 VALUE 0.

      * ------------------------: Status bar (32-bit only)
      *                         : Shows states of NumLock,
CapsLock,
      *                         : and Insert mode, if enabled;

          10  SCREENIO-STATUS-NUMLOCK PIC X VALUE 'Y'.
          10  SCREENIO-STATUS-CAPLOCK PIC X VALUE 'Y'.
          10  SCREENIO-STATUS-INSERT  PIC X VALUE 'Y'.

      * ------------------------: Location of upper left window
corner
      *                         : expressed as window
coordinates/pixels

          10  SCREENIO-ORIGIN.
            15  ORIGIN-X              PIC S9(4) COMP-5 VALUE 20.
            15  ORIGIN-Y              PIC S9(4) COMP-5 VALUE 20.

      * ------------------------: Size of your application's window
      *                         : expressed as window
coordinates/pixels

          10  SCREENIO-NORMAL-SIZE.
            15  SIZE-X                PIC S9(4) COMP-5 VALUE 500.
            15  SIZE-Y                PIC S9(4) COMP-5 VALUE 500.

          10  FILLER                  PIC X(49).

      * ------------------------: Allow/disallow multiple instances
of
      *                         : your application to run on a
single
      *                         : computer.

          10  SCREENIO-INSTANCES      PIC X VALUE 'Y'.
            88  MULTIPLE-INSTANCES-NO       VALUE 'N'.
            88  MULTIPLE-INSTANCES-YES      VALUE 'Y'.

      * ------------------------: Mouse parameters
      *                         :
      *                         : Action returned on left button
      *                         : double-click if no other action
is
      *                         : defined for the location (menu or
      *                         : selector field); default is Enter
      *                         : (value 13).

          10  SCREENIO-MB1-DOUBLE.
            15  SCREENIO-MB1DBL-CHAR  PIC S9(4) COMP-5 VALUE 13.
            15  SCREENIO-MB1DBL-FUNC  PIC S9(4) COMP-5 VALUE 0.
      *
      * ------------------------: Mouse action in selector/pointer
flds
      *                         : when button is released
      *                         :  0 = No action
      *                         :  1 = Simulate Enter key
      *
          10  SCREENIO-MB1-SELECTOR   PIC 9 VALUE 1.
      *
      * ------------------------: Translate mouse actions to arrow
keys
      *                         : if button 1 is held down
(dragging).
      *                         :
      *                         : Note:  Generally useful only if
an
      *                         : arrow key exit is active, as
      *                         : for drawing or painting...
      *
          10  SCREENIO-MB-ARROW-KEYS  PIC X VALUE 'N'.
            88  MB1-ARROW-ON                  VALUE 'Y'.
            88  MB1-ARROW-OFF                 VALUE 'N'.
      *
      * ------------------------: Keystroke simulated if you
release
      *                         : button 1 while in the ScreenIO
menu
      *                         : line, but didn't hit a menu item.
      *                         : This could invoke a HELP key...
      *
          10  SCREENIO-MB1-MISSED-CHAR PIC S9(4) COMP-5 VALUE 0.
          10  SCREENIO-MB1-MISSED-FUNC PIC S9(4) COMP-5 VALUE 0.
      *
      * ------------------------: Action of caret when you mouse to
      *                         : a new field; normally, position
the
      *                         : caret to the first position, then
      *                         : match to mouse cursor if there's
      *                         : another mouse action.  Other
option
      *                         : immediately matches caret to
mouse
      *                         : position (automatic if MB1-ARROW-
ON).
      *
          10  SCREENIO-MB1-FIELD-OPTION PIC X VALUE LOW-VALUE.
            88  MB1-SIMULATE-TAB              VALUE LOW-VALUE.
            88  MB1-MATCH-MOUSE               VALUE X'01'.
      *
      * ------------------------: Keystroke simulated if left
button
      *                         : is released when NOT in a field.
      *
          10  SCREENIO-MB1-NO-FIELD-CHAR PIC S9(4) COMP-5 VALUE 0.
          10  SCREENIO-MB1-NO-FIELD-FUNC PIC S9(4) COMP-5 VALUE 0.
      *
      * ------------------------: Value returned on Right button
      *                         : double-click.
      *
          10  SCREENIO-MB2-DOUBLE-CHAR   PIC S9(4) COMP-5 VALUE 0.
          10  SCREENIO-MB2-DOUBLE-FUNC   PIC S9(4) COMP-5 VALUE 0.
      *
      * ------------------------: Value returned on Right button
      *                         : release (254).  Could be used to
      *                         : display context-sensitive menu.
      *
          10  SCREENIO-MB2-RELEASE-CHAR  PIC S9(4) COMP-5 VALUE 0.
          10  SCREENIO-MB2-RELEASE-FUNC  PIC S9(4) COMP-5 VALUE 0.

      * ------------------------: Reserved for future use...

          10  FILLER                       PIC X(666).
```

Here is a brief description of the significance of the fields in this copybook. The same copybook is used as a parameter in the initialization call, and in a Function 17 (information) call to acquire information from ScreenIO. See further discussion in the Function 17 documentation.

SCREENIO-HWND-32 and SCREENIO-HWND-16

Not used for initialization.

SCREENIO-HINST-32 and SCREENIO-HINST-16

Not used for initialization.

SCREENIO-TITLE

Sets the title of your application window.

SCREENIO-ABOUT-TITLE

Sets the title of your application's About box.

SCREENIO-ABOUT-TEXT

Sets the contents of your application's About box. You can have more than one line by inserting a line-feed character (hexadecimal 0D) where you want a line break.

SCREENIO-FONT

Sets the font used by your application. The parameters are more-or-less defined by the Microsoft Windows Software Developers Kit and are not at all straightforward.

The easiest way to acquire the correct values to select a font is to select the font you want to use and then perform a Function 17 call to ScreenIO, which will fill this area in for you. You can then save it in a file for later inspection or later use, or you can examine the parameters using your debugger.

SCREENIO-OPTIONS-MENU

You can specify how ScreenIO will treat your menu. Just set the appropriate 88-level item to TRUE to make your selection. There are three possibilities:

SCREENIO-MENU-WIN

Setting SCREENIO-MENU-WIN to TRUE will cause ScreenIO to translate your panel menus to a Windows style menu, if they are of the form *function-key=menu-option*. If the user selects one of your choices from the Windows menu, ScreenIO will return the value of *function-key* for that choice to your program.

SCREENIO-MENU-NOWIN

Setting SCREENIO-MENU-NOWIN to TRUE will cause ScreenIO to display a minimal Windows style menu, but it will not translate your menus. The user can select File/Close, View/Fonts, and Help/About from the menu.

SCREENIO-MENU-NOMENU

Setting SCREENIO-MENU-NOWIN to TRUE will suppress the display of a Windows style menu

SCREENIO-OPTIONS-ALT

This parameter specifies ScreenIO behavior when the user presses the Alt key.

SCREENIO-ALT-WIN

Setting SCREENIO-ALT-WIN to TRUE will cause ScreenIO to treat the Alt key in a standard Windows manner; e.g., it will activate the Windows menu.

SCREENIO-ALT-NOWIN

Setting SCREENIO-ALT-NOWIN to TRUE will suppress the default Windows action of the Alt key (except for Alt-Tab, Alt-Space, and Alt-F6). ScreenIO will allow you to use Alt- combinations in your application.

SCREENIO-SYSMENU

This parameter enables or disables the Windows system menu, which allows the user to select File/Exit or the X box to close the application. If enabled, specify the value that ScreenIO is to return if this option is selected by placing it in SCREENIO-CLOSE-VALUE.

SCREENIO-SYSMENU-OFF

The Windows options for shutting down your application will be grayed out and cannot be selected.

SCREENIO-SYSMENU-ON

The Windows options for shutting down your application will be active. If the user selects the option to close the application, ScreenIO will present a message box asking them if they are sure they want to exit. If the user selects Yes, ScreenIO will return the value of SCREENIO-CLOSE-VALUE to you in *panel*-EXIT-KEY.

Note: ScreenIO will not shut down your application; you have to do it yourself. All this option does it to tell you that the user selected File/Exit from a Windows menu.

SCREENIO-CLOSE-VALUE

Value returned to your program in *panel*-EXIT-KEY if SCREENIO-SYSMENU was on. For example, if you MOVE ALT-F4 TO SCREENIO-CLOSE-VALUE, your program will receive an Alt-F4. The value does not have to be a real key value; it can be any value.

SCREENIO-STATUS-NUMLOCK

(32-bit only) A value of **Y** enables ScreenIO's Num Lock status in the status bar.

SCREENIO-STATUS-CAPLOCK

(32-bit only) A value of **Y** enables ScreenIO's Caps Lock status in the status bar.

SCREENIO-STATUS-INSERT

(32-bit only) A value of **Y** enables ScreenIO's Insert status in the status bar.

SCREENIO-ORIGIN

Sets the X and Y coordinates (in pixels) of the upper left corner of your ScreenIO application window. Negative values will offset the window to the left or above the top of the physical display. Caution: You can cause your application to be displayed completely off the physical display, which can make things difficult!

SCREENIO-NORMAL-SIZE

Sets the X and Y size (in pixels) of your application window.

ScreenIO will not let you set this below a minimum size. If you set this to a very large number, ScreenIO will display your application at the largest size possible for the font you have selected. This may exceed the size of the physical display.

SCREENIO-INSTANCES

A value of **Y** allows more than one instance of your application to run at one time. Otherwise, an attempt to fire up multiple copies of your application will result in the first one being activated.

SCREENIO-MB1-DOUBLE

Sets the value that ScreenIO will return if the user double-clicks the left mouse button when they are not in a field. The default is 13, which returns an Enter key to your program.

SCREENIO-MB1-SELECTOR

Sets the action of the mouse when the user releases the left mouse button in a Selector (type S) field. The default is to simulate an Enter key.

0	No action.
1	Simulates an Enter key (default).

SCREENIO-MB-ARROW-KEYS

Translates mouse actions to their arrow key equivalent when the left mouse button is held down while the mouse is moved. For certain rare types of applications (such as ScreenIO's panel editor), this is used for dragging, in conjunction with an arrow key user exit.

N	No action (default).
Y	Simulates arrow keys when left mouse button is depressed and mouse is moved. A user exit is needed to handle this meaningfully..

SCREENIO-MB1-MISSED

Specify the value you want returned if the user releases the left mouse button while in ScreenIO's menu area (the bottom line of the panel) but is not on a menu option. Key values are specified as discussed in Function 6, Key Redefinition. Usually zero (no action).

SCREENIO-MB1-FIELD-OPTION

Determines where the caret is placed when you position the mouse cursor in a field and press the left mouse button.

X’00’ (Default). (Low-value). Simulates a Tab; regardless of where the mouse is in the target field, the field will be activated and the caret will be placed in the first position. Pressing the left mouse button again will cause the caret position to match the mouse cursor position.

X’01’ Activates the field and matches the caret to the mouse cursor position.

SCREENIO-MB1-NO-FIELD

Values returned when the left mouse button is released when the mouse cursor is not in a field or the menu line. Key values are specified as discussed in Function 6, Key Redefinition. Usually zero.

SCREENIO-MB2-DOUBLE

Values returned when the right mouse button is double-clicked. Key values are specified as discussed in Function 6, Key Redefinition. Usually zero.

SCREENIO-MB2-RELEASE

Values returned when the right mouse button is released. Key values are specified as discussed in Function 6, Key Redefinition. Usually zero. This could be used for context-sensitive menus or windows as in Windows 95.

The Panel Copybook

This copybook was produced by the panel editor. Each panel you create will produce its own unique copybook. This copybook contains control information used by ScreenIO, and an area that will be used to pass your data between ScreenIO and your program.

```
       01 SAMPLEP-PASS-TO-EXIT         PIC X.
       01 SAMPLEP-WORK-S.
        05  (Filler items containing hexadecimal values...)

       01 SAMPLEP-WORK-D.
        05  (Filler items containing hexadecimal values...)

*PANEL*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%-SCREENIO-
%%%%%%%%%%%%%%%%%%%%%%%%%
      *  Panel Name:  SAMPLEP V:2.4B      Date Created: 03/17/98
      *  Panel Title: Interest Payment sample
*PANEL*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%

       01 SAMPLEP-PANEL.
        03 SAMPLEP-FIXED-PORTION.
         05 SAMPLEP-HEADER-DATA.
          10 SAMPLEP-NAME              PIC X(8)         VALUE
'SAMPLEP '.
          10 SAMPLEP-NUMB-OF-FIELDS    PIC 9(4) COMP-5  VALUE 005.
          10 SAMPLEP-BASE-COLOR        PIC 9            VALUE 4.
          10 SAMPLEP-ACTIVE-COLOR      PIC 9(4) COMP-5  VALUE 112.
          10 SAMPLEP-ERROR-COLOR       PIC 9(4) COMP-5  VALUE 206.
          10 SAMPLEP-MENU-COLOR        PIC 9(4) COMP-5  VALUE 071.
          10 SAMPLEP-MENU-LINE.
            15 SAMPLEP-MENU-LINE-A     PIC X(40)        VALUE
                        '     Enter data and press the Enter
key.'.
            15 SAMPLEP-MENU-LINE-B     PIC X(40)        VALUE
                         '                         End = Exit
'.
          10 SAMPLEP-MENU-MSG          PIC X(80)        VALUE
SPACE.
          10 SAMPLEP-DISPLAY-OPTION    PIC 9(4) COMP-5  VALUE ZERO.
          10 SAMPLEP-CURSOR-FIELD      PIC 9(4) COMP-5  VALUE 001.
          10 SAMPLEP-OFFSET-THE-CURSOR PIC X            VALUE 'N'.
          10 SAMPLEP-CURSOR-OFFSET     PIC 9(4) COMP-5  VALUE ZERO.
          10 SAMPLEP-EXIT-KEY          PIC S9(4) COMP-5.
          10 SAMPLEP-REPAINT-SCREEN    PIC X            VALUE 'Y'.
          10 SAMPLEP-SHOW-WAIT-MSG     PIC X            VALUE 'N'.
          10 SAMPLEP-ARROW-EXIT-NAME   PIC X(8)         VALUE '
'.
          10 SAMPLEP-PRE-EXIT-NAME     PIC X(8)         VALUE '
'.
          10 SAMPLEP-POST-EXIT-NAME    PIC X(8)         VALUE '
'.
          10 FILLER                    PIC X            VALUE 'C'.
          10 SAMPLEP-BORDER-COLOR      PIC S9(4) COMP-5 VALUE 04.
          10 SAMPLEP-PRESENT-FREQ      PIC S9(4) COMP-5 VALUE 3000.
          10 SAMPLEP-PRESENT-TIME      PIC S9(4) COMP-5 VALUE 0005.
          10 SAMPLEP-ERROR-FREQ        PIC S9(4) COMP-5 VALUE 0200.
          10 SAMPLEP-ERROR-TIME        PIC S9(4) COMP-5 VALUE 0010.
          10 SAMPLEP-CURSOR-TYPE       PIC 9            VALUE 1.
          10 SAMPLEP-INDICATORS        PIC 9            VALUE 0.
          10 FILLER                    PIC S9(4) COMP-5 VALUE ZERO.
          10 FILLER                    PIC S9(4) COMP-5 VALUE ZERO.
          10 FILLER                    PIC S9(4) COMP-5 VALUE ZERO.
          10 SAMPLEP-SHIFT-STATES      PIC X.
          10 FILLER                    PIC XX           VALUE LOW-
VALUE.
          10 FILLER                    PIC XX           VALUE LOW-
VALUE.
          10 SAMPLEP-CURSOR-ROW        PIC S9(4) COMP-5.
          10 SAMPLEP-CURSOR-COL        PIC S9(4) COMP-5.
          10 SAMPLEP-SCAN-CODE         PIC X.
          10 SAMPLEP-MOUSE-ROW         PIC S9(4) COMP-5.
          10 SAMPLEP-MOUSE-COL         PIC S9(4) COMP-5.
          10 SAMPLEP-CLOSE-KEYVALUE    PIC S9(4) COMP-5 VALUE 0.
          10 FILLER                    PIC X(69)        VALUE LOW-
VALUE.
        03  SAMPLEP-VARIABLE-PORTION.
         05 SAMPLEP-NAMED-ATTRIBUTES.
          10 S-LOAN-AMOUNT-P           PIC X.
          10 S-LOAN-AMOUNT-C           PIC S9(4) COMP-5.
          10 S-LOAN-AMOUNT-O           PIC X.
          10 S-LOAN-MONTHS-P           PIC X.
          10 S-LOAN-MONTHS-C           PIC S9(4) COMP-5.
          10 S-LOAN-MONTHS-O           PIC X.
          10 S-LOAN-RATE-P             PIC X.
          10 S-LOAN-RATE-C             PIC S9(4) COMP-5.
          10 S-LOAN-RATE-O             PIC X.
          10 S-LOAN-PAYMENT-P          PIC X.
          10 S-LOAN-PAYMENT-C          PIC S9(4) COMP-5.
          10 S-LOAN-PAYMENT-O          PIC X.
          10 S-LOAN-TOTAL-P            PIC X.
          10 S-LOAN-TOTAL-C            PIC S9(4) COMP-5.
          10 S-LOAN-TOTAL-O            PIC X.
         05 SAMPLEP-USER-ATTRIBUTE-TABLE REDEFINES
```

```
                                      SAMPLEP-NAMED-ATTRIBUTES.
          10 SAMPLEP-ATTRIBUTE-TABLE OCCURS 5 INDEXED SAMPLEP-IDX.
           15 SAMPLEP-FIELD-PAINT          PIC X.
           15 SAMPLEP-FIELD-COLOR          PIC 9(4) COMP-5.
           15 SAMPLEP-FIELD-OPTION     PIC X.
        05 SAMPLEP-USER-FIELDS.
Fld001   10 S-LOAN-AMOUNT              PIC S9(6)V99    VALUE ZERO.
Fld002   10 S-LOAN-MONTHS              PIC 999         VALUE 12.
Fld003   10 S-LOAN-RATE                PIC 99V99       VALUE 9.90.
Fld004   10 S-LOAN-PAYMENT             PIC S9(6)V99    VALUE ZERO.
Fld005   10 S-LOAN-TOTAL               PIC S9(8)V99    VALUE ZERO.
```

The DATA DIVISION of your program must have a COPY statement to include the copybook for each panel it will use. Each panel's copybook contains four 01 level items that are required to call ScreenIO.

panel-**PASS-TO-EXIT**

panel-PASS-TO-EXIT is defined as a one-byte item in the copybook. This area will not concern you unless you find it necessary to write a user exit subroutine. It is provided solely to allow programmers to pass data through ScreenIO to their user exit subroutine(s).

panel-**WORK-S**

panel-WORK-S is used to pass the definition of your panel's static data to ScreenIO at execution time. You must not modify this data.

panel-**WORK-D**

panel-WORK-D is used to pass the descriptions of your panel's fields to ScreenIO. This includes locations, colors, field type, processing options, and validation information. You must not modify this data area.

panel-**PANEL**

panel-PANEL contains control information for the panel and your data fields. Its contents are described in detail below in the following section.

Panel Copybook Fields

This section examines each field in the panel copybook and its significance. The fields are grouped by function.

The fields are referred to as *panel*-**NAME-OF-FIELD**, where *panel*- would be the name of your panel if you use the (default) panel-name prefix option, or PANEL if you do not.

Note:

You should NEVER directly modify the panel copybook directly by using a text editor. Your changes will of course be lost the next time you generate the panel. ALWAYS use the panel editor to modify your panels!

Panel Repaint Control

The number one ScreenIO technical support question is… *(Drum roll, please…)*

Q:	***"How can I display a panel and return to my program without waiting for the user to hit a key?"***
A:	***Just set this field to 'O' and call ScreenIO!***

This field allows you to control how much of the image ScreenIO will update. It is used primarily for improving efficiency when updating the screen from your program.

Usually, ScreenIO paints the entire panel image and all fields when a panel is first displayed. After that, ScreenIO will only paint the fields (but it, by default, paints all of them).

panel-**REPAINT-SCREEN**

On the call to ScreenIO:

Y	All user fields are repainted (default). It isn't necessary to set *panel*-REPAINT-SCREEN to Y when you first display a panel.[3]
N	Only the user fields which have been marked for repaint or in error will be repainted. All fields are automatically painted on the initial display of a panel.[4]
O	Display the panel and immediately simulate an Enter key, thereby returning control to your program without waiting for the user to take any action. Only the fields marked for repaint or in error will be repainted. This is useful for displaying status messages while your program is performing a long process.

[3] ScreenIO senses that you are displaying a different panel, and will automatically redisplay all user fields. If you set *panel*-REPAINT-SCREEN to I, you must explicitly mark fields for repainting. This is rarely used.

[4] For efficiency reasons, we recommend you set *panel*-REPAINT-SCREEN to N and explicitly repaint only fields you have changed when you next call ScreenIO with the same panel. Fields are marked for repaint using *panel*-FIELD-PAINT, described on the following pages.

R	This is a sort of realtime mode of processing. If no key has been pressed, ScreenIO immediately returns to your program as if this were an **O**. If a key was pressed, ScreenIO will act as though this field contained an N, and will process keystrokes in the usual way. This allows your program to operate in real time, calling ScreenIO occasionally to see if the user has done anything.
I	Suppress the static paint when first displaying the panel. Only the fields marked for repaint or in error will be repainted. Useful when switching between window panels.
S	Force the static data to be redisplayed, exactly as on the initial display of the panel. All fields are automatically painted.
W	Similar to **O** and **I** combined. Used mostly for updating fields in a separate window panel, and then returning to another window panel.
X	Similar to **O** and **S** combined.

Note:

The menu isn't refreshed for values **O**, **R**, *and* **W**; *it would degrade performance.*

On return from ScreenIO:

Y	All user fields will be repainted (default).

Field Attributes

These fields allow you to change the colors of fields, protect[5] fields that are normally unprotected, mark a field in error, and a variety of other field-specific operations.

Hint:

The name of a field attribute is the name of your data field name, followed by a suffix indicating the function. You can also refer to attributes by field number, since we redefine the attributes as a table. It's easiest to use the field name, though, unless you want to handle your fields as a table.

5 As elsewhere in this manual, protected fields are for output only. You may not enter data into a protected field, and with the exception of field types P, R, and S, you may not Tab to a protected field.

*panel-**NAMED-ATTRIBUTES*** or *panel*-**USER-ATTRIBUTE-TABLE**

This area contains a set of three attributes for each of your data fields. You may refer to them by the field name with the appropriate suffix, or as the table element using the field number, which is useful if your panel is displaying a table of fields.

field-name-**P** or *panel*-**FIELD-PAINT (***field-number***)**

This field is used to tell ScreenIO whether the data in the field is to be redisplayed after you have modified it, to indicate an error, or which field should become the active field. It also returns information concerning validation and HOT fields.

On entry to ScreenIO:

C	Make this the active field when the panel is displayed. ScreenIO begins at the Home field and activates the first field in the tabbing order. This takes precedence over the field number in *panel*-CURSOR-FIELD.
E	Repaint the field in the error color, and, beginning at the Home field, position to the first error field on the panel. This overrides *all* other ways of setting the active field.
R	Repaint the field. Necessary if you changed the data in the field and *panel*-REPAINT-SCREEN is **I**, **N**, **O**, **R**, or **W**.
	Any other values are ignored.

On return from ScreenIO:

	LOW-VALUES (default). This results in no action on the subsequent call to ScreenIO.
H	HOT Field return was invoked from this field. This is very useful in conjunction with named attributes. It makes it easier to determine which field is the HOT field when there are many HOT fields on the panel.
V	Indicates that data in the field has passed the validation criteria. Validation is only applied when the user leaves a field by filling it or by tabbing out of it.

Note:

If the field is to be validated but is returned with low-values in this attribute, you cannot be assured that the data has passed your validation specifications. ScreenIO does not validate a field until the user fills or Tabs from the field.

If the field was both HOT and had validation data the H *will be returned instead of the* V. *ScreenIO won't return from a HOT field with validation unless the validation passed. So, H implies V as well…*

Here's a trick you can use to determine which field was the active field (by name) when ScreenIO returned to your program. The nice thing is that you can use the attribute *field-name*-P instead of the field number.. Be sure to place this code AFTER any tests for validation indicators (V) or HOT field indicators (H).

```
MOVE '*' TO panel-FIELD-PAINT (panel-CURSOR-FIELD).

IF field-name-P = '*'
  do what is necessary......etc.
```

field-name-**C** or *panel*-**FIELD-COLOR (***field-number***)**

This field is normally zero, which causes the field to be displayed using the color defined in the panel editor.

On call to ScreenIO:

If you wish to change the color of a field when it is displayed, move the color value to this field.[6] If the field is marked in error using *field-name*-P, the error color will take precedence until the user leaves the field.

On return from ScreenIO:

Unchanged.

field-name-**O** or *panel*-**FIELD-OPTION (***field-number***)**

This field will protect a normally unprotected field.[7] It is normally LOW-VALUE.

On call to ScreenIO:

P	Protect the field.
Q	Protect the field and disable predisplay exit.
S	Disable a predisplay exit on the field.

On return from ScreenIO:

Unchanged.

6 See the table of color numbers which appears in the section on Color Control near the beginning of this chapter.

7 Sorry, you can't unprotect a type X (protected) field.

Active Field/Caret Position/Mouse Information

These fields return the caret position to your program and allow you specify the active field and caret position the next time the panel is displayed.

First, a word about terminology. In the old MS-DOS days, the thingy that indicated where you were about to type was known as the cursor. Then Windows came along and people started calling the mouse thingy the cursor and the-what-used-to-be-the-cursor became known as the insertion point, or caret. We had nothing to do with it!

Some of ScreenIO's field names hearken back to its MS-DOS heritage, hence the "cursor" part of some field names. We otherwise use the Windows terminology.

panel-**CURSOR-FIELD**

On call to ScreenIO:

This is the number of the field[8] which will become the active field.

If you have marked errors using E in *field-name*-P or used C in *field-name*-P for caret positioning, they will take precedence over this field.

Note:

It's easiest to specify the active field using named attributes, because you don't have to know the field number. If the field you specify is a protected field, ScreenIO will automatically move to the first unprotected field or selector field that follows in the tabbing order.

On return from ScreenIO:

Set to the number of the field that will be active the next time you call ScreenIO to redisplay the panel. If the user returned by hitting a function key, it will be the field that was active when the user pressed the key or returned because of a mouse action.

HOT field returns are more subtle. If ScreenIO returned because the user tabbed out of a HOT field, this won't always be the same as the HOT field; it will be the field that is the *destination* of the tab. You can determine which direction the user tabbed when leaving the HOT field. Compare *panel*-CURSOR-FIELD to the HOT field number extracted from *panel*-EXIT-KEY.

If the panel has no unprotected fields, *panel*-CURSOR-FIELD will be 0.

[8] By field number, we mean the sequential field number, as they are listed in the copybook. The fields are also numbered in the copybook in columns 1-6 so you don't have to count them to determine the field number.

panel-**OFFSET-THE-CURSOR**

This field is used in conjunction with *panel*-CURSOR-OFFSET to tell ScreenIO that you want the caret to be somewhere other than the first byte of the field. For example, to call attention to a specific word in a line of text, you could instruct ScreenIO to position the caret at the first character position of that word.

On the call to ScreenIO:

Y	The caret will be offset *panel*-CURSOR-OFFSET characters into the field when the panel is displayed.
N	The caret will be positioned at the first byte of the field.

Note:

The exception: If the panel name, field number, and offset are unchanged from the values they had on the return from ScreenIO, the caret will be placed in the same position it was when ScreenIO returned to your program.

The result - if your calling program changes nothing and just calls ScreenIO again, as would occur for an undefined function key, you don't have to worry about the caret position; it will stay where it was, which is what users normally expect.

On return from ScreenIO:

N	Default value.

panel-**CURSOR-OFFSET**

On the call to ScreenIO:

Set this field to the offset of the caret within the active field. Remember, this is an offset - 0 means the first position of the field (no offset), 1 means to offset the caret one character, which is position 2, etc. See *panel*-OFFSET-THE-CURSOR.

Note:

If this value exceeds the field length, it will be ignored and the caret positioned at the start of the field. Likewise, if you position the caret on a protected field, ScreenIO will tab to the next unprotected field and ignore panel-CURSOR-OFFSET.

On return from ScreenIO:

This field contains the offset of the caret within the current field. If the panel has no unprotected fields, it will be 0.

panel-**CURSOR-ROW,** *panel*-**CURSOR-COL**

On return from ScreenIO, these fields contain the row and column the caret was in, relative to the entire panel image.

panel-**MOUSE-ROW,** *panel*-**MOUSE-COL**

On return from ScreenIO, these fields contain the row and column the mouse was in, relative to the entire panel image.

Error Messages

These fields instruct ScreenIO to issue error messages.

panel-**MENU-MSG**

On call to ScreenIO:

> The text of a message to be displayed in the status bar (32-bit) or in a message box (16-bit) if you specify a value of 1 in *panel*-DISPLAY-OPTION.
>
> All ScreenIO error messages are handled the same way; they are removed when another error message is issued, or when the user hits a key or clicks the mouse.

On return from ScreenIO:

> Unchanged.

panel-**DISPLAY-OPTION**

On call to ScreenIO:

1	The contents of *panel*-MENU-MSG will be displayed and the Windows asterisk sound will be played.
nn	Your message numbered nn will be displayed, where *nn* is a number greater than one. You can modify the SCRMSG[9] subroutine (provided with ScreenIO) to contain the text of your error messages. The Windows asterisk sound will also be played.

On return from ScreenIO:

0	Default setting is restored.

[9] It could be convenient to utilize SCRMSG for your message text when your application will be utilized in a non-English language environment, since all of your messages reside in a single location. Translating and documenting the message text is relatively simple in this case.

Miscellaneous Fields

This group contains some fields that control the way that ScreenIO will display the panel, and others that carry information back to your program from ScreenIO.

panel-**NAME**

Name of this panel. Do not change this field.

panel-**NUMB-OF-FIELDS**

Number of fields on this panel. Provided for information only.

panel-**SHOW-WAIT-MSG**

Not used.

panel-**CURSOR-TYPE**

Specifies the style of caret to use for the panel unless a field specifies a different caret style. The value in this field is that which you selected in the panel editor.

On call to ScreenIO:

0	No caret.
1	Normal caret.
2	½ width caret.
3	¾ width caret.

On return from ScreenIO:

Unchanged.

panel-**SHIFT-STATES**

This one byte field contains the shift states of the Shift, Control, Alt, Num Lock, Scroll Lock, Insert, and Delete keys at the time the user pressed the function key that returned control to your program.[10]

Use the subroutine SCR_BYBI to extract these shift states. If the value returned for the key state is LOW-VALUE, the key is off (not pressed). Any other value indicates an on condition. The layout of the bits in this byte is shown below.

[10] This may well be different than the shift state at the current time, as the user may have let up on the shift keys since control was returned to your program. You can get the real-time shift state by calling SCR_SHFT.

1	Insert key.[11]
2	Caps lock.
3	Num lock.
4	Scroll lock.
5	Alt key.
6	Control key.
7	Left shift key.
8	Right shift key.[12]

panel-**OFFSET-ROW and** *panel*-**OFFSET-COL**

These two fields (normally zero) can be loaded with the number of rows and columns that you want a window panel to be offset from its defined position.[13]

To use this feature, design your window panel in the upper left hand corner of the full screen editor (although it could be defined anywhere). You can then locate the window anywhere on the ScreenIO area at display time by simply specifying the offsets, which ScreenIO will add to the current locations.

If you want to shift a window one column right and one row down from its originally defined position, move 1 to these fields and display the window panel.

The feature is fail safe in that if you try to locate a window off of the ScreenIO area, ScreenIO will move it as far as possible within the bounds of its area. If you moved a large number to both offsets, the panel will be positioned in the lower right corner.

Runtime Color Control

These fields allow you to set or modify the colors[14] of various items on your panel. Background colors may range from 0 through 7, and foreground colors from 0 through 15. The composite background/foreground color number is a value from 00 through 255.

[11] The Insert status is the PC's internal status of this key, and not necessarily the status with regard to ScreenIO, since each application must maintain its our own, independent of the hardware status. Because of this, the hardware shift state of the Insert key isn't very useful.

[12] Unfortunately, it is not possible to detect which shift key was depressed in Windows versions earlier than Windows NT 4.0. If either Shift key is down, both bits are set ON.

[13] Negative offsets will be ignored. Also, these fields are named FILLER unless the panel is a window panel. Our example panel is not a window, so consequently these fields do not appear in the sample panel's copybook.

If you wish to calculate the color number yourself, this is the formula:

```
COMPUTE COLOR-NUMBER = (BACKGROUND-COLOR * 16) + FOREGROUND-COLOR.
```

If you want the background to be high intensity:

```
ADD 128 TO COLOR-NUMBER.
```

The possible color combinations are shown in the table below. As an example, to select a blue background with a bright white foreground text, select color 31.

	FOREGROUND NORMAL								FOREGROUND HIGHLIGHT							
Bkgnd	Blk	Blu	Grn	Cyn	Red	Mag	Brn	Whi	Blk	Blu	Grn	Cyn	Red	Mag	Yel	Whi
Black	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
Blue	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
Green	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
Cyan	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
Red	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
Magnta	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
Brown	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
White	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

panel-**BORDER-COLOR**

Unused.

panel-**BASE-COLOR**

The base color is the background color of the panel. It may be changed the first time you call ScreenIO to display a panel.

If you change this field on subsequent calls to ScreenIO it will be ignored, unless an intervening panel was called, or you specifically request that ScreenIO repaint the static data. This is done for efficiency reasons, since it is unnecessary to paint static data except on the initial display of a panel.

The base color is a numeric value from 0-8, where values 0-7 will clear the background before painting and 8 will not (which allows a window panel to visually overlay another panel).

[14] These colors are selected in the panel specifications portion of the panel editor.

Note:

A non-windowed panel may not *be converted into a windowed panel by specifying an 8 for a background color. Furthermore, you cannot change the base color of a window panel.*

panel-**ACTIVE-COLOR**

The active color is the color of the active field; the field containing the caret.

A field is painted the active color when you tab to the field. This draws attention to the field and makes it easy for users to find the caret. This variable is normally set in the panel editor but it can be changed by your program. Allowable values are 0 through 255[15].

panel-**ERROR-COLOR**

The color that will be used for fields that you marked in error, or which do not pass ScreenIO's validation. Allowable values are 0 through 255.

panel-**MENU-COLOR**

The color that the menu will be painted when it is displayed. The values are 0 through 255.

Menu Contents

These fields control what is displayed in the menu line. The bottom line of the panel is always reserved by ScreenIO for displaying menus.

You may not place data fields on this line.

panel-**MENU-LINE**

This contains the text of your menu. It may be changed anytime by your program. The VALUE clause in the copybook contains the text that you entered in the panel editor.

panel-**INDICATORS**

Not used.

[15] See the table above.

ScreenIO returned because…

panel-**CLOSE-KEYVALUE**

On call to ScreenIO:

If greater than zero, this enables the File/Exit or Close option in the Windows system menu *for this panel only*[16]. This way you can prevent your users from selecting File/Exit except from certain panels.

You can set *panel*-CLOSE-KEYVALUE to any numeric value. If the user selects File/Exit or Close from a Windows system menu, ScreenIO will return this value to your program in *panel*-EXIT-KEY. *ScreenIO will not terminate your program.*

On return from ScreenIO:

Unchanged.

panel-**EXIT-KEY**

On return from ScreenIO:

A numeric value that tells you why ScreenIO returned to your program; usually the user pressed a function key or selected a menu option using the mouse.[17]

There are three cases where this may not be a function key value.

- You have specified HOT fields on your panel. When the user leaves a hot field, control will be immediately returned to your program. *panel*-EXIT-KEY will be set to 1000 plus the hot field number.
- You set a value other than zero for *panel*-CLOSE-KEYVALUE or SCREENIO-CLOSE-VALUE, and the user selects File/Exit from the Windows system menu, ScreenIO will return the value of *panel*-CLOSE-KEYVALUE or SCREENIO-CLOSE-VALUE.
- A type R field was active and the user pressed a character key. In this case, *panel*-EXIT-KEY is greater than 2048. The character pressed can be determined by examining the first byte of *panel*-EXIT-KEY. Be sure to test for both[18] upper and lower case!

```
01  EXIT-KEY-C.
   05  CHARACTER-PRESSED            PIC X.
```

[16] If you set a global SCREENIO-CLOSE-VALUE in your initialization call, it will override any individual panel settings (that is, the user will be able to close your application from any panel).

[17] The possible values for function keys are contained in the copybook KEYVALUE.COB.

[18] Or use SCR_CASU to force the character to uppercase before testing uppercase values.

```
        05  FILLER                      PIC X.
    01  EXIT-KEY-N REDEFINES EXIT-KEY-C  PIC S9(4) COMP-5.
                                  .
                                  .
                                  .
        IF panel-EXIT-KEY > 2048
           MOVE panel-EXIT-KEY TO EXIT-KEY-N

   * ----- Check both upper and lower case characters!!!

           IF CHARACTER-PRESSED = ‘a’ OR ‘A’.....
```

panel-**SCAN-CODE**

On return from ScreenIO:

nn	Scan code of the key that was pressed.
X’01’	ScreenIO timed out (you had set a timeout).
X’FE’	User selected an item from the Windows menu.
X’FF’	Mouse action other than Windows menu.

User Exits

See Advanced Programming. User exits are subroutines that ScreenIO calls dynamically at certain points. Very few applications require user exits, and the vast majority of ScreenIO users will never need to write one.

panel-**ARROW-EXIT-NAME**

Name of the .DLL called by ScreenIO when an arrow key is pressed.

panel-**PRE-EXIT-NAME**

Name of the .DLL called by ScreenIO before displaying each character for fields having a processing option specifying a predisplay exit.

panel-**POST-EXIT-NAME**

Name of the .DLL called by ScreenIO following every keystroke when in fields having a processing option specifying a postkeystroke exit.

Sound Control

These fields are a vestige of the DOS product. They have no effect in Windows. This version of ScreenIO is silent when a panel is presented, and will play the Windows asterisk sound when an error occurs.

panel-**PRESENT-FREQ**

Unused.

panel-**PRESENT-TIME**

Unused.

panel-**ERROR-FREQ**

Unused.

panel-**ERROR-TIME**

Unused.

Your Data Fields

panel-**USER-FIELDS**

These are the data fields that you defined via the field editor when you created your panel.

Note:

You can not change the length or order of any field by changing the copybook. You must use the panel editor to revise the field definitions.

On call to ScreenIO:

The data that will be displayed in your fields.

Hint:

If you elect to improve efficiency of your application by moving N *to panel-REPAINT-SCREEN, remember to mark the changed fields for repaint by moving* R *to the corresponding* field-name-*P. If you don't, your changed data will not be reflected on the screen next time you call ScreenIO. It will mysteriously appear, however, when the user tabs to the field (when it's repainted the active color).*

On return from ScreenIO:

Fields contain the data keyed into them by the user. If the fields were masked, the mask insertion characters are stripped.

Numeric values are returned in a valid numeric format.

Character fields are not normally justified, but are blank filled. The mask you select will determine whether character is justified when it is returned to you (as well as how it is justified when it is displayed).

Programming Techniques

As in all things, there are easy ways and there are hard ways to do what you want to do. Here's the way we use ScreenIO, which we think is the easy way!

The ScreenIO Loop

After you do any initialization, you display the panel by calling ScreenIO. Usually, you build a little loop[19] to handle your panel. When ScreenIO returns, you examine *panel*-EXIT-KEY to see what caused ScreenIO to return to your program (usually the user selected a menu option by pressing a function key or using the mouse).

Next, you probably edit the data and either mark the errors and redisplay the panel, or process the data. When you're done with that, you probably display the panel again, ready for the next data the user will enter.

If you want to use the Windows standard Alt-F4 to close your application, your ScreenIO loop will look like this. You could also specify the value of Alt-F4 in the *panel*-CLOSE-KEYVALUE, or in the initialization field SCREENIO-CLOSE-VALUE, if you want the Close option to be enabled on your Windows system menu.

Your program will remain in this loop until ScreenIO returns an Alt-F4. When that happens, you will fall through the PERFORM loop because of the UNTIL clause. At that point, your program will clean up and quit.

```
PERFORM WITH TEST AFTER
      UNTIL panel-EXIT-KEY = ALT-F4

  CALL 'SCREENIO' USING  panel-PANEL
                         panel-PASS-TO-EXIT
                         panel-WORK-S
                         panel-WORK-D

  EVALUATE panel-EXIT-KEY
    WHEN ENTER-KEY
      PERFORM enter-key-paragraph
    WHEN F1
      PERFORM F1-key-paragraph
    WHEN F3
      PERFORM F3-key-paragraph
```

[19] This is not in any way related to a Windows message loop. Forget about Windows issues, we take care of them within ScreenIO!

```
              WHEN OTHER
                CONTINUE
            END-EVALUATE

          END-PERFORM.
     *
     *----Tell ScreenIO to shut down before STOP RUN (IMPORTANT!)
     *
          CALL 'SCREENIO' USING    FUNCTION-SHUTDOWN
                                   FUNCTION-SHUTDOWN
                                   FUNCTION-SHUTDOWN
                                   FUNCTION-SHUTDOWN.
      STOP RUN.
```

Make a Subroutine for Each Panel!

We modularize our code for two reasons; simplicity and flexibility. We generally code a separate subroutine for each panel. We do this because all logic associated with the panel (such as file I/O, computation, etc.) is entirely contained in the subroutine, and if we want to invoke the panel from several places, it requires only a simple CALL to the subroutine.

It's far easier to debug ten 400 line subprograms than one 4,000 line monolith of a program.

Browsing a File Easily

We generally define browse panels using Selector fields; one field per line. Set the Return Up-Down Arrow option for ONLY the first and last fields. We redefine *panel*-USER-FIELDS so that we can easily subdivide the fields. That way, the entire line is highlighted when the user tabs to it. We do it this way:

```
*    This is the panel copybook...
             .
             .
             .
        05 panel-USER-FIELDS.
Fld001   10 FILLER                   PIC X(76).
Fld002   10 FILLER                   PIC X(76).
             .
             .
Fld044   10 FILLER                   PIC X(76).

*    This is YOUR code following the COPY panel statement:

        05 FILLER REDEFINES panel-USER-FIELDS.
         10 CUST-DATA OCCURS 44 INDEXED CUST-INDEX.
          15 CUST-LAST-NAME          PIC X(18).
          15 FILLER                  PIC X.
          15 CUST-FIRST-NAME         PIC X(12).
          15 FILLER                  PIC X.
          15 CUST-STATE              PIC XX.
          15 ... and so on
```

When you first display the browse, you read 44 records and display them in the 44 fields on your panel. Because we redefined the fields as shown above, you can easily format the data into the panel area by using your redefinition of the ScreenIO *panel*-USER-FIELDS area. The nice thing about this is, you only have to define 44 fields to ScreenIO, and you don't have to strain your brain to figure out which field number refers to which record. There is a simple one-to-one correspondence of panel field number to data record occurrence.

Let's say that you define a 44 element table in your program (not part of the panel definition) that contains the keys to those records, too. That way, if the user highlights the record and hits Enter (they're selector fields, remember?) you can just look up the key in the table based on the field number and then read the record directly to display the entire record in another panel. Easy.

The user will most likely use the up and down arrow keys to move up and down. Now, what happens when the user is in the first (or last) field and they press the up or down arrow? Because you set Return Up-Down Arrow on these fields, ScreenIO will return to your program with either CURSOR-UP or CURSOR-DOWN in *panel*-EXIT-KEY.

Your program then performs a little inline loop to shift all of the records (and their keys) up or down one field. Next, it reads NEXT or PRIOR just one record, depending on the direction of the arrow key, loads that record in the first (or last) field, and redisplays the panel. You only do one I/O operation, and your user can easily scroll up and down through a file. Nothing to it.

Here's some sample code. It is NOT complete but it should give you the general idea... We'll leave the status checking and other details to you:

```
        PERFORM WITH TEST AFTER
              UNTIL panel-EXIT-KEY = ALT-F4

          CALL 'SCREENIO' USING  panel-PANEL
                                 panel-PASS-TO-EXIT
                                 panel-WORK-S
                                 panel-WORK-D

          EVALUATE panel-EXIT-KEY
            WHEN ENTER-KEY
              PERFORM display-selected-record
            WHEN CURSOR-UP
              PERFORM SCROLL-BACKWARD-1
            WHEN CURSOR-DOWN
              PERFORM SCROLL-FORWARD-1
            WHEN PAGE-UP
              PERFORM SCROLL-BACKWARD-PAGE
            WHEN CURSOR-DOWN
              PERFORM SCROLL-FORWARD-PAGE
            WHEN OTHER
              CONTINUE
          END-EVALUATE

        END-PERFORM.
   *
   *----Fell through because of Alt-F4; close files & quit
   *
        CALL 'SCREENIO' USING    FUNCTION-SHUTDOWN
                                 FUNCTION-SHUTDOWN
                                 FUNCTION-SHUTDOWN
                                 FUNCTION-SHUTDOWN.
    STOP RUN.
```

```
*
*----Performed routines
*
 SCROLL-BACKWARD-1.
*    Position the file to the first record using our key table.
     MOVE KEY-ENTRY (1) TO CUSTOMER-RECORD-KEY.
     START customer-file KEY = CUSTOMER-RECORD-KEY.
*                          Shift the panel lines down one
     IF START-operation was successful
       PERFORM
             VARYING KEY-INDEX CUST-INDEX FROM 1 BY 1
                      UNTIL CUST-INDEX > 43
         MOVE CUST-AREA (CUST-INDEX) TO CUST-AREA (CUST-INDEX + 1)
         MOVE KEY-ENTRY (KEY-INDEX) TO KEY-ENTRY (KEY-INDEX + 1)
       END-PERFORM
       READ customer-file PRIOR RECORD
       MOVE customer-record-fields TO CUST-fields (1)
       MOVE CUSTOMER-RECORD-KEY TO KEY-ENTRY (1)
     ELSE
       MOVE 1 TO panel-DISPLAY-OPTION
       MOVE 'Beginning of the file!' TO panel-MENU-MSG.
*
 SCROLL-FORWARD-1.
*    Position the file to the last record using our key table.
     MOVE KEY-ENTRY (44) TO CUSTOMER-RECORD-KEY.
     START customer-file KEY = CUSTOMER-RECORD-KEY.
*                          Shift the panel lines down one
     IF START-operation was successful
       PERFORM
             VARYING KEY-INDEX CUST-INDEX FROM 2 BY 1
                     UNTIL CUST-INDEX > 44
         SET KEY-INDEX TO CUST-INDEX
         MOVE CUST-AREA (CUST-INDEX) TO CUST-AREA (CUST-INDEX - 1)
         MOVE KEY-ENTRY (KEY-INDEX) TO KEY-ENTRY (KEY-INDEX - 1)
       END-PERFORM
       READ customer-file NEXT RECORD
       MOVE customer-record-fields TO CUST-fields (44)
       MOVE CUSTOMER-RECORD-KEY TO KEY-ENTRY (44)
     ELSE
       MOVE 1 TO panel-DISPLAY-OPTION
       MOVE 'End of file!' TO panel-MENU-MSG.
*
 SCROLL-BACKWARD-PAGE.
*    Position the file to the first record using our key table.
     MOVE KEY-ENTRY (1) TO CUSTOMER-RECORD-KEY.
     START customer-file KEY = CUSTOMER-RECORD-KEY.
*                          Read the previous 44 records (or to EOF)
     IF START-operation was successful
       PERFORM
             VARYING KEY-INDEX CUST-INDEX FROM 44 BY -1
                     UNTIL CUST-INDEX < 1
         READ customer-file PRIOR RECORD
         MOVE customer-record-fields TO CUST-fields (CUST-INDEX)
         MOVE CUSTOMER-RECORD-KEY TO KEY-ENTRY (KEY-INDEX)
       END-PERFORM
     ELSE
       MOVE 1 TO panel-DISPLAY-OPTION
       MOVE 'Beginning of file!' TO panel-MENU-MSG.
*
 SCROLL-FORWARD-PAGE.
*    Position the file to the last record using our key table.
     MOVE KEY-ENTRY (44) TO CUSTOMER-RECORD-KEY.
     START customer-file KEY = CUSTOMER-RECORD-KEY.
*                          Read the next 44 records (or to EOF)
     IF START-operation was successful
       PERFORM
             VARYING KEY-INDEX CUST-INDEX FROM 1 BY 1
                     UNTIL CUST-INDEX > 44
```

```
      READ customer-file NEXT RECORD
      MOVE customer-record-fields TO CUST-fields (CUST-INDEX)
      MOVE CUSTOMER-RECORD-KEY TO KEY-ENTRY (KEY-INDEX)
    END-PERFORM
  ELSE
    MOVE 1 TO panel-DISPLAY-OPTION
    MOVE ‘End of file!’ TO panel-MENU-MSG.
```

Now, if your application needed to browse the customer file from lots of places, you could bundle this up as a subroutine that was passed the starting key, and returned the key of the record selected (if the user selected a specific record). It saves lots of work, and once you get the subroutine working it’s a snap to apply the same general technique to other files and applications.

Handling User Errors

One of the most common aspects of interactive applications is telling users (nicely) that they screwed up! The error could be anything from pressing the wrong key (an alphabetic key when in a numeric field, for example), to entering an illegal value in a field (such as a numeric value which is out of range).

ScreenIO always indicates internally detected errors, such as validation errors or an attempt to enter data of the wrong type in a field, as follows:

- The field in error is repainted the error color.
- The caret is positioned to the field in error.
- The error message is displayed.
- The Windows asterisk sound is played.

This adds up to an unmistakable indication of an error. You can easily issue your messages in a similar way.

Hint:

If you edit your fields from bottom to top, all of the erroneous fields will be painted the error color, but ScreenIO will display the last message you loaded to panel-*MENU-MSG; the one for the first erroneous field on the panel.. This is the easiest way to perform edits against all the fields on a panel.*

Marking fields in error

You can indicate that field(s) are in error after you have performed an edit in your program:

- Move E to the corresponding *field-name*-P attribute of all fields in error.
- Call ScreenIO.

All the fields you marked are displayed using the error color; the first will be the active field. You can also issue a message using *panel*-MENU-MSG and/or *panel*-DISPLAY-OPTION.

Issuing error messages

- Move your message text to *panel*-MENU-MSG.
- Move 1 to *panel*-DISPLAY-OPTION.
- Call ScreenIO again!

It's that simple. ScreenIO takes care of the rest for you.

Issuing messages by number

You can also store your error messages in the SCRMSG subroutine, in which case all you need do is tell ScreenIO which message number to issue.

- Move your message number to *panel*-DISPLAY-OPTION.
- Call ScreenIO.

Other methods

You obviously can design your panels to include a field or fields that you use strictly for displaying error messages. This is occasionally required when you must issue long or complex messages, or error messages that must remain on the screen for some period of time.

You can also use a general-purpose window panel to display error messages; if the window panel is contained in a subroutine for the purpose, all you have to do is to call your error subroutine to display your message.

What NOT To Do

Do not use the COBOL DISPLAY verb to directly issue a message from your program. After all, you got ScreenIO to do your screen management for you. Why on earth would you want to revert back to ACCEPTs and DISPLAYs??

Unsupported Function Keys

ScreenIO returns all function keys. We usually just ignore the ones we're not interested in. To the user, it looks as though nothing happened. Which is precisely correct!

Ignoring Unsupported Keys

In this example, assume we are displaying a panel which supports only three function keys, F1, F2, and Alt-F4. If another key is pressed, it is ignored by simply calling ScreenIO again.

```
PERFORM WITH TEST AFTER
      UNTIL panel-EXIT-KEY = ALT-F4

  CALL 'SCREENIO' USING panel-PANEL
                        panel-PASS-TO-EXIT
                        panel-WORK-S
                        panel-WORK-D

  EVALUATE panel-EXIT-KEY
    WHEN F1
      PERFORM F1-key-paragraph
    WHEN F2
      PERFORM F2-key-paragraph
    WHEN OTHER
      CONTINUE
  END-EVALUATE

END-PERFORM.
```

Issuing an Error Message

Some users may want to issue a message to the effect that the user pressed the wrong key. It's simple; just insert a couple of lines for the WHEN OTHER condition.

```
PERFORM WITH TEST AFTER
      UNTIL panel-EXIT-KEY = ALT-F4

  CALL 'SCREENIO' USING panel-PANEL
                        panel-PASS-TO-EXIT
                        panel-WORK-S
                        panel-WORK-D

  EVALUATE panel-EXIT-KEY
    WHEN F1
      PERFORM F1-key-paragraph
    WHEN F2
      PERFORM F2-key-paragraph
    WHEN OTHER
      MOVE 1 TO panel-DISPLAY-OPTION
      MOVE 'Sorry, unsupported option!' TO panel-MENU-MSG
  END-EVALUATE

END-PERFORM.
```

Using ScreenIO's Internal Validation

You can specify validation data for a user field when you define it in the panel editor. ScreenIO will compare the contents of the field with the validation data you specified when the user fills or tabs out of a validated field.

Caution:

Your field is only validated if the user filled it or tabbed out of it. If the user didn't fill in the entire field, and hit a function key, ScreenIO will not edit the data and you can't count on its validity. This could cause you difficulties, no?

Check the *field-name*-P attribute field to determine if the field was validated. If valid, it will contain a V. If it was also a HOT field, it will contain H which indicates that it was a HOT field return, and that implies that validation took place as well (you couldn't return from a HOT field with validation unless the validation passed).

See the discussion of the Must Validate option below…

Field Processing Options

You can set a variety of field processing options for your data fields. These are explained in the section on the panel editor, but there are a few additional points:

HOT Field

HOT fields cause ScreenIO to return to your program when the user either fills a field or tabs out of the field. A HOT field which has No Autoskip selected will return to your program as soon as the field is filled, as well as when the user presses Tab.

You can use HOT fields to perform inline edits, to activate other fields depending on the content of other fields, or to, say, read a record and display the data when the user enters the key of the record and fills or tabs out of the key field. We use HOT fields all over the place.

ScreenIO returns to your program indicating a HOT field has been encountered in two ways:

- The control field *panel*-EXIT-KEY will contain a value of 1000 plus the field number instead of a function key value.
- The paint attribute *field-name*-P will contain H (for HOT return).

If ScreenIO returned because the user tabbed out of a HOT field, *panel*-CURSOR-FIELD won't necessarily be the same as the HOT field; it will be the field that is the *destination* of the tab. You can therefore determine the direction of the tab: Just compare *panel*-CURSOR-FIELD to the HOT field number extracted from *panel*-EXIT-KEY.

HOT Modify

Same as HOT, but ScreenIO only returns if the user modifies the field. The field is modified if the user types a character, clears the field with Ctrl-End, or presses Backspace or Delete.

This is desirable in panels with lots of HOT fields primarily used for editing; it improves performance when tabbing from field-to-field since unnecessary processing is suppressed.

Must Validate

ScreenIO won't return until *all* fields are valid if you specify Must Validate on them.

GREAT, you think, that solves the hole in validation mentioned above… There are a couple of things you should think about, though, before you get crazy with Must Validate:

What happens if your user doesn't know the correct entry for one of the fields on a panel? Sure, they know that a value of 1-4 is *valid*, but they don't know which answer is *true* right now. The only problem is, they can't get out of your panel because they're completely trapped by your Must Validate field.

Or, what if your user accidentally gets to a panel with Must Validate fields, and all they want to do is leave? They're stuck until they enter valid data in the fields.

Furthermore, if your panel has HOT fields as well as Must Validate fields, ScreenIO may issue validation error messages for fields that the user has not yet been able to fill, due to the fact that HOT fields will invoke Must Validate testing when the HOT field tries to return! Generally, if your application uses HOT fields on a panel, you should avoid Must Validate, except for those cases where the field must absolutely be validated.

HINT:

See Function 11, which lets you pass ScreenIO a table of values for panel-*EXIT-KEY that can bypass the Must Validate logic. For example, you could exempt HOT fields or Alt-F4 from the Must Validate trap using this function.*

Required

Much of the commentary regarding Must Validate also applies to the Required option. This option requires that a field contain other than spaces or LOW-VALUE.

Clear First

The Clear First option will clear the field when the first character the user types is in the first position of the field. If the character is typed in any other position of the field, it will simply replace the character in that position.

This option is useful where your users will be entering data into fields which already contain data, such as when updating a record from a file.

Return Up-Down Arrow

Normally, the up and down arrow keys are treated by ScreenIO as Shift-Tab and Tab. This option causes ScreenIO to return CURSOR-UP when the up arrow key is pressed, and CURSOR-DOWN when the down arrow key is pressed.

Efficiency Considerations

ScreenIO displays the contents of *all* of your data fields every time you call it unless you tell it otherwise. This is, of course, unnecessary if the contents of those fields have not been changed by your program. You only need to update the fields whose contents have changed.

ScreenIO works this way because the default setting of *panel*-REPAINT-SCREEN is Y, which tells ScreenIO to redisplay *all* user fields. We implemented ScreenIO this way to eliminate problems for inexperienced users, who might not remember to mark a field to be repainted after moving data to it in their program.[20]

If you set *panel*-REPAINT-SCREEN to N and you moved data to a field but neglected to move R to the field's attribute *field-name*-P, the new data would not be displayed on the screen when you next called ScreenIO. This could obviously cause confusion for beginners, which is why the default setting for *panel*-REPAINT-SCREEN is Y.

Although the default setting repaints all fields each time ScreenIO redisplays a panel, it's more efficient to repaint only those fields changed by your program. Do this by moving R to *field-name*-P or *panel*-FIELD-PAINT (*field-number)* of each field you change, and moving N to *panel*-REPAINT-SCREEN.

Note:

This is even more important when you use HOT fields. HOT fields should always be processed in the most efficient manner possible so as to minimize the delay while moving from one field to the other.

Removing Window Panels

You don't have to; when you have finished with a window panel, simply redisplay the screen that was "behind" the window.

It's important to realize that windows are not really removed; ScreenIO simply redisplays underlying panel. What was underneath is not restored, but rather completely regenerated.

Note:

One rule applies to windows; each level of window that you intend to remove must lie entirely within the underlying panel*. If it does not, you will not remove all of the image when you redisplay the underlying image, which looks sloppy.*[21]

Finishing Up

You must call ScreenIO using Function 15 before you terminate your application.

[20] This does not affect fields changed by the operator. The condition we are addressing occurs upon a redisplay of the current panel after some fields have been changed by your program.

[21] If you absolutely must break this rule, see Subroutines & Functions for instructions on saving and then restoring screen images. We really don't recommend it, however.

Using tables in your panel

DON'T modify the copybook!

Some users think they have to modify the copybooks generated by the panel editor in order to change the way things are laid out or to alter the definitions of fields. Or, they want to treat the fields on the panel as an array or a table.

ScreenIO provides for this automatically, in that your data fields always occur at the end of the copybook, and a data name is provided specifically for allowing you to redefine this area. To manipulate the fields of a panel as a table, you simply code a REDEFINES clause immediately following the COPY statement that copies in your panel copybook:

```
        WORKING-STORAGE SECTION.

            COPY panel.

      *---------------------Redefine the panel fields as an array:

            05 MY-TABLE REDEFINES panel-USER-FIELDS.
               10  TABLE-FIELD OCCURS .....
```

Some programmers have been known to move the copybook into their programs rather than using the COPY statement! It should be fairly obvious that this is a mistake you will quickly come to regret.

Advanced Programming

Distributing Your Applications

16-bit executables will contain all necessary ScreenIO runtime support, which will be included from the ScreenIO object library SCREENIO.LIB when you LINK your program.

32-bit executables use runtime support contained in SCRWIN32.DLL, which in turn requires the file CARCLW60.DLL. We implemented SCRWIN32.DLL as a standard Windows .DLL, using the CA-Realia Workbench 3.0 COBOL development environment.

Your 32-bit applications require support contained in the following .DLL files, which must be distributed[1] with your application:

SCRWIN32.DLL	(32-bit ScreenIO runtime)
CARCLW60.DLL	(32-bit CA-Realia runtime)
CARFSW20.DLL	(32-bit CA-Realia file system runtime; only required if you use ScreenIO's Event Logging facility)

[1] Our agreement with Computer Associates allows you to redistribute the CA-Realia runtime .DLL files with your derivative ScreenIO applications.

Large Applications

Small applications may be distributed as a single, monolithic .EXE module. Large applications, however, are generally structured as a main program that calls a bunch of subprogram .DLLs dynamically. You may CANCEL these .DLLs when you are finished with them, thereby recovering the memory they were using.

Each .DLL may consist of a group of statically linked subroutines.

COBOL 101: Static versus Dynamic CALLs

For the benefit of those of you unfamiliar with dynamic calls, here's a little primer.

Static CALLs

CALL '*literal'*	(*'literal'* -in quotes- is the subroutine name)

When you use the static form of the CALL, your compiler generates code that causes the linker to include the subroutine's .OBJ module in your executable (.EXE or .DLL) module. There are no external references that must be resolved at runtime; everything your program needs is bundled into the executable module.

Some compilers allow you to specify an option which overrides this syntax and generates all CALLs as dynamic calls. This is a really bad idea; you're much better off maintaining some control over how your application is structured. Read on.

Dynamic CALLs

CALL *dataname*	(*dataname* is defined in WORKING-STORAGE as a PIC X(8) variable containing the subroutine's name)

When you use the dynamic form of the CALL, your compiler generates code that will attempt to load a .DLL of the name specified in *dataname* when you run your program and it encounters the CALL statement. Your application will not be contained in a single .EXE file, it will consist of the main program .EXE file, plus the .DLLs that it calls.

This strategy has a couple of nice features; if you have to upgrade just part of your application, you usually only have to distribute the new .DLL; not the entire application. The other thing that can be nice is that you can use the COBOL CANCEL verb to recover the memory that was used by the .DLL when you're finished with it.

The .DLL subroutines may themselves contain static calls to other subroutines. That's fine, it just means that the linker will include those subroutines in the .DLL.

One thing to consider, however, is that if several .DLLs call the same subroutine statically, each of the .DLLs will have its own copy of that statically called module linked in. If these .DLLs are in memory at the same time, you will have multiple copies of that statically called subroutine in memory, which wastes memory.

If the statically called subroutine is ScreenIO and you have several copies in memory, you will have other problems, too. Keep reading!

ScreenIO and Dynamic CALLs

If your application consists of multiple .DLLs that are calling ScreenIO, you must contrive to have them share the same copy of ScreenIO. You can readily do this by making ScreenIO available as a .DLL, and then calling it dynamically: If you have multiple copies of ScreenIO linked into different .DLLs, *you will have problems*, because each copy of ScreenIO is unaware of the others. To avoid this code your programs as shown below.

Write this small program that calls ScreenIO statically and link it as a .DLL. When your other programs need to call ScreenIO, they should call this module (SCRNIO.DLL) *dynamically*, instead of calling ScreenIO *statically*.

You may use this program *exactly* as it is shown below.[2]

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. SCRNIO.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. IBM-PC.
    OBJECT-COMPUTER. IBM-PC.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    LINKAGE SECTION.
    01  D1                      PIC X.
    01  D2                      PIC X.
    01  D3                      PIC X.
    01  D4                      PIC X.

    PROCEDURE DIVISION USING    D1 D2 D3 D4.
    STARTUP.

   *    Call ScreenIO statically in this module ONLY!
   *    Note the quotes around the subroutine name ‘SCREENIO’!

        CALL 'SCREENIO' USING   D1 D2 D3 D4.
        GOBACK.
```

Everywhere else, to call ScreenIO, do this:

```
    WORKING-STORAGE SECTION.
        .
    01  SCREENIO                PIC X(8) VALUE ‘SCRNIO’.
        .
        .
    PROCEDURE DIVISION.
        .
```

[2] Yes, we know the one byte arguments in the LINKAGE SECTION are too short. Trust us; it works and is perfectly legal because only the *addresses* of arguments are passed in a CALL, not the data.

```
*    Dynamically call SCRNIO.DLL to share one copy of ScreenIO
*    Note the lack of quotes around the data name SCREENIO!

     CALL SCREENIO USING      panel-PANEL
                              panel-PASS-TO-EXIT
                              panel-WORK-S
                              panel-WORK-D.
```

Note: Because your SCRNIO program sees everything passing between your programs and ScreenIO, you could easily implement a system wide help facility, or intercept certain values of *panel*-EXIT-KEY to perform functions that you wanted to be available from *anywhere* within your system.

Supporting Other 32-bit Compilers

We presently distribute support for several compilers with ScreenIO. If you are using a compiler that we don't presently support, don't despair. You can do it yourself! It is fairly simple to call ScreenIO from any 32-bit COBOL compiler. Here's a little background, and how to do it if you have a compiler we don't support.

Compiler-specific information is located in the same subdirectory as the compiler-specific SCREENIO.LIB provided with ScreenIO. It resides in a file called LINKING.TXT.

The ScreenIO runtime, SCRWIN32.DLL, will work with any 32-bit language; it is self-contained. All you need to do is to call it in the right way. We do that, for the compilers we support, via the library SCREENIO.LIB.

SCREENIO.LIB contains an import library for the entry points that are exported by SCRWIN32.DLL, plus several stub programs (compiled using the target compiler) that properly pass the arguments to SCRWIN32.DLL. In order to make SCRWIN32.DLL more-or-less universal, we utilize the Windows system linkage conventions and pass the arguments BY REFERENCE.

The stub programs are:

SCREENIO.COB	ScreenIO main entry point
EDITMASK.COB	Program containing your edit masks; created by the Panel Editor using your mask definitions from EDITMASK.SEQ
SCRMSG.COB	Program containing your messages
SCR_UTIL.COB	Program that exports various Windows APIs

For example, when your program calls ScreenIO, it's actually calling a small stub called ScreenIO (using your compiler's native linkage conventions). ScreenIO in turn just calls SCRWIN32 using the Windows conventions. This way, you don't have to deal with nonstandard (to COBOL) calling conventions elsewhere in your application, plus, it is a method whereby any 32-bit language can call our SCRWIN32.DLL runtime. Here's a *portion* of ScreenIO illustrating the various calling methods for several compilers. We deleted some of the code in the actual stub for clarity:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SCREENIO.
*                         :-------------------------------------
* ------------------------: ScreenIO interface routine stub.
*                         :-------------------------------------
.
 LINKAGE SECTION.
 01  ARG1                     PIC X.
 01  ARG2                     PIC X.
 01  ARG3                     PIC X.
 01  ARG4                     PIC X.

 PROCEDURE DIVISION USING ARG1 ARG2 ARG3 ARG4.
*
 0001-CALL-SCREENIO-DLL.
.
* ------------------------: Pass-through call to the REAL
*                         : ScreenIO, which resides in
*                         : SCRWIN32.DLL.
*                         :
*                         : In order to use a .DLL for ScreenIO,
*                         : we make it callable using Windows
*                         : conventions.  This routine just
*                         : maps the native compiler calls to
*                         : Windows calls to ScreenIO...

*    If Micro Focus COBOL:

     CALL WINAPI 'SCRWIN32' USING BY REFERENCE      ARG1
                                             ARG2
                                             ARG3
                                             ARG4.

*    If Fujitsu COBOL:

*    CALL 'SCRWIN32' WITH STDCALL USING BY REFERENCE        ARG1
                                                     ARG2
                                                     ARG3
                                                     ARG4.

*    If CA-Realia COBOL 6.0:

*    CALL 'S_SCRWIN32' USING BY REFERENCE    ARG1
                                             ARG2
                                             ARG3
                                             ARG4.
     GOBACK
```

In this case, your stub would only include the CALL SCRWIN32 statement that was for your compiler.

When you have modified and recompiled these stub programs, place them in a copy of SCREENIO.LIB using the library manager LIB.EXE. Then link your programs as usual, including SCREENIO.LIB in your LINK command.

Call us if you have questions on porting ScreenIO to your 32-bit compiler. It generally is very easy to do.

Edit Masks

Edit masks are a feature that allows you to tell ScreenIO how to format your fields. Edit masks work by supplying ScreenIO with the specifications for displaying a field. These specifications have place holders to indicate where data characters should be placed, and insertion characters which will be displayed in the corresponding place in the output field.[3]

Character masks are evaluated from left to right, whereas numeric masks are evaluated from right to left (think about it). Masks should usually be longer than the field they are used with, and the excess length should appear at the end of the mask that is evaluated last. The panel editor will warn you if the mask is too short for the field when you select the mask. ScreenIO will only use as much of the mask as it needs; it's OK (and advisable) to make the mask longer than necessary.

For example,

> Assume we have a data item with a picture clause of 9(6)V99, a value of 123456.78, and an edit mask of ZZZ,ZZZ,ZZZ,ZZZ.99-.
>
> This mask instructs ScreenIO to substitute a digit for each 9 or Z. (We will explain the significance of each character in the mask in a moment).
>
> Beginning with the rightmost character of the mask, ScreenIO will display a space for the sign, then the digit 8, then 7, the decimal point (an insertion character), followed by 6,5,4 and then a comma (another insertion character), etc.
>
> This continues until the data is exhausted. The display will look like this:
>
> **123,456.78**

Data will be returned to your program with all insertion characters removed. In other words, you can rely on ScreenIO to ensure that your data field will contain valid data; numeric fields will always be numeric, just as you would expect.

Each edit mask is up to 82 bytes in length, and is constructed of three parts; a mask type character, a justify/fill character, and up to 80 bytes of mask text.

[3] You don't allow room for insertion characters in your COBOL PICTURE clause when you define the field in the panel editor. When you are specifying fields, use picture clauses long enough to hold only the data. The panel editor will automatically size the display field long enough to accommodate the insertion characters from the mask you specify.

Changing ScreenIO Edit Masks

These are the steps involved to change edit masks:

1. Modify EDITMASK.SEQ to contain your new mask data.

2. Run the panel editor and verify your changed masks are present by editing the fields that use the new masks. Then, generate the panel copybooks that use the new masks.

3. Go to the Copybook generator in the panel editor and create EDITMASK.COB.

4. Compile EDITMASK.COB, producing EDITMASK.OBJ.

5. Replace the EDITMASK.OBJ we provided in the SCREENIO.LIB file with your new EDITMASK.OBJ.

6. Compile and link your programs.

Now let's talk about how masks are implemented within the panel editor and ScreenIO in order to understand what you must do to change them.

Pay attention, this is important!

Your masks must be available to the panel editor so it can use them to calculate the size of the display area for a masked field, and so it can verify that you selected a valid mask for your field. The panel editor uses the file EDITMASK.SEQ for its mask data. EDITMASK.SEQ is an ASCII text file, and may be modified using an ASCII text editor.

The panel editor stores only the mask ***number*** in the panel copybook when you generate it.

When you run your program and call ScreenIO to display a panel, ScreenIO in turn calls the subroutine EDITMASK, passing it the mask number for the fields. EDITMASK passes the mask text for that mask number back to ScreenIO, which then formats the field.

Now, as you can guess, the text for the mask ScreenIO uses at runtime must correspond to the text that the panel editor used to generate the panel or you will cause a mask error. Therefore, you have to compile a version of EDITMASK that contains your masks and link it with your application. It's easy; here's how you do it:

First, you have to create an EDITMASK.COB subroutine that contains the masks in the EDITMASK.SEQ file used by the panel editor. That's easy, just select the Copybooks option from the main menu. Then, instead of generating a panel copybook, press F3 to create the source file EDITMASK.COB. The panel editor will place this file in the same directory as your panel copybooks.

The default EDITMASK subroutine that we supply resides in the ScreenIO object library SCREENIO.LIB. ScreenIO calls this subroutine statically when it needs mask information. When you link your program, the linker includes the EDITMASK object file from the library in your executable code.

In order to provide your new masks to ScreenIO at runtime, you have to replace the EDITMASK.OBJ file provided by us with your own.

First, compile the EDITMASK program you produced above, producing an EDITMASK.OBJ file.

Next, use a library manager to update the file SCREENIO.LIB with your new EDITMASK.OBJ. That is usually done as shown below, but you should check your compiler's documentation to verify the commands for your environment:

For 16-bit library managers (LIB and OPTLIB), the command is:

```
LIB SCREENIO.LIB -+EDITMASK.OBJ
```

For the 32-bit Microsoft library managers (LIB), the command is:

```
LIB SCREENIO.LIB EDITMASK.OBJ
```

Now when you link your program, the linker will include your updated EDITMASK.OBJ from the library, and when you run your program, your modified or new masks will be used.

Defining Character Masks

The first character indicates the mask type. The type indicator must be:

C	Character mask.

The second character specifies how data is to be justified. Allowable values are:

C	Display centered, and returned left justified.
R	Display right justified, and returned left justified.
L	Display left justified, and returned left justified.
c	Display centered, and returned centered.
r	Display right justified, and returned right justified.
l	Display left justified, and returned left justified. (Lower case "L")

The remaining characters define the mask itself.

(Blank)	Specifies the insertion of one character of data field at this position.
&&	Specifies the insertion of one blank space. If single ampersand is used, it is treated as any other insertion character.

All other characters are assumed to be insertion characters. Insertion characters are any characters (except special meaning characters) which will be inserted in the display data provided there are enough data field characters (digits) to extend beyond the position of the insertion character.

Examples of character masks follow:

CC

Centers the data within the field but returns data typed in the field left justified to your program.

CL * *

Left justifies the data and inserts an asterisk (*) in the third and seventh positions. (Two blanks precede the first asterisk, and three more are between them). So, if your data field contained <u>abcdefgh</u>, the display would contain the string <u>ab*cde*fgh</u>.

Defining Numeric Masks

The first character indicates the mask type. The type indicator must be:

N	Numeric mask.

The second character is a fill character. It will occupy the unused display positions to the left of the most significant numeric digit. For example, if your display field was five characters in length but you displayed a numeric value of 1, and if you specified the asterisk as the fill character, the display would show <u>****1</u>.

The remaining portion of the mask can contain these characters. You should make the mask long enough to handle the largest value that you could use; unneeded mask positions are simply ignored by ScreenIO:

9	Specifies the insertion of one character of data at this position.
Z	Same as 9, except that if the digit is a non significant zero, a blank will be substituted. This will suppress leading/trailing zeroes.
$[4]	Currency symbol will be inserted preceding the first digit. The currency symbol always adds one to the display length.

[4] The dollar sign is used here for purposes of explanation only. In actual practice, this may be any currency symbol as long as it is the currency symbol for the current country. If the dollar sign were used in Great Britain, for example, it would be treated like any other insertion character, and inserted in the position in which it appeared in the mask, rather than following the rule for processing currency symbols. ScreenIO applies the currency symbol based upon the country dependent information in the user machine.

	This symbol will be in the first (leftmost) position of the field unless there are multiple currency symbols, in which case it is treated like a Z, and leading zeros will not be displayed - except for the first leading zero position which will contain the currency symbol. This is known as a floating currency symbol.
-	Minus sign. Causes negative numbers to have a trailing minus sign (in the rightmost position of the field). Positive numbers will have a trailing blank. A sign always adds one to the required display length.
CR	Credit. Similar to minus, except that the characters CR are appended and display length increases by two.
+	Plus sign. Causes positive numbers to be displayed with a trailing plus sign. Negative numbers are displayed with a trailing blank. Again, the sign adds one to the length of the display field.
DB	Debit. Similar to a plus sign, except that the characters DB are appended; it adds two to the display length.
±	Plus/Minus sign means that the field will be appropriately signed if it is negative or positive, adding one position to its length. This option is specified with a single plus/minus character (ASCII 241, hexadecimal F1).
&&	Used to signify the insertion of one blank space. If single ampersand is used, it is treated as any other insertion character.

The mask must end with an ASCII 221 character.

Examples of numeric masks follow:

N ZZZ,ZZZ,ZZZ,ZZ9.99

Displays the data for a PIC 9(9)V99 field with a decimal point, thousand separators, and leading zero suppression. A value of **1234.56** will be displayed as **1,234.56**.

N $$$,$$$,$$$,$$9.99

Displays the data for a PIC 9(9)V99 field with a decimal point, thousand separators, and a floating currency symbol. A value of **1234.56** will be displayed as **$1,234.56**.

N*$$$,$$$,$$$,$$9.99-

Displays the data for a PIC 9(9)V99 field with a decimal point, thousand separators, and a floating currency symbol. Unused display positions are filled with asterisks. A value of **-1234.56** will be displayed as *******$1,234.56-**. Note that ScreenIO only displays trailing signs.

N (999)&&999-9999

Displays the data for a PIC 9(10) telephone number field. The number **1234567890** will be displayed as **(123) 456-7890**. A single space is inserted between the right parenthesis and the digit 4, as specified by the && marker.

Date Separator character (Field type 5 ONLY)

The first character indicates the mask type. The type indicator must be:

D	Date separator character (applies ONLY to field type 5).

The second character is unused.

The third character is the date separator that you want to use. In the United States, this will usually be the / character. ScreenIO will insert the date separator in the correct positions of the date field, based upon the country-specific date format.

The mask must end with an ASCII 221 character.

EBCDIC Data

Some compilers allow you to work in EBCDIC instead of the PC's usual ASCII coding scheme. ScreenIO handles this automatically.

User Exit Subroutines

ScreenIO allows you to regain control at various points within the processing cycle through the use of user exit subroutines. User Exits are .DLL files you create, which are called dynamically by ScreenIO at certain points in its processing.

Very few applications require user exits. Virtually all processing done in normal applications can be handled by the general facilities of ScreenIO. The use of exits is best left until you have a reasonable amount of experience with ScreenIO, not because you may cause any harm to your application, but because you are probably doing it the hard way if you think you have to use an exit. Call us for help if you need advice on how to do what you want to do.

A user exit subroutine may be invoked at any or all of four cycle points:

- Postkeystroke - immediately following each key pressed.
- Post Arrow key - immediately following each of the arrow keys.
- Predisplay - immediately prior to displaying each character of a field.
- Timer - at specified time intervals.

Applications for User Exits

The panel editor is an example of an application that must regain control following each arrow keystroke when you draw lines with the arrow keys or mouse.

An arrow key exit is called to provide the drawing and painting facilities. We do this by changing the color or character that was about to be displayed, and then returning to ScreenIO. ScreenIO treats the character returned by the exit as though you typed it, and displays it on the screen. In the case of arrow keys, ScreenIO also positions the caret one character in the direction of the arrow key.

We also use the predisplay and postkeystroke exits to handle painting and other situations that occur in the Panel Editor.

If you have an application that requires every byte of a field be a different color, or you want to highlight a few characters in a field, it is far simpler and much more efficient to code a user exit than it is to define a bunch of 1 byte fields. That's how we wrote the full screen editor, which consists of one field per line. The different colors of characters within the line are handled by a predisplay exit.

The timer exit point is used to pass control to time-dependent tasks. ScreenIO will call this exit point while waiting for the user to hit a key. Naturally, the exit will only be called while ScreenIO is in control. This is not multi threading!

The global postkeystroke exit point may be used to record keystrokes (and then play them back using the timer exit) for a tutorial or demo of your application. Other applications may be for systemwide editing on a character-by-character basis.

Coding User Exits

All of the user exits are called with the same parameter string, which consists of the following arguments:[5]

1. Parameters passed to the exit by ScreenIO, which consist of:

[5] We provide a copybook named EXITPARM.COB which supplies the definitions for these fields.

EXIT-TYPE - The type of exit - A (Arrow key), D (preDisplay), K (postKeystroke), P (global Postkeystroke) or T (Timer). Specifies to the exit program which exit point is being taken.

EXIT-FIELD-NUMBER - The current field number.

EXIT-FIELD-POSITION - Offset within the current field.

EXIT-FIELD-ROW - Line of the screen the character is in.

EXIT-FIELD-COLUMN - Column of screen the character is in.

EXIT-COLOR-NUMBER - The color attribute of the field this character is in, and on return to ScreenIO, the color attribute to use when displaying the character.[6]

EXIT-KEY-VALUE - The character just keyed (postkeystroke exits), about to be displayed (predisplay exits), or the character at the cursor position (in arrow key exits). This can be changed by the user to change what is about to appear on the screen for predisplay exits, or to change the character just typed in postkeystroke exits. You can even have your exit simulate a function key.

EXIT-ACTIONS-POSTKEYSTROKE - A group of fields to specify that ScreenIO perform end-of-field processing, ignore the keystroke, or reject the keystroke and issue an error message.

EXIT-ACTIONS-TIMER redefines the field above, EXIT-ACTIONS-POSTKEYSTROKE, and allows you to change the timer interval and issue messages.

EXIT-KEY-SCAN-CODE is the scan code equivalent of EXIT-KEY-VALUE.

2. The entire user panel data area, documented elsewhere.

3. Any other 01 level data area you place under the panel-PASS-TO-EXIT argument in your call to ScreenIO. ScreenIO just passes this argument to your exit when it calls the exit.

In simplest terms, the arrow key, predisplay, or postkeystroke exit is called with a character and a color value, and may return a different character and/or color value. You can easily see that this is how lines are drawn; the arrow key exit is passed the direction of movement and the cursor position, and it passes back the appropriate line character for ScreenIO to display, depending on the surrounding characters.

[6] Effective only for predisplay exits.

After you have compiled your user exit, you must LINK it to produce a .DLL module (of the appropriate form) which will be called dynamically by ScreenIO. 16-bit exits must be a CA-Realia 5.x .DLL. 32-bit exits must be a standard Windows .DLL; the arguments are passed BY REFERENCE. See your compiler's documentation.

You do not have to make any PROCEDURE DIVISION code changes within your driver program when you use a user exit. If you need to pass other data to your exit, you substitute that argument in the CALL to ScreenIO instead of the USER-DATA-FOR-EXITS argument, which ScreenIO will pass through to your exit unchanged. All of the information ScreenIO needs to process a user exit is placed in your copybook by the Panel Editor.

Postkeystroke Exit

The postkeystroke exit may be utilized to translate one character to another; a possible use would be to add non-English characters to unused positions on the keyboard. You can even substitute a function key for a letter.

This exit is called immediately following each keystroke in fields for which a postkeystroke exit is specified. It is invoked prior to any data editing or validation functions. The principal use is for character translation or to examine the character just keyed.

We use postkeystroke exits in the panel editor to prevent you from typing a character where you have defined a user field. If you try it, you will see we merely pass back the field mark character instead of the one you typed, effectively ignoring any character typed by you in an illegal area.

You can also use the postkeystroke exit to ignore characters, throwing keystrokes away as it were. This is useful when a keyboard attachment such as a bar-code scanner reads more characters than you want, for example.

You can instruct ScreenIO to perform end-of-field logic, which will tab out of the field as though it were full (invoking HOT field logic and validation if applicable).

You may, if you wish, edit the user's data on a keystroke by keystroke basis. Specification of an error message number results in the keystroke being ignored and that error message being issued (the rules are the same as for *panel*-DISPLAY-OPTION). You may, for example, reject all function keys not supported by your application.

You don't have to be particularly concerned with efficiency with postkeystroke exits (compared to the other exits), since it will take a very fast typist to get ahead of most things you would do in such an exit.

Arrow Key Exits

This exit is applied on the panel level, not the field level. It is called each time the user presses an arrow key.

The presence of a name in the field *panel*-ARROW-EXIT-NAME signifies the activation of the arrow key exit processing logic. To turn the arrow key exits off or on your program can remove and restore the name of the arrow key exit in this field.

The sequence of operation is this:

- User presses an arrow key.
- ScreenIO calls the arrow key exit, passing the value of the key in the field *panel*-EXIT-KEY, as well as the other exit parameters.
- The user exit returns a character and a color attribute which will be used to load the data to the user field at the cursor location and to display the data.
- The caret is moved one space in the direction of the arrow key.

The arrow key exit logic in ScreenIO assumes that you are working in the first fields on the panel, and that those fields are immediately above one another on the screen. It is not necessary to make the fields any special length, although they must all be the same length.

Arrow key exits are called every arrow keystroke, like the postkeystroke exit.

Predisplay Exits

This exit is called immediately before each character is displayed.

User applications that perform text editing may find this exit useful to highlight text in the middle of a field, for example. This exit could also be used to highlight erroneous characters in a field, or to highlight individual characters in a field by changing the color.

This exit is very costly, since it is called for every character in every field that is being displayed. In case you wondered, the reason that the full screen editor is slower than other panels is that a predisplay exit is called for every character on the image, except the menu line. It doesn't take much imagination to see that minor inefficiencies in a predisplay exit become very significant when multiplied by several thousand calls!

Timer Exits

Most of ScreenIO's time is spent waiting for the user to do something. This exit point allows you to get control during this time. If you return a value in EXIT-KEY-VALUE, ScreenIO will accept it as though the user hit that key, allowing you to drive automatic demos of your application or simulate keystrokes.

If the exit task is written efficiently, your users can be typing as fast as possible and they won't even know another process is operating. The minimum interval is 0.10 second.

Sample Predisplay Exit

The sample user exit below will evaluate the character it receives, and if it is a digit from 0 through 7, it will output the character in the foreground color of that number. All other characters will be returned with their value and color unchanged. You could conceivably use such a routine to display a bar chart on the screen, for example.

```
 LINKAGE SECTION.
     COPY EXITPARM.
*            DUMMY-PANEL is a copy of the panel copybook.
*            It is not used in this sample.
 01  DUMMY-PANEL                      PIC X.
*            USER-DATA-FOR-EXITS is any data you want to
*            pass from your main program to the exit.
 01  USER-DATA-FOR-EXITS              PIC X.

 PROCEDURE DIVISION USING BY REFERENCE       EXIT-PARMS
                                      DUMMY-PANEL
                                      USER-DATA-FOR-EXITS.
 0001-STARTUP.
     IF EXIT-FIELD-CHAR < '0' OR > '7'
       NEXT SENTENCE
     ELSE
       MOVE EXIT-FIELD-CHAR TO EXIT-COLOR-NUMBER.
     GOBACK.
```

Sample Timer Exit

This sample timer exit will make a noise at two second intervals while ScreenIO has control. You would probably want to do something more constructive in your exit program! The code that sets up the exit is shown first.

```
01  FUNCTION-FLAG             PIC S9(4) COMP-5 VALUE 13.

01  TIMER-EXIT-PARAMETERS.
  05  TIMER-EXIT-SWITCH       PIC X VALUE 'Y'.
  05  TIMER-EXIT-NAME         PIC X(8) VALUE 'METRONOM'.
  05  TIMER-EXIT-INTERVAL     PIC S9(3)V9 COMP-5 VALUE 2.

    CALL 'SCREENIO' USING     FUNCTION-FLAG
                              TIMER-EXIT-PARAMETERS
                              DUMMY
                              DUMMY.


DATA DIVISION.
WORKING-STORAGE SECTION.
01  ARG1                      PIC S9(4) COMP-5 VALUE 1.
LINKAGE SECTION.
    COPY EXITPARM.
01  DUMMY-PANEL               PIC X.
01  USER-DATA-FOR-EXITS       PIC X.

PROCEDURE DIVISION USING BY REFERENCE       EXIT-PARMS
                                     DUMMY-PANEL
                                     USER-DATA-FOR-EXITS.
0001-STARTUP.
    CALL 'SCR_BEEP' USING     ARG1
                              ARG1.
    GOBACK.
```

Subroutines & Functions

SUBROUTINES

SCR_BEEP (Make a noise)

This routine plays the Windows asterisk sound. The content of the argument is not significant. The routine expects two arguments, however; this example is correct.

```
    01 ARGUMENT                  PIC S9(4) COMP-5.


        CALL 'SCR_BEEP' USING    ARGUMENT
                                 ARGUMENT.
```

SCR_BIBY (Sets Bits in a Byte)

This routine allows you to set individual bits in a byte. Set the 05 level fields in BIT-DATA to LOW-VALUE to set the corresponding bit to 0, or anything else to set the bit to 1.

```
    01 BYTE-DATA.
      05 SOURCE-BYTE             PIC X.
      05 FILLER                  PIC X.

    01 BIT-DATA.
      05 BIT-1                   PIC X.
      05 BIT-2                   PIC X.
      05 BIT-3                   PIC X.
      05 BIT-4                   PIC X.
      05 BIT-5                   PIC X.
      05 BIT-6                   PIC X.
      05 BIT-7                   PIC X.
      05 BIT-8                   PIC X.


        CALL 'SCR_BIBY' USING    BYTE-DATA
                                 BIT-DATA.
```

SCR_BYBI (Expand Byte to Bits)

This routine is the complement to SCR_BIBY. It allows you to extract values of individual bits of a byte.

The 05 level fields in BIT-DATA will be LOW-VALUE if the corresponding bit was set to 0, or 01H if it was set to 1.

```
01 BYTE-DATA.
  05 SOURCE-BYTE             PIC X.
  05 FILLER                  PIC X.

01 BIT-DATA.
  05 BIT-1                   PIC X.
  05 BIT-2                   PIC X.
  05 BIT-3                   PIC X.
  05 BIT-4                   PIC X.
  05 BIT-5                   PIC X.
  05 BIT-6                   PIC X.
  05 BIT-7                   PIC X.
  05 BIT-8                   PIC X.


    CALL 'SCR_BYBI' USING    BYTE-DATA
                             BIT-DATA.
```

SCR_CASL (Force Characters to Lower Case)

This routine translates letters in the given string from upper to lower case. It correctly handles non-English alphabets.

```
01 MY-STRING                 PIC X(???).
01 MY-STRING-LENGTH          PIC S9(4) COMP-5 VALUE ???.
                             .
                             .
                             .
    CALL 'SCR_CASL' USING    MY-STRING
                             MY-STRING-LENGTH.
```

SCR_CASU (Force Characters to Upper Case)

This routine translates letters in the given string from lower to upper case. It correctly handles non-English alphabets.

```
01 MY-STRING                 PIC X(???).
01 MY-STRING-LENGTH          PIC S9(4) COMP-5 VALUE ???.
                             .
                             .
                             .
    CALL 'SCR_CASU' USING    MY-STRING
                             MY-STRING-LENGTH.
```

Note:

These two routines will be quite happy to stomp through memory converting everything they find to upper or lower case. Be sure the string length is correct to avoid memory violations or difficult-to-debug program errors.

SCR_ENVG (Get Environmental Value)

This will return the value of an environmental variable.

ENV-STR-NAME must contain the name of the environmental variable, followed by a LOW-VALUE.

```
    01  ENV-STR-NAME             PIC X(121).
    01  ENV-STR-VALUE            PIC X(121).
                                 .
                                 .
                                 .
        STRING env-variable-name
               LOW-VALUE DELIMITED SIZE INTO ENV-STR-NAME.

        CALL 'SCR_ENVG' USING    ENV-STR-NAME
                                 ENV-STR-VALUE.
```

SCR_SHFT (Get Shift Status)

This returns the asynchronous shift state of these keys. You may then analyze the shift states with SCR_BYBI as shown below:

The 05 level fields in SHIFT-LIST will be low-value if the shift state was off, and 01H if shift state was on.

```
    01 SHIFT-STATES.
      05 SHIFT-STATE-BYTE        PIC X.
      05 FILLER                  PIC X.

    01 SHIFT-LIST.
      05 SL-INSERT               PIC X.
      05 SL-CAP-LOC              PIC X.
      05 SL-NUM-LOC              PIC X.
      05 SL-SCROLL               PIC X.
      05 SL-ALT                  PIC X.
      05 SL-CTRL                 PIC X.
      05 SL-LEFT-SHIFT           PIC X.
      05 SL-RIGHT-SHIFT          PIC X.

        CALL 'SCR_SHFT' USING    SHIFT-STATES.

        CALL 'SCR_BYBI' USING    SHIFT-STATE-BYTE
                                 SHIFT-LIST.
```

FUNCTIONS

Calls to ScreenIO to set processing modes or to perform non-display functions.

0	Disables an alternate color table enabled using function 1.

1	Alternate Color Table: Causes ScreenIO to substitute colors within your application.
4	Window Save
5	Window Restore.
6	Key Redefinition.
8	Floppy Menus: Displays alternate menus when Alt, Ctrl, or Shift keys are pressed.
9	Miscellaneous Functions lets you specify various options.
10	More Miscellaneous Functions lets you specify things we left out of Function 9.
11	Required/Must-Validate Override Table: Establishes a table of key values that will be honored in spite of Must Validate logic. You can also specify this for HOT fields.
12	Timeout Interval: Causes ScreenIO to return to your program after a specified period of time with no keyboard activity. Useful for multi-user applications where you don't want records to be locked for too long.
13	Timer Exit: Causes ScreenIO to call, at specified intervals, a subroutine .DLL provided by you.
14	Global Postkeystroke Exit: Causes ScreenIO to call a subroutine provided by you following every keystroke. This may be used for recording keystrokes or system wide character-by-character editing.
15	Shutdown performs an orderly shutdown of your Windows application. Required before you STOP RUN.
17	Returns ScreenIO information, including window size and position, font information, Windows instance and window handles, etc.

Function 0: Disable Alternate Color Table

Disables an alternate color table you previously enabled using function 1.

```
01 FUNCTION-0              PIC S9(4) COMP-5 VALUE 0.


    CALL 'SCREENIO' USING   FUNCTION-0
                            DUMMY
                            DUMMY
                            DUMMY.
```

Function 1: Alternate Color Table

To define an alternate color table, you must specify a 272 element table. In the table, you specify the color substitution you desire. For example, if you want areas normally mapped normal-white-on black (color attribute 007) to be mapped high-intensity-white-on-blue (color attribute 31), place the value 031 in the field COLOR-007 (the eighth table element).

Every time ScreenIO encounters a character which has a color attribute of 007, it will automatically substitute the value you specified in the table at this position.

The alternate color table will remain in effect for the duration of the processing with no further action on your part. You may modify the color table without the need of another function call.

```
01 FUNCTION-1                PIC S9(4) COMP-5 VALUE 1.

01 ALTERNATE-COLOR-TABLE.
  05 FILLER                  PIC X(4).
  05 COLOR-000               PIC S9(4) COMP-5 VALUE 000.
  05 COLOR-001               PIC S9(4) COMP-5 VALUE 001.
    .                        .
    .                        .
  05 COLOR-254               PIC S9(4) COMP-5 VALUE 254.
  05 COLOR-255               PIC S9(4) COMP-5 VALUE 255.
  05 FILLER-000              PIC S9(4) COMP-5 VALUE 000.
    .                        .
    .                        .
  05 FILLER-015              PIC S9(4) COMP-5 VALUE 000.

    CALL 'SCREENIO' USING    FUNCTION-1
                             ALTERNATE-COLOR-TABLE
                             DUMMY
                             DUMMY.
```

Function 4: Window Save

This function allows you to save the contents of your application window. You are responsible for providing a large enough area to store the image. The minimum area is 4020 bytes for a 25 line image, 6900 bytes for a 43 line image, or 8020 bytes for a 50 line image.

You cannot print the image directly because it contains color attribute and other information, as well as the text of the panel. The internal layout of this area may be changed in the future.

```
01 FUNCTION-4                PIC S9(4) COMP-5 VALUE 4.

01 WINDOW-SAVE-AREA          PIC X(8020).

    CALL 'SCREENIO' USING    FUNCTION-4
                             WINDOW-SAVE-AREA
                             DUMMY
                             DUMMY.
```

Function 5: Window Restore

The mate to the Window Save operation, this function call allows you to restore an image previously saved. It is called as follows:

```
01 FUNCTION-5                PIC S9(4) COMP-5 VALUE 5.

01 WINDOW-SAVE-AREA          PIC X(8020).

    CALL 'SCREENIO' USING    FUNCTION-5
                             WINDOW-SAVE-AREA
                             DUMMY
                             DUMMY.
```

Function 6: Key Redefinition

This facility allows you to redefine any key to ScreenIO as any other key, including the redefinition of a character key as an extended function key. The redefinition is local to ScreenIO; it will not affect any other application.

The table may be of any size necessary to accomplish the redefinitions you desire. Be certain to show the correct size of the table in the field NUMBER-OF-KEY-REDEFS or you may experience some undesired redefinitions.

The following code will redefine the Enter key as a Tab, and the Escape key as an End. Each key definition consists of two numeric values. These correspond to the character sequences that are returned by the PC BIOS.[1] See the Appendix for a table of key values.

Character codes

Character codes include the values returned when you press a character key (letters, numbers, punctuation, etc.) and the ASCII control codes Backspace (08), Tab (09), Enter (13), and Escape (27). These are returned as a single numeric value.

In ScreenIO, you specify these as the first value in the key definition, and the second (unused) value is zero.

Extended function codes

Extended function codes are returned when you press function keys and Alt- key combinations. These cause two numeric values to be sent to the application (ScreenIO, in this case). The first is zero, and the second contains the extended function code.

The following is an example of redefining the character key Enter to be treated as a character key Tab, and to redefine the character key Escape as the extended function key End.

[1] We provide function key values in the KEYVALUE.COB copybook supplied with ScreenIO.

```
01 FUNCTION-6                PIC S9(4) COMP-5 VALUE 6.

01 KEY-REDEF-TABLE.
  05 FILLER                  PIC X(4).
  05 NUMBER-OF-KEY-REDEFS    PIC S9(4) COMP-5 VALUE 2.

  05 SOURCE-KEY-1.
    10 ENTER-KEY-R           PIC S9(4) COMP-5 VALUE 13.
    10 FILLER                PIC S9(4) COMP-5 VALUE 0.
  05 NEW-REDEF-1.
    10 TAB-KEY-R             PIC S9(4) COMP-5 VALUE 09.
    10 FILLER                PIC S9(4) COMP-5 VALUE 0.

  05 SOURCE-KEY-2.
    10 ESCAPE-KEY-R          PIC S9(4) COMP-5 VALUE 27.
    10 FILLER                PIC S9(4) COMP-5 VALUE 0.
  05 NEW-REDEF-2.
    10 FILLER                PIC S9(4) COMP-5 VALUE 0.
    10 END-KEY-R             PIC S9(4) COMP-5 VALUE 79.


    CALL 'SCREENIO' USING    FUNCTION-6
                             KEY-REDEF-TABLE
                             DUMMY
                             DUMMY.
```

The important points to remember about the redefinition table are:

- Characters have the decimal equivalent in the first field followed by zero in the second.

- Function keys have zero in the first field and the function key value in the second.

You should make the key redefinition function call from your main program. This will make the key redefinitions active for the entire application.

You can change the meaning of a redefined key without making another function call to ScreenIO by simply moving the new values into your table.

Function 8: Specify Floppy menus

This will cause ScreenIO to change the contents of the menu line based upon the status of the Alt, Ctrl, or Shift keys. It is useful if you have more options on a panel than will fit on a single menu. You may specify menus which are active on Alt- or Ctrl- key combinations, and which are automatically displayed when the key is depressed. Take a look at the Full Screen Editor, and press the Alt key to see this in action.

This function is enabled system wide, so you will have to clear the switches for your menus by setting them to "N" when you leave the panel for which this feature is enabled.

It is not necessary to make another function call to change the contents of this area or to switch it on or off. Just move the new data to the area or set the switches, and it will be immediately honored by ScreenIO.

```
    01 FUNCTION-8                PIC S9(4) COMP-5 VALUE 8.

    01 MENU-DATA.

      05 FILLER                  PIC X(4).

      05 MENU-ALT-SWITCH         PIC X VALUE 'Y'.
      05 MENU-CTRL-SWITCH        PIC X VALUE 'Y'.
      05 MENU-LEFT-SWITCH        PIC X VALUE 'Y'.
      05 MENU-RIGHT-SWITCH       PIC X VALUE 'Y'.

      05 MENU-ALT                PIC X(80) VALUE
           'The ALT key is down!  '.

      05 MENU-CTRL               PIC X(80) VALUE
           'The CONTROL key is down!  '.

      05 MENU-LEFT               PIC X(80) VALUE
           'The LEFT SHIFT key is down!  '.

      05 MENU-RIGHT              PIC X(80) VALUE
           'The RIGHT SHIFT key is down!  '.


        CALL 'SCREENIO' USING    FUNCTION-8
                                 MENU-DATA
                                 DUMMY
                                 DUMMY.
```

Function 9: Miscellaneous Functions

Suppress HOT exit on field full

When you specify a field is HOT or HOT CHANGED, ScreenIO will return control to your program when the user tabs from the field, or if autoskip is suppressed, when the field becomes full.

There are some occasions where it is desirable to restrict HOT field activity to explicit Tab operations; that is, when the user presses the Tab key. This combination of options will effectively restrict HOT field activity to explicit tabbing, when the user presses the Tab key.

Suppress Alternate Key Table in Selector fields

If you redefine the Enter key as a Tab key you may make it difficult to use Selector fields. This is because pressing Enter will move the cursor out of the field rather than returning control to your program. This feature allows you to override the alternate key definition in field types S and R.

```
    01 FUNCTION-9                PIC S9(4) COMP-5 VALUE 9.

    01 MISC-SWITCHES.
      05 FILLER                  PIC X.
      05 FILLER                  PIC X.
      05 SUPPRESS-HOT-FULL       PIC X VALUE 'Y'.
      05 SUPPRESS-KEYDEF-IN-SELECTOR     PIC X VALUE 'Y'.


        CALL 'SCREENIO' USING    FUNCTION-9
                                 MISC-SWITCHES
```

```
                              DUMMY
                              DUMMY.
```

A subsequent call is required to modify the settings of any of these switches.

Function 10: More Miscellaneous Functions

Force Num-Lock in Numeric Fields

This function call will automatically force Num-Lock on each time the cursor enters a numeric field. To use the arrow keys with this feature, the user will have to press the shift key. This can be useful for rapid numeric data entry using the numeric keypad.

Specify Field Mark Character

This option will fill all completely blank fields with the character you specify in the variable FIELD-MARK-CHAR-xxx (an underbar or asterisk in this example). The character is stripped from the data returned to your program. Fields that are not entirely spaces are not affected. You may specify different characters for protected and unprotected fields.

```
    01 FUNCTION-10                 PIC S9(4) COMP-5 VALUE 10.

    01 MORE-MISC-SWITCHES.
      05 FORCE-NUM-LOCK            PIC X VALUE 'Y'.
      05 FIELD-MARK-CHAR-UNP       PIC X VALUE '_'.
      05 FIELD-MARK-CHAR-PRO       PIC X VALUE '*'.
      05 FILLER                    PIC X.
      05 FILLER                    PIC X.
      05 SUPPRESS-HOT-FULL         PIC X VALUE 'Y'.
      05 SUPPRESS-KEYDEF-IN-SELECTOR      PIC X VALUE 'Y'.
      05 FILLER                    PIC X(8).
      05 FILLER                    PIC S9(4) COMP-5.
      05 FILLER                    PIC X(33).


        CALL 'SCREENIO' USING      FUNCTION-10
                                   MORE-MISC-SWITCHES
                                   DUMMY
                                   DUMMY.
```

A subsequent call is required to modify the settings of any of these switches. The FILLER in the data area is required. Once invoked, this function is in effect system wide.

Function 11: Required/Must-Validate Bypass Table

Normally, the only way to exit a panel that has Required or Must Validate fields is to enter data that passes the edits in each such field.

Function 11 will allow specification of a table of function keys that will bypass the ScreenIO logic for Required and Must Validate fields. You can also specify bypassing this logic for HOT fields, which is usually the preferred way of doing business.

This example specifies that the End key, Enter key, and HOT fields will bypass validation logic. See key redefinition for an explanation of how to define key values.

```
01 FUNCTION-11              PIC S9(4) COMP-5 VALUE 11.

01 VALIDATE-BYPASS-TABLE.
  05 FILLER                 PIC X(4).
  05 NUMBER-OF-BYPASS-KEYS  PIC S9(4) COMP-5 VALUE 3.

  05 BYPASS-KEY-1.
    10 HOT-FIELD            PIC S9(4) COMP-5 VALUE 1000.
    10 FILLER               PIC S9(4) COMP-5 VALUE 0.

  05 BYPASS-KEY-2.
    10 ENTER-KEY-B          PIC S9(4) COMP-5 VALUE 13.
    10 FILLER               PIC S9(4) COMP-5 VALUE 0.

  05 BYPASS-KEY-3.
    10 FILLER               PIC S9(4) COMP-5 VALUE 0.
    10 END-KEY-B            PIC S9(4) COMP-5 VALUE 79.


    CALL 'SCREENIO' USING   FUNCTION-11
                            VALIDATE-BYPASS-TABLE
                            DUMMY
                            DUMMY.
```

Function 12: Timeout Interval

In multi-user environments such as local area networks you don't want to allow stations to monopolize resources. Users shouldn't lock a record and then go to lunch, for instance.

This function tells ScreenIO to return after a specified period of time of user inactivity.

There are two varieties of timeout available:

- Panel Timeout - This form of timeout will return control to your program a specified period of time after the panel is displayed, regardless of what the user is doing.

- Keystroke Timeout - This form will return control to your program a specified period of time after the last keystroke by the user.

Once invoked this function is in effect system wide, or until you shut if off.

```
01 FUNCTION-12              PIC S9(4) COMP-5 VALUE 12.


01  TIMEOUT-PARAMETERS.
  05  T-O-SWITCH            PIC X VALUE 'Y'.
  05  T-O-MSG-SWITCH        PIC X VALUE 'Y'.
  05  T-O-WARNING-LEADTIME  PIC S9(4) COMP-5 VALUE 10.
  05  T-O-SECONDS-PANEL     PIC S9(4) COMP-5 VALUE 60.
  05  T-O-PANEL-EXIT-KEY.
    10  T-O-PANEL-CHAR      PIC S9(4) COMP-5 VALUE 0.
    10  T-O-PANEL-FUNCTION  PIC S9(4) COMP-5 VALUE 79.
  05  T-O-SECONDS-KEYSTROKE PIC S9(4) COMP-5 VALUE 15.
  05  T-O-KEYSTROKE-EXIT-KEY.
```

```
10  T-O-KEYSTROKE-CHAR   PIC S9(4) COMP-5 VALUE 13.
10  T-O-KEYSTROKE-FUNCTION      PIC S9(4) COMP-5 VALUE 0.

CALL 'SCREENIO' USING    FUNCTION-12
                         TIMEOUT-PARAMETERS
                         DUMMY
                         DUMMY.
```

The items in the TIMEOUT-PARAMETERS are as follows:

- T-O-SWITCH - Yes/No switch to tell ScreenIO if timeouts are to be generated or not. Values are "Y" or "N".
- T-O-MSG-SWITCH - Yes/No switch to tell ScreenIO if timeouts are to be preceded by a warning to the user via a message in the menu line. The warning will occur at 2 second intervals, and will count down the remaining seconds till timeout. This gives users a chance to finish their work. Values are "Y" or "N".
- T-O-WARNING-LEADTIME - The number of seconds of lead time that the user should be warned before timeout will actually occur. Used with the message switch above. Legal values are from zero to the timeout duration below.
- T-O-SECONDS-PANEL - The timeout duration, or period of time before the timeout will occur. Timing commences when the panel is displayed. The timer is reset each time control returns to your program.
- T-O-PANEL-EXIT-KEY - The key that ScreenIO will simulate if the Panel Timeout duration is exhausted.[2]
- T-O-SECONDS-KEYSTROKE - The number of seconds of keyboard inactivity before a Keystroke Timeout will occur.
- T-O-KEYSTROKE-EXIT-KEY - The keystroke that will be simulated when a keystroke timeout occurs.

As coded above, our example will simulate an Enter key after 15 seconds without any keyboard activity, and will simulate an End key 60 seconds after the panel is displayed. A warning will be issued in the menu line at two second intervals 10 seconds before timing out.

[2] These simulated function keys are coded the same as they are for function 6, with Tab, Enter, Escape and Backspace being entered in the first value and all other function keys in the second value. Note that this allows you to simulate the pressing of any key or key-combination. Note also that by setting the first value to some weird number (greater than 9000 for example) you may signal your program in a way the user could never accomplish via the keyboard.

Function 13: Specify Timer Exit

This instructs ScreenIO to call a user exit every so often. You can't update the screen.

Unlike Function 12 timeouts, this function does not cause a return to your program, but rather calls another subprogram. The rules for coding exit programs are discussed elsewhere.

```
01  FUNCTION-13             PIC S9(4) COMP-5 VALUE 13.

01  TIMER-EXIT-PARAMETERS.
  05  TIMER-EXIT-SWITCH      PIC X VALUE 'Y'.
  05  TIMER-EXIT-NAME        PIC X(8) VALUE 'yourexit'.
  05  TIMER-EXIT-INTERVAL    PIC S9(3)V9 COMP-5 VALUE 2.


    CALL 'SCREENIO' USING    FUNCTION-13
                             TIMER-EXIT-PARAMETERS
                             DUMMY
                             DUMMY.
```

The items in the TIMER-EXIT-PARAMETERS are as follows:

- TIMER-EXIT-SWITCH - Simple Yes/No switch to tell ScreenIO if timer exit is to be called or not. Values are "Y" or "N".

- TIMER-EXIT-NAME - Name of user exit program to be dynamically called by ScreenIO. See the section on User Exit Subroutines in the Advanced Topics Chapter for tips on how to code timer exits.

- TIMER-EXIT-INTERVAL - The seconds and tenths of seconds between each timer exit call.

A timer exit is in effect system wide until you turn it off or end your application. You must make a new function call to change the time interval or the exit program name. Only one exit program will be called each interval; you can't have two timers running at once.

Function 14: Global Postkeystroke Exit

This function is designed to instruct ScreenIO to CALL a subroutine (user exit) following every keystroke the user hits, system wide. It is invoked before, and in addition to the field-specific postkeystroke exit that may be defined in the panel editor.

This exit may perform any functions allowed within the field-specific equivalent. A more interesting application, however, would be for a keystroke recording facility, which could be used in conjunction with the Timer Exit (Function 13) to playback the recorded keys, perhaps in a tutorial or a demo of your application

```
01  FUNCTION-14             PIC S9(4) COMP-5 VALUE 14.

01  POSTKEY-EXIT-PARAMETERS.
  05  POSTKEY-EXIT-SWITCH    PIC X VALUE 'Y'.
  05  POSTKEY-EXIT-NAME      PIC X(8) VALUE 'yourexit'.
```

```
           CALL 'SCREENIO' USING    FUNCTION-14
                                    POSTKEY-EXIT-PARAMETERS
                                    DUMMY
                                    DUMMY.
```

The items in the POSTKEY-EXIT-PARAMETERS are as follows:

- POSTKEY-EXIT-SWITCH - Simple Yes/No switch to tell ScreenIO if timer exit is to be called or not. Values are "Y" or "N".
- POSTKEY-EXIT-NAME - Name of user exit program to be dynamically called by ScreenIO. See the section on User Exit Subroutines in the Advanced Topics Chapter for tips on how to code timer exits.

Once invoked, this exit is in effect system wide until you turn it off via the switch.

Function 15: Shutdown

This function must be issued prior to terminating your Windows application. For the technically inclined, this function destroys your application's window, which closes all of your application windows, flushes the application message queue, destroys outstanding timers, and a few other important things. It does NOT terminate your program, however.

```
       01  FUNCTION-15              PIC S9(4) COMP-5 VALUE 15.

           CALL 'SCREENIO' USING    FUNCTION-15
                                    DUMMY
                                    DUMMY
                                    DUMMY.
```

Function 17: Return ScreenIO Information

This function call fills the SCRWINIT.COB copybook with the current settings, including the Windows instance handle and window handle of your application. You can save this in a file and, next time the user runs your application, you can make your initialization call using this saved information.

That way, your application will come up in the same place and size on the screen as it was when the user quit last time.

See the section in Programming/Initialization for details on the data in this copybook.

```
       01  FUNCTION-17              PIC S9(4) COMP-5 VALUE 17.

           CALL 'SCREENIO' USING    FUNCTION-17
                                    WINDOWS-INITIALIZATION
                                    DUMMY
                                    DUMMY.
```

Function Call Technical Notes:

How they work:

You may be wondering how we are able to detect changes you make to the Floppy Menu items, changed key definitions, and changed color table values without requiring you to call ScreenIO again.

When you make one of these function calls to ScreenIO, we save the address of the storage you passed us when you made the function call to ScreenIO. We can then reference your storage at any time to detect changes.

The benefits of this technique are that it requires no storage allocation in ScreenIO, and it allows you to modify the action of these functions (e.g. change alternate menu contents) directly without the need to perform additional function calls.

However, if you CANCEL the program that contained the storage, ScreenIO will no longer be able to refer to the storage and it will quit working. Generally, that's what you expected in the first place, so it is fail-soft. If you want the function to remain available you should perform the function call from your main program; that storage will remain accessible to ScreenIO for the duration of your application.

Notes regarding window save and restore:

These facilities are provided for the case (1) where the underlying base panels are not created by ScreenIO, and can not easily be reconstructed, or (2) where the desired effect is to remove one of many windows overlaying a base panel without having to call ScreenIO for the base panel and then each remaining window in turn.

If you use these facilities you are responsible for saving and restoring images at the proper time, as ScreenIO has no way of knowing when to restore what!

There is also a risk of restoring a panel image that does not match what your software 'thought' was there, or restoring a panel image and then recalling ScreenIO with the most recent panel name. In these cases you will likely see fields popping up from nowhere on a background that does not otherwise correspond to the panel ScreenIO is currently displaying.

Support

We provide excellent technical support. Just ask any ScreenIO user.

If you have a problem, you may contact us via e-mail, snail mail, or by calling us. You can also get tips from our Internet site. Our address and telephone numbers are in the front of this manual.

If you call us, remember that we operate on Alaska time, 8:00 to 5:00. Alaska time is 4 hours earlier than Eastern Time, and 1 hour West of (earlier than) Pacific Time. Please read this section before calling, and have the appropriate information available when you call.

If you are reporting a problem, it really helps if you have tested it by reducing it down to the smallest possible program source code and files necessary to demonstrate the problem. Funny how the process of reducing a problem down to its essence tends to help you figure out what's wrong!

It is also very helpful if you have used your compiler's debugger to verify that the problem you are having is in fact a ScreenIO problem and not an error in your program.

Please Use Your COBOL Debugger!

And, please use it *before* calling us!

All of the facilities of your compiler's debugger are available when you use ScreenIO. You will probably be able to resolve most problems yourself if you utilize your compiler's interactive debugging facilities.

Test Your Panel Individually!

We have taken many technical support calls where a user thought that ScreenIO(!) was at fault but the problem was in fact their own error, something known around here as an *SEP*.[1]

The easiest way to verify whether a panel has a problem is to build a little test program that only displays the panel that you're having difficulties with. Just include your panel in this minimal ScreenIO program:

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. TESTPAN.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. IBM-PC.
        OBJECT-COMPUTER. IBM-PC.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01  FUNCTION-SHUTDOWN       PIC S9(4) COMP-5 VALUE 15.
            COPY KEYVALUE.
            COPY SCRWINIT.

       *    Your panel goes here:

            COPY panel.
       *
        PROCEDURE DIVISION.

            MOVE ALT-F4 TO SCREENIO-CLOSE-VALUE.

            CALL 'SCREENIO' USING    WINDOWS-INITIALIZATION
                                     WINDOWS-INITIALIZATION
                                     WINDOWS-INITIALIZATION
                                     WINDOWS-INITIALIZATION.
            PERFORM WITH TEST AFTER
                  UNTIL panel-EXIT-KEY = ALT-F4
              CALL 'SCREENIO' USING  panel-PANEL
                                     panel-PASS-TO-EXIT
                                     panel-WORK-S
                                     panel-WORK-D
            END-PERFORM.
            CALL 'SCREENIO' USING    FUNCTION-SHUTDOWN
                                     FUNCTION-SHUTDOWN
                                     FUNCTION-SHUTDOWN
                                     FUNCTION-SHUTDOWN.
        STOP RUN.
```

If it works properly in this test case, it isn't ScreenIO that has a problem! Many of these situations are caused by memory trashing within your program; that is, your program is writing (in error) on storage that it shouldn't. These problems are often caused by subscripts or indexes out of range, or by passing the wrong arguments to subroutines. Here's a good case where your debugger can help you.

[1] Somebody Else's Problem!

ScreenIO Event Logging

ScreenIO contains an event logging facility that can help you debug your application. ScreenIO's internal event logging is enabled by setting an environmental variable before you fire up your ScreenIO application.

ScreenIO will write an event log named SCREENIO.LOG in your Windows directory when logging is enabled. The log file may be printed or examined using a text editor.

ScreenIO events

ScreenIO's event log contains, among other things, its version number and compile date, the masks it got from the EDITMASK program (which you can use to verify if you were successful in modifying your masks), and a considerable amount of detail about ScreenIO's internal processing that could be helpful.

To turn on event logging:

For Windows 3.x:

Close Windows and give this command at the DOS level:

SET SCRWIN32=LOG

Now, fire up Windows 3.x and run your application.

For Windows 95 or Windows NT:

Open a COMMAND window and give this command at the DOS level:

SET SCRWIN32=LOG

Now run your application by invoking it within the DOS window (not from a Shortcut!)

For Windows NT only:

Settings/Control Panel/System/Environment; set the variable **SCRWIN32** to **LOG**.

Now run your application.

When you're finished, you can see much of what went on by examining the log.

Logging events in your application

You can also use ScreenIO's event logger to log events in your own application by calling the subroutine SCR_LOG and passing it the arguments contained in the copybook LOGDATA.COB. The nice thing is, you don't have to worry about initializing anything or performing any I/O. Just call SCR_LOG with whatever you want to log, and ScreenIO will write a record out to the event log.

The data structure in the copybook LOGDATA.COB is designed to be as flexible as possible. Everything in the area LD-TEXT is freeform. If you like, you may move any data to LD-TEXT and it will be written to the log. A careful inspection of the way we did this will reveal that we set up four fields, which can be loaded with textual data by using the name LD*n*-TEXT or with a numeric value by using the redefinition data name LD*n*-NUM.

```
*                          :----------------------------------------
* ------------------------: Event logging parameters.
*                          :----------------------------------------
*
 01  LOG-DATA                 VALUE SPACE.

*                          :----------------------------------------
* ------------------------: Info about calling program; you can
*                          : load this in a first-time paragraph.
*                          :
*                          : This section is only printed when
*                          : it changes, to reduce clutter.
*                          :
*                          : Use is optional.
*                          :----------------------------------------

     05  LD-PROGRAM-INFO.
       10  LD-PROGRAM         PIC X(8).
       10  FILLER             PIC X.
       10  LD-VERSION.
         15  LD-VERSION-MAJOR PIC X(4).
         15  LD-VERSION-MINOR PIC X(3).
       10  FILLER             PIC X.
       10  LD-COMPILE-DATE    PIC X(12).
       10  FILLER             PIC X.
       10  LD-COMPILE-TIME    PIC X(5).

*                          :----------------------------------------
* ------------------------: Separator; usually a vertical bar.
*                          :----------------------------------------

     05  LD-SEPARATOR         PIC X.

*                          :----------------------------------------
* ------------------------: Event data.  The layout is not
*                          : critical; the subdefinitions are
*                          : for convenience.  You can simply
*                          : move a text message to LD-TEXT if
*                          : you prefer, up to 160 bytes.
*                          :----------------------------------------

     05  LD-TEXT              PIC X(185).
     05  FILLER REDEFINES LD-TEXT.

* ------------------------: Name of event

       10  LD-TITLE           PIC X(25).

* ------------------------: Specific event info  (freeform OK)
```

```
       10  LD-INFO            PIC X(160).
       10  FILLER REDEFINES LD-INFO.

 * -----------------------: 22 character text or numeric value

         15  LD1-TEXT         PIC X(22).
         15  FILLER REDEFINES LD1-TEXT.
           20  LD1-NUM        PIC S9(9) SIGN LEADING SEPARATE.
           20  FILLER         PIC X(12).

 * -----------------------: 15 character text or numeric value

         15  LD2-TEXT         PIC X(15).
         15  FILLER REDEFINES LD2-TEXT.
           20  LD2-NUM        PIC S9(9) SIGN LEADING SEPARATE.
           20  FILLER         PIC X(5).

 * -----------------------: 15 character text or numeric value

         15  LD3-TEXT         PIC X(15).
         15  FILLER REDEFINES LD3-TEXT.
           20  LD3-NUM        PIC S9(9) SIGN LEADING SEPARATE.
           20  FILLER         PIC X(5).

 * -----------------------: 15 character text or numeric value

         15  LD4-TEXT         PIC X(15).
         15  FILLER REDEFINES LD4-TEXT.
           20  LD4-NUM        PIC S9(9) SIGN LEADING SEPARATE.
           20  FILLER         PIC X(5).

         15  FILLER           PIC X(78).
```

We included the sample program TESTLOG.COB with the package to show you how to set up event logging. Here's a listing of the program. As you can see, it's easy to use.

```
  IDENTIFICATION DIVISION.
 *                        :---------------------------------------
 * -----------------------: Program to demonstrate how to use
 *                        : event logging facilities of ScreenIO
 *                        : from within your application.
 *                        :---------------------------------------
  PROGRAM-ID. TESTLOG.
  DATE-COMPILED.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. IBM-PC.
  OBJECT-COMPUTER. IBM-PC.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  V--VERSION-LEVEL        PIC 9(3) VALUE 2.

 *                        :---------------------------------------
 * -----------------------: Note:  Our editor automatically
 *                        : increments V--VERSION-LEVEL when we
 *                        : change a program and save it.
 *                        :---------------------------------------
 * -----------------------: ScreenIO copybooks for Windows
 *                        : initialization call and for
 *                        : obtaining runtime information.
 *                        :---------------------------------------
 *
         COPY SCRWINIT.
         COPY INITPAN.
         COPY KEYVALUE.

  01  FUNCTION-SHUTDOWN       PIC S9(4) COMP-5 VALUE 15.
 01  FUNCTION-INFO            PIC S9(4) COMP-5 VALUE 17.

*                          :--------------------------------------
* ------------------------: Get/Process environmentals...
                           :--------------------------------------
*
 01  ENV-STR-NAME             PIC X(121).
 01  ENV-STR-VALUE            PIC X(121) VALUE LOW-VALUES.
 01  FILLER REDEFINES ENV-STR-VALUE.
     05  ES-FLAGS             PIC X(10).
     05  FILLER               PIC X(111).
 01  FILLER REDEFINES ENV-STR-VALUE.
     05  ES-3                 PIC X(3).
     05  FILLER               PIC X(118).
 01  ENV-LENGTH               PIC S9(4) COMP-5.
*
*                          :--------------------------------------
* ------------------------: Stuff for Event Logging.
*                          :--------------------------------------
*
 01  COMPILE-DATA.
     05  COMPILE-TIME         PIC X(5).
     05  COMPILE-DATE         PIC X(12).
         COPY LOGDATA.
*
 01  FILLER                   PIC X VALUE 'Y'.
     88  FIRST-TIME           VALUE 'Y'.
     88  NOT-FIRST-TIME       VALUE 'N'.
*
 01  FILLER                   PIC X.
     88  DEBUG-LOG            VALUE 'L'.
*
 PROCEDURE DIVISION.
 0001-INITIALIZE.
*                          :--------------------------------------
* ------------------------: Check environmental PROGRAM-ID to see
*                          : if event logging is wanted; you can
*                          : use any technique you wish, but this
*                          : is what we do:
*                          :
*                          :     SET TESTLOG=LOG
*                          :
*                          : Will enable logging for TESTLOG.
*                          :--------------------------------------
*
     IF FIRST-TIME
       SET NOT-FIRST-TIME    TO TRUE
       MOVE 'TESTLOG'        TO LD-PROGRAM
       MOVE '1.1.'           TO LD-VERSION-MAJOR
       MOVE V--VERSION-LEVEL TO LD-VERSION-MINOR
       MOVE WHEN-COMPILED    TO COMPILE-DATA
       MOVE COMPILE-DATE     TO LD-COMPILE-DATE
       MOVE COMPILE-TIME     TO LD-COMPILE-TIME
       MOVE '|'              TO LD-SEPARATOR
       STRING LD-PROGRAM     DELIMITED SPACE
               LOW-VALUE     DELIMITED SIZE INTO ENV-STR-NAME
 * ------------------------: Get the environmental value using
*                          : subroutine supplied with ScreenIO:
       CALL 'SCR_ENVG' USING ENV-STR-NAME
                             ENV-STR-VALUE
*
* ------------------------: Force value to uppercase.
*
       IF ENV-STR-VALUE > SPACE
         MOVE 3 TO ENV-LENGTH
         CALL 'SCR_CASU' USING ENV-STR-VALUE ENV-LENGTH
         IF ES-3 = 'LOG'
           SET DEBUG-LOG TO TRUE.
```

```
*
* ------------------------: Log entry to this program
*
     MOVE 'Program started' TO LD-TITLE.
     PERFORM 0002-LOG-EVENT.
*
*                         :--------------------------------------
* ------------------------: Initialize for Windows environment.
*                         :--------------------------------------
*
     MOVE ALT-F4 TO SCREENIO-CLOSE-VALUE.
     CALL 'SCREENIO' USING    WINDOWS-INITIALIZATION
                              WINDOWS-INITIALIZATION
                              WINDOWS-INITIALIZATION
                              WINDOWS-INITIALIZATION.
*
* ------------------------: Log another event
*
     MOVE 'Initialized ScreenIO' TO LD-TITLE.
     PERFORM 0002-LOG-EVENT.
*
     PERFORM WITH TEST AFTER
           UNTIL panel-EXIT-KEY = ALT-F4

       CALL 'SCREENIO' USING  INITPAN-PANEL
                              INITPAN-PASS-TO-EXIT
                              INITPAN-WORK-S
                              INITPAN-WORK-D
     END-PERFORM.

* ------------------------: Get info from ScreenIO to show
*                         : how data is written to event log.
*
     CALL 'SCREENIO' USING    FUNCTION-INFO
                              WINDOWS-INITIALIZATION
                              WINDOWS-INITIALIZATION
                              WINDOWS-INITIALIZATION.
*
* ------------------------: Log another event and display values
*
     MOVE 'ScreenIO Info' TO LD-TITLE.
     MOVE 'hWnd         ' TO LD1-TEXT.
     MOVE SCREENIO-HWND-32 TO LD2-NUM.
     MOVE 'hInstance    ' TO LD3-TEXT.
     MOVE SCREENIO-HINST-32 TO LD4-NUM.
     PERFORM 0002-LOG-EVENT.
*
* ---------------------- Clean up and quit.
*
     CALL 'SCREENIO' USING    FUNCTION-SHUTDOWN
                              FUNCTION-SHUTDOWN
                              FUNCTION-SHUTDOWN
                              FUNCTION-SHUTDOWN.
*
     STOP RUN.
* -====-
 0002-LOG-EVENT.
*                         :--------------------------------------
* ------------------------: Check if event logging is enabled;
*                         : if so, call the event logger SCR_LOG.
*                         :--------------------------------------
     IF DEBUG-LOG
       CALL 'SCR_LOG' USING LOG-DATA.
```

Nonfatal Problems

Problems that are nonfatal to your application cause unexpected results when you call ScreenIO. The common ones are discussed below.

Masking/Edit masks

New or changed mask is not displayed in the panel editor

You modified the text file EDITMASK.SEQ to add or change a mask, but you don't see it in the panel editor. Make sure that the EDITMASK.SEQ file that you changed is in the subdirectory you specified on the Options panel. If you don't see your changes, you probably have more than one EDITMASK.SEQ file, and you pointed the panel editor at the wrong one.

New or changed mask is not working when you run your application

Set the environmental SCRWIN32=LOG and then run your application. This will cause ScreenIO to generate an event log named SCREENIO.LOG in your Windows subdirectory. Examine the contents of SCREENIO.LOG with a text editor. One of the first things listed is the table of edit masks. If it doesn't agree with your EDITMASK.SEQ file, you just found your problem.

A review about mask modification is in order.

First, you make your mask changes available to the panel editor by changing the text file EDITMASK.SEQ. The panel editor will use the contents of EDITMASK.SEQ to figure out how to display your data, and to produce the panel copybook. The panel copybook will contain the mask number, not the text of the mask.

Next, you must arrange for ScreenIO to get the text of your mask at runtime (remember, your panel copybook only has the number of the mask, not the text). ScreenIO does it by calling the subroutine EDITMASK passing it the number of the mask. EDITMASK returns the mask text to ScreenIO. ScreenIO then displays your field with the text of the mask it was passed by EDITMASK.

So: If you modify EDITMASK.SEQ and don't see the results of the change when you run your application, review the discussion on modifying edit masks.

Problems with painting

Wrong image is displayed

You have called ScreenIO with arguments for two different panels. The most common cause is copying a CALL statement and only changing the WORK-S or WORK-D panel name. This usually results in the static portion of one panel being displayed with the fields of another.

You may have two copies of ScreenIO in memory at once. See the discussion in Programming regarding static and dynamic subroutine calls.

If you are using function calls to save and restore the image, you have restored the wrong image.

You may have used **I** in *panel*-REPAINT-SCREEN, which suppresses the painting of the static portion of a new screen. Any of these three errors typically results in fields from one panel appearing over the image of another panel.

Performance

Tabbing is slow on a panel with HOT fields

You are probably painting all user fields instead of only those that changed and require updating. Several things will help:

- Look into the HOT Modified option to avoid returning to your program unnecessarily.
- Paint only the fields you change when you call ScreenIO after processing a HOT field. Move N to *panel*-REPAINT-SCREEN and move R to *field-name*-P of the fields you changed.
- Streamline your HOT field logic.

Changed data not displayed

Colors changed by your program are not displayed

You suppressed painting the user fields by moving N to *panel*-REPAINT-SCREEN and forgot to move R or E to the corresponding attribute *field-name*-P.

Data changed by your program is not displayed

Same as above.

Validation

Validated fields return invalid data

REMEMBER, fields are only validated by ScreenIO if the user leaves the field (or attempts to). Your program must check for the V validation indicator if it is going to rely on ScreenIO to perform all editing.

You can also specify Must Validate on the fields, which will eliminate this requirement. Be sure to familiarize yourself with the full ramifications of Must Validate before your do this.

Data fields contain wrong data

You (most unwisely!) tried to rearrange the order of your fields in the panel copybook by editing it directly. Now you know why we told you not to do this!

This can also be caused by not correctly modifying your edit masks. If you generate the panel with edit masks that do not correspond to the masks that ScreenIO uses at runtime, data may be shifted into adjacent fields. Review the section on changing edit masks.

Caret/Active field positioning

Caret is not in the HOME field when a panel is redisplayed

If you do not reset *panel*-CURSOR-FIELD to the HOME field, the active field will be the same as it was when you last left the panel. This is not always a bad thing, since your panels will 'remember' where the user was in the panel.

ScreenIO is skipping newly unprotected fields following a HOT field

ScreenIO processes tabbing logic before returning to your program from a HOT field.

If you tab from a HOT field which is followed by a protected field and then you unprotect this field in your HOT field logic, you will find that ScreenIO skipped the newly unprotected field. You will have to explicitly set the active field if you want it to be the newly unprotected field.

Fatal Problems

Most of the problems that are fatal and commonly encountered are accompanied by a message number and a message of the form:

ScreenIO version aborted in panel *panelname* Reason = number

The Reason number may be used to refer to the appropriate message and explanation in ScreenIO Messages. It is also possible to encounter a few other fatal errors in ScreenIO, described below:

Memory violations or General Protection Faults

You have linked your application incorrectly. See your compiler's documentation on linking.

Your application's .EXE file may be corrupt. Recompile and LINK it and try again.

SCREENIO.LIB or SCRWIN32.DLL may be corrupted. If the panel editor works but your programs do not, call us to obtain a fresh copy.

The most common problem is memory trashing caused by an index or subscript out of range; enable your compiler's subscript checking option, recompile and LINK your programs. Here's a great place to use your interactive debugger.

If you don't follow the instructions carefully, you can cause memory violations by using subroutines or function calls we provide. Common problems are:

- Calling SCR_CASU or SCR_CASL with a string length that exceeds your intended data area.
- Calling a ScreenIO function with the wrong argument type; e.g. S9(4) COMP-4 instead of COMP-5.
- Requesting a Screen Save via a function call but providing less than the required amount of space for the save operation. This will result in memory following the too-short screen-save buffer being overwritten by the screen-save data.

ScreenIO Error Messages

The following messages are issued when ScreenIO terminates abnormally. They are listed in ascending order by error number. All possible messages are listed, although it is highly unlikely you will encounter most of them.

01: color-value **is an invalid BG color.**

The field *panel*-BASE-COLOR contains a value other than 0 through 8. This is probably one of the most commonly encountered error messages, and may be caused by a variety of errors. Check these things:

- Arguments are out of order in ScreenIO call. Most common.
- The panel copybook has been corrupted by your calling program prior to calling ScreenIO. This is usually caused by your subscripts exceeding the bounds of a table just before the panel copybook. Check with the debugger. If necessary, recompile your program with the subscript checking enabled so you can trap out of range subscripts.
- If on a function call, FUNCTION-FLAG is not PIC S9(4) COMP-5.
- Invalid arguments in ScreenIO call, e.g., mixing arguments from two panels or using wrong data names in CALL statement.

02: program-name **subroutine not found.**

The user exit subroutine named in the message was not found in the current directory or on the path.

03: Invalid WORK-D area.

The field attribute data is invalid or has been corrupted. This area must not be modified by your program. Check these things:

- Arguments out of order in ScreenIO call. Most common.
- Incorrect arguments in ScreenIO call.
- The WORK-D area has been corrupted by your calling program prior to calling ScreenIO (Copybook option only). Recompile with the subscript checking directive and retest. See error number 01 above.

04: Invalid WORK-D validation data.

The field validation data is invalid or has been corrupted. This area must not be modified by your program. Check these things:

- Arguments out of order in ScreenIO call. Most common.
- Incorrect arguments in ScreenIO call.
- The WORK-D area has been corrupted by your calling program prior to calling ScreenIO. .Recompile with the subscript checking directive and retest. See error number 01 above.

05: program-name **call to active program.**

You have probably made a recursive call to an already active dynamically called subroutine named program-name. This can only occur if you are using a user exit subroutine and it calls itself or some other already active program.

06: Invalid WORK-S area.

The static data area is invalid or has been corrupted. This area must not be modified by your program. Check these things:

- Arguments out of order in ScreenIO call. Most common.
- Incorrect arguments in ScreenIO call.
- The WORK-S area has been corrupted by your calling program prior to calling ScreenIO. Recompile with the subscript checking directive and retest. See error number 01 above.

Hint:

This is by far the most frequent cause of this error message. Typically you will find a runaway indexing problem in a data area that resides immediately prior to the WORK-S area. The easiest way to test this is to place a large (1000 bytes or so) filler area ahead of the WORK-S area and see if the problem disappears. If it does, you have an indexing problem: Use the debugger to see if the WORK-S area is modified unintentionally by your program before you call us!

07: Memory control blocks destroyed.

We have never actually encountered this one in over ten years. In fact, we don't even know what it means anymore.

09: Undefined user exit subroutine.

Your panel specified that a user exit was to be called on the current field but the exit name was not specified in *panel*-PREDISPLAY-EXIT, *panel*-POSTKEY-EXIT, or *panel*-ARROW-EXIT.

11: Program file is invalid format.

A dynamically called subroutine (user exit) .DLL file was invalid.

20: Invalid mask operation; data exceeds display length.

See the section discussing the EDITMASK subroutine. If you have modified the EDITMASK subroutine, you may have done it incorrectly. If you have not, call us for assistance.

24: Invalid mask operation; data exceeds mask length.

Always caused by modifying your masks incorrectly. Enable logging in ScreenIO and see if the masks in the event log correspond to those listed in the panel editor. If not, you just found your problem.

See the section discussing edit masks.

32: Invalid function call.

You issued an invalid function call.

Non-fatal errors:

These messages are issued when the user attempts an operation that is not permitted, or are issued purely for informative purposes. The messages are in alphabetical order.

A valid date is required in this field.

If the country code is 001 (USA), the date must be of the form MM DD YY. If Europe, the correct format is DD MM YY. If Japan, it is YY MM DD.

Data is required in this field.

User tried to exit a panel without entering non-blank data in all fields having the 'required' field processing option. Enter data in the field. You may want to set up a Bypass table (see Function 11).

Field overflowed, digits truncated.

Fields which are masked using a mask that contains insertion characters are longer than the PICTURE clause of the data item. If you fill all of the characters of such a field with data, some of that data will necessarily be discarded by the masking routine when the mask operation takes place. Since your fields will normally be large enough to contain any valid data the user may enter, this is purely a warning message.

Must be: validation-list

This message will be issued when the user attempts to tab out of a field that specified validation, and the data in the field does not pass the validation criteria. *validation-list* is the exact string that specifies the acceptable validation data.

Only alphabetic data is allowed in this field (A-Z).

User tried to enter non-alphabetic data in an alphabetic field. Alphabetic data includes the characters A-Z and space, which is the COBOL definition of alphabetic.

Only alphanumeric data is allowed in this field (A-Z, 0-9).

User tried to enter non-alphanumeric data in an alphanumeric field. Valid alphanumeric characters include the letters A-Z, digits 0-9, and space. This is the COBOL definition of alphanumeric.

Only hex digits are allowed in this field (0-9, A-F).

Means exactly what it says; hexadecimal fields will only allow valid hexadecimal characters (0-9, A-F).

Only numeric data is allowed in this field (0-9).

Exactly what it says, except that signs, DB/CR, decimals, and currency symbols are also allowed.

You may not enter data here - only function keys.

If the user tries to enter data on a panel with no fields, or while in a protected (Selector or Byte pointer) field, this message will be issued. Either press a function key for the desired action, or tab to an unprotected field.

ScreenIO

Appendix A: License

This license agreement constitutes an agreement between Northern Computing Consultants, hereafter "NORCOM", and the licensee of the ScreenIO software product, hereafter "YOU".

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE REMOVING THE DISKETTES FROM THE PACKAGE. OPENING THE DISKETTE PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED, AND YOUR MONEY WILL BE REFUNDED.

NORCOM provides this program and licenses its use worldwide. You assume responsibility for the selection and use of the program to achieve your intended results, and for the installation, use, and results obtained from the program.

TERM

This license is effective until terminated. You may terminate at any other time by destroying or returning the program together with all copies in any form.

This license will also terminate in the event you breach or violate any of the provisions of this Agreement. In this event, you shall, upon written demand by NORCOM or its authorized representative, return all copies of the program and documentation to NORCOM within ten (10) days.

DEVELOPMENT ENVIRONMENT:

You may use the ScreenIO panel editor (PEF16.EXE and PEF32.EXE) only on a single computer at any one time. If you are running in a multiple user environment or local area network, you must license one copy of ScreenIO for each workstation that will be using the Panel Editor;

You may copy the program into any machine readable or printed form for backup purposes in support of your use of the program on the single machine;

You may transfer the program and license to another party if the other party agrees to accept the terms and conditions of this Agreement. If you transfer the program, you must notify NORCOM in writing, and at the same time either transfer all copies whether in printed or machine readable form to the same party or destroy any copies not transferred;

DERIVATIVE SOFTWARE: Distribution License

The license granted to the purchaser of a ScreenIO license allows unrestricted distribution of your derivative software products; that is, the executable code resulting from linking the ScreenIO subroutines into your applications, subject to the following conditions:

You may freely distribute necessary runtime .DLL files provided with ScreenIO, provided that they accompany programs that you developed using ScreenIO.

You may not distribute a derivative software product which substantially duplicates the functions of ScreenIO, or which in the opinion of NORCOM, competes with ScreenIO.

You may not utilize the ScreenIO panel editor in or allow others to use the ScreenIO panel editor in any multiple user arrangement. If you are running in a multiple user environment or local area network, you must license one copy of ScreenIO for each developer workstation;

You may not offer the panel editor or ScreenIO object libraries for sharing or for distribution by resale, sublicense, or lease; except that you may transfer your copy of ScreenIO;

You may not modify the ScreenIO panel editor or runtime .DLL files in any manner.

MAINTENANCE AND SUPPORT

Maintenance and Support shall include all formal updates to ScreenIO during the Maintenance Period. It will include timely response to reported errors, problem analysis, and, at our discretion, replacement or correction of ScreenIO software.

NORCOM will provide, directly to the licensee, Maintenance and Support at no charge for one year from the date of license. Maintenance and Support following the first year will be provided in accordance with the License Agreement and maintenance fees then in effect.

LIMITATION OF LIABILITY

YOU AGREE THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU, NOT NORCOM, ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

NORCOM'S LIABILITY IS EXPRESSLY LIMITED TO REFUNDING THE RETAIL PRICE OF THE PRODUCT.

YOU AGREE THAT NORCOM WILL NOT BE LIABLE FOR ANY LOST PROFITS, OR FOR ANY CLAIM OR DEMAND AGAINST YOU BY ANY OTHER PARTY, EXCEPT A CLAIM FOR PATENT OR COPYRIGHT INFRINGEMENT. IN NO EVENT WILL NORCOM BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF NORCOM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

LIMITED WARRANTY

NORCOM warrants that the software functions substantially in accordance with the accompanying documentation. NORCOM warrants that it has full power and authority to grant the rights granted herein.

THIS CONSTITUTES THE ENTIRE WARRANTY COVERING THIS SOFTWARE PRODUCT. THIS WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE.

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

Appendix B: Key Values

Character codes and ASCII control codes

Character codes include the values returned when you press a character key (letters, numbers, punctuation, etc.) and the ASCII control codes Backspace (08), Tab (09), Enter (13), and Escape (27).

These four values are ASCII control codes. ScreenIO will return Enter to your program as a 13 in *panel*-EXIT-KEY, as though it was an extended function code. The other three keys are trapped by ScreenIO and processed internally.

If you are using function 6 for key definition, you specify these as the first value in the key definition, and the second (unused) value is zero.

08	Backspace* (Destructive Backspace)
09	Tab* (Go to next field)
13	Enter
27	Esc* (Restore original data in field)

Extended function codes

Extended function codes are returned when you press function keys and Alt- key combinations. These cause two numeric values to be sent to ScreenIO. The first is zero, and the second contains the extended function code.

This is the number returned in *panel*-EXIT-KEY if the user hits one of these keys.

If you are using function 6 for key definition, you specify the first key definition value as zero, and the second is the value below.

15	Shift-Tab* (Go to previous field)
16-25	Alt- Q,W,E,R,T,Y,U,I,O,P
30-38	Alt- A,S,D,F,G,H,J,K,L

44-50	Alt- Z,X,C,V,B,N,M
59-68	F1 through F10
71	Home* (Go to Home field)
72	Cursor Up*[1] (Same as Shift-Tab)
73	Page Up
75	Cursor Left* (Move cursor within field)
77	Cursor Right* (Move cursor within field)
79	End
80	Cursor Down* (Same as Tab)
81	Page Down
82	Ins* (Toggle Insert mode on/off)
83	Del* (Delete character at cursor)
84-93	Shift-F1 through F10
94-103	Ctrl-F1 through F10
104-113	Alt-F1 through F10
115	Ctrl-Cursor Left
116	Ctrl-Cursor Right
117	Ctrl-End* (Delete to end of field)
118	Ctrl-Page Down
119	Ctrl-Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0,-,= (Top row)
132	Ctrl-Page Up
133-134	F11, F12

[1] Cursor positioning key values (cursor up, down, left, right) are only returned if you select the appropriate field processing option.

135-136	Shift-F11, F12
137-138	Ctrl-F11, F12
139-140	Alt-F11, F12

*These keys are trapped and handled internally by ScreenIO.

ScreenIO