Fitrix™
CASE Tools
Enhancement Toolkit ♦
Technical Reference
Version 4.11

Fourth Generation Software Solutions
2814 Spring Road, Suite 300
Atlanta, GA 30339

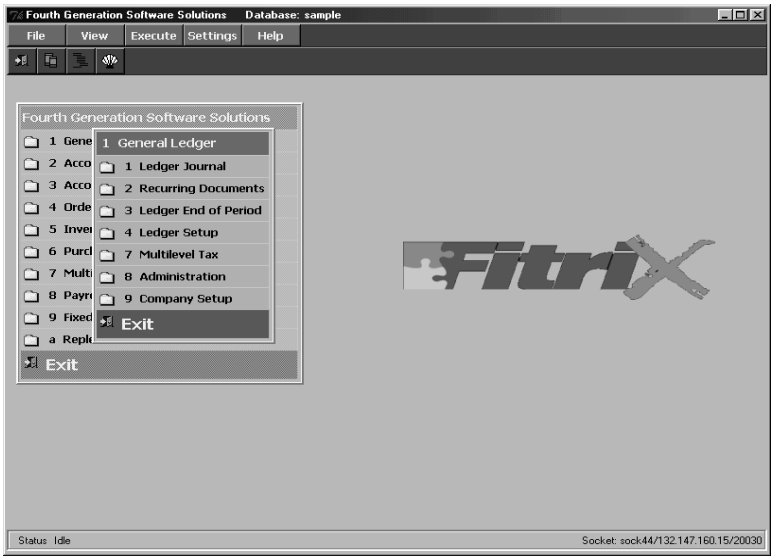
Corporate: (770) 432-7623
Fax: (770) 432-3448
E-mail: info@fitrix.com

Copyright

Copyright (c) 1988-2002 - Fourth Generation Software Solutions Corporation - All rights reserved.

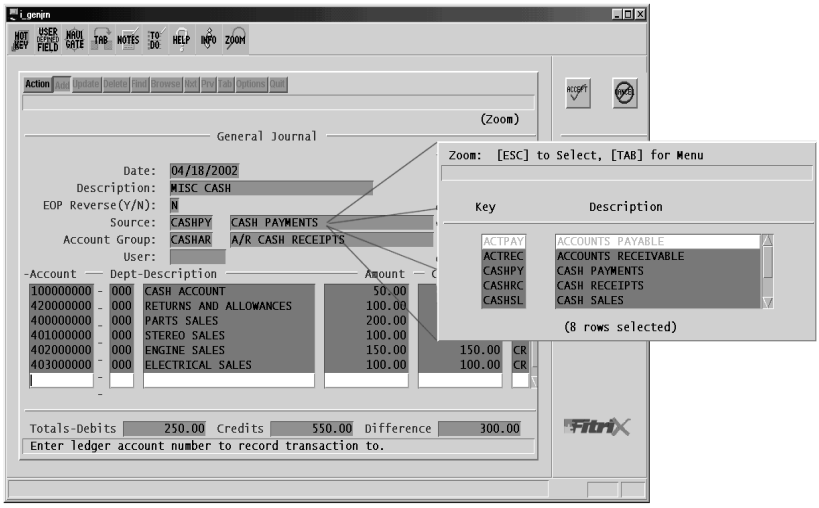
No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated.

Welcome to the Fitrix Enhancement Toolkit Technical Reference. This manual is designed to be a focused step-by-step guide. We hope that you find all of this information clear and useful.
All of the screen images in this document are show with the products using the character user interface. While the Fitrix Rapid Application Development (RAD) Tools operate in character mode only, the software applications created by the RAD tools offer the option of being viewed in a graphic based Windows (or X11) mode as well as the character mode shown. Examples of graphic based product viewing modes are shown below in Example 1 and Example 2.



Example 1: Menu Graphical Windows Mode

Here is another example:



Example 2: Data Entry Graphical Windows Mode

Displaying our products in graphic mode, as shown in Example 1 and Example 2, is customary for many Fitrix product users.
However, your viewing mode is a user preference. Changing from character based to graphical based is a product specific procedure, so if you wish to view some applications in character mode, and some in graphical mode, that can be done as well.
If you have any questions about how to view your products in graphical mode, please consult your Installation Instructions or contact the Fitrix helpdesk at 1(800)374-6157. You can also contact us by email: support@fitrix.com. Please be prepared to offer your name, your company, telephone number, the product you are using, and your exact question.
We hope you enjoy using our products and look forward to serving you in the future.

Thank You,
Fourth Generation

Table of Contents

Chapter 1: Introduction

Chapter 2: User Control Library

Chapter 3: Pull-Down Menus

Chapter 4: Program Control Library

Chapter 5: Fitrix Security

1

Introduction

This chapter introduces you to the Enhancement Toolkit and covers the following:

- Features of the Enhancement Toolkit
- Documentation conventions

Enhancement Toolkit Overview

Enhancement Toolkit is a collection of features you can add to programs created with Fitrix *CASE* Tools: Fitrix *Screen* and Fitrix *Report*. Enhancement Toolkit comes with precompiled libraries and programs.

Some of the features in Enhancement Toolkit are added to your programs simply by linking in a special library. Other features are in the form of library functions which can be called from your own programs.

In order for your programs to take advantage of any Enhancement Toolkit feature, Enhancement Toolkit must be purchased for each run-time system.

Enhancement Toolkit Features

The Enhancement Toolkit contains two categories of features: end user and developer. End-user features are found in the User Control Library, while developer features are found in the Developer's Toolbox.

The User Control Library

The User Control Library adds a variety of features that give the end user control over applications created with Fitrix *Screen*. Your applications can utilize many of these features just by compiling this library with your program. The User Control data-entry features are designed to expand the usability of your application.

Data-entry features found in the User Control Library package allow end users to:

- Navigate to any function or program inside or outside of the current program.
- Map a number of function keys that can perform a variety of functions.

- Create, modify, and use help text.
- Create, modify, and use error text.
- Define up to 50 new fields per document.
- Create free-form notes to supplement any document.
- Create and maintain a personal to-do list.

The Developer's Toolbox

The Developer's Toolbox contains three kinds of tools: a library of useful C functions, a security module, and a graphical application menuing system.

Program Control Library

The Program Control Library contains a variety of useful functions designed to give the programmer even more flexibility when creating programs. The Program Control Library features allow you to do the following:.

- Create dynamic menus.
- Create dynamic ring menus.
- Create scrolling input fields.
- Create warning windows.

Application Security

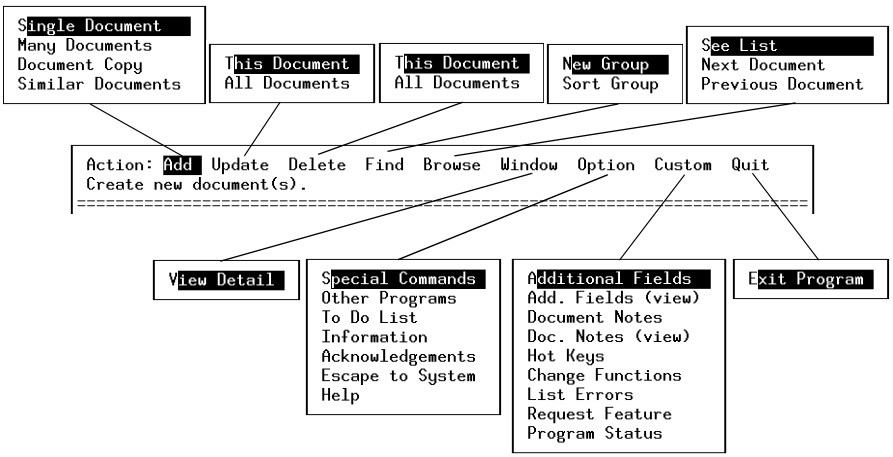
The security module lets you secure input programs created with Fitrix *Screen,* output programs created with Fitrix *Report*, and any option on Fitrix *Report* Writer menus. You can set up security for your programs by individual, group, or default. You can restrict access to modules, programs, or events.

Pull-Down Menus

Applications generated by the Code Generator can utilize an optional pull-down menuing system in place of the standard application ring menu. Pull-Down Menus:

- Give your applications a modern appearance.
- Let you fit more menu items on pull-down menus than you can fit on a standard "flat" ring menu.
- Make all User Control Library features visible to the user.
- Let you add and subtract menu items on an program-specific basis.
- Allow you to remove menu items that have no use in a particular program.
- Allow you to turn on or off any menu item for any program.
- Make multilingual programs easier to maintain.
- Allow you to change any menu item to familiar terminology. For example, if your existing programs use Erase instead of Delete, you can easily change a pull-down menu to say Erase.

The Pull-Down Menu System:



Enhancement Toolkit Documentation

The Fitrix CASE *Tools Enhancement Toolkit Technical Reference* contains documentation for features available with the Enhancement Toolkit. It is organized by chapter as follows:

Chapter 1: Introduction—an overview and a brief look at the features available in the package**.**

Chapter 2: User Control Library—covers several user-definable features that enhance the generated application.

Chapter 3: Program Control Library—explains advanced functions that can be used to customize your application.

Chapter 4: Pull-Down Menus—describes an advanced program menuing system.

Chapter 5: Fitrix Security—explains how to use the security utility.

Documentation Conventions Used in This Manual

Although many similar versions of UNIX and XENIX may run INFORMIX-4GL and the Code Generator, the manual refers to this general category of operating systems with the single term UNIX.

Some information is difficult to convey in words, such as a series of keystrokes or a value you supply. This manual uses several conventions to convey information that has special meaning. These conventions use different fonts, formats, and symbols to help you discern commands, program code, file names, and keystrokes from other text.

Text Format	Meaning	Example
Courier Bold	Represents command syntax in addition to variable and file definitions.	**fg.screen**
Courier Bold Italic	Represents text you should replace with the appropriate value.	**-dbname *database_name***
Courier	Represents commands; file, directory, table, and column names; and computer responses.	header.4gl Makefile stxhelpd $fg/bin
Small Courier	Represents program code or text in a file.	#################### function llh_add() #################### # This function inserts

Symbol	Meaning	Example
[ ]	Represents optional command flags and arguments.	**fg.screen [-yes]**
...	Represents command arguments that can be repeated.	***filename*** **...**

When not part of an explicit instruction, single keyboard characters, field values, and prompt responses are shown in uppercase. For example:

Choose Y or N.
Enter an A for ascending or D for descending.
Press Q to quit.

Named keys, such as Tab, are shown in uppercase and enclosed in brackets.

[TAB]
[CTRL]
[F1]
[ESC]
[ENTER]
[DEL]
[SPACEBAR]

When a series of keys should be entered at the same time, they are shown with a hyphen connecting them. For example:

To close the menu, type [CTRL]-[d].

Some key names are not consistent from keyboard to keyboard. This manual makes repeated mention of the [ENTER] and [DEL] keys, but both of these may be missing entirely from some keyboards. Different hardware manufacturers give different names to keys that perform the same functions. In addition to the keyboards themselves, software-controlled settings in terminal control files may also alter the interpretations of keystrokes.

The table below lists keys that are named differently on different keyboards.

KEYS	COMMONLY USED VARIATIONS
ENTER	RETURN, RTRN, ↵
ESC	STORE
DEL	BREAK, CTRL-C, CTRL-BREAK

2

User Control Library

This section describes the added functionality of the User Control Library available with the purchase of the *CASE* Tools Enhancement Toolkit. The added features in the User Control Library give both the user and the programmer further control over the application. This section covers:

- The Navigate Feature
- Hot Keys
- Online Help
- Error Text
- User-Defined Fields
- Freeform Notes
- Personal To Do List

The Navigate Feature

The user-definable Navigate feature allows you to interrupt operation of a Code Generator generated program at any point and "jump" or "navigate" to any other program on the system and then return to your original position. You can navigate to the desired program through a special internal navigate menu or by defining a function key that automatically loads the desired program. The navigation feature prevents you from having to quit your current program and step through a number of menu options in order to make a simple change elsewhere, then make your way back to your starting point. You can define your own shortcuts from one program to another.

The Navigate menu gives you the option of selecting from any number of things to do outside of the current program. Items may include reading mail, printing a report, loading another application, and anything else that can be done at the operating system. You can also select any "event" that is internal to the program such as help, zoom, or view notes.

"Local" events can be added to any program. An example might be an "update customer" event that runs the customer update program & positions you on the current customer. Another might be a "calculator" event that calls on an internal calculator function returning the data into the current field.

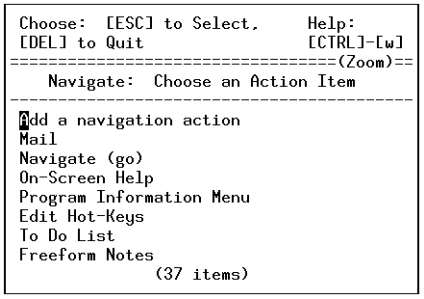
You can assign all navigation events to special keys, which you can then execute by the press of a button. Assigning navigation events to Hot Keys is covered on page 2-7.

Navigation is a feature built into the User Control Library. You must have the *CASE* Tools Enhancement Toolkit on your system in order to use the Navigation feature.

For more information on the philosophy of our event handling logic, refer to the Fitrix Screen *Technical Reference*.

The Navigate Menu

When you press [CTRL]-[g] the Navigate menu is displayed. You can select an event from the Navigate menu by moving the highlight to the desired item then pressing [ESC] or [ENTER].

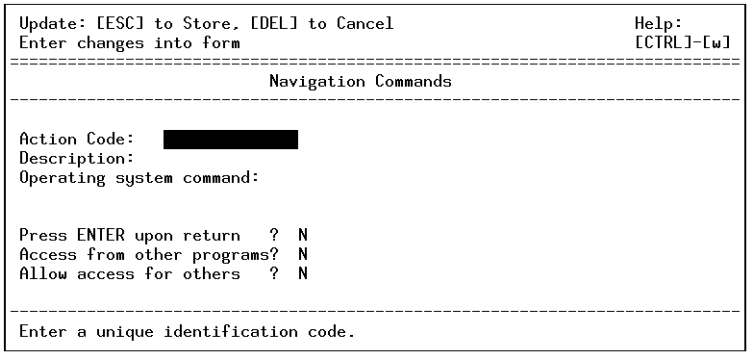


All established Navigation events appear on the Navigate menu. The items on the Navigation menu are ordered as follows:

1. The Add a navigation action option is always listed first.
2. User-defined events ordered by the action code entry on the navigation screen.
3. Hardcoded ordering of internal navigation events.

The Navigation Commands Form

The Add a navigation action event, which is the first item on the Navigate menu, lets you add custom events to the Navigate menu. When you select Add a navigation action, the Navigation Commands form appears:



The fields found on the Navigation Commands form are as follows:

Action Code: This 15-character field is intended to be used as a shorthand method of referring to the action itself. Some action codes are reserved as basic navigation events (established during code generation).

Description: This field stores a description (up to 30 characters) of the action. The description specified represents the action on the Navigate menu.

Operating system command: This field stores the operating system command for actions that are external to the current program. For instance, to create an event that checks for E-mail, enter the following text into this field:

```
mail
```

This field is left null for actions that are considered internal to the program. Examples of internal actions include: Zoom, Browse, User-Defined fields, Freeform notes, and the Program Information menu.

Press [ENTER] upon return: This one-character field accepts an entry of Y or N (Yes/No). The entry determines whether you must press the [ENTER] key upon returning from an action. The entry in this field is for external events; that is, no entry is required if the event being defined is internal. The default value for this field is N.

Access from other programs: This one-character field accepts an entry of Y or N (Yes/No). An entry of Y indicates that this action is available for use from other programs within the application. That is, if you have an orders program as well as a customers program within the same general application, you can use this navigation event from either program. In short, the entry in this field determines whether this navigation event is system-wide. The default value for this field is N.

Allow access for others: This is another one-character field that accepts an entry of Y or N (Yes/No). It determines whether other users have access to the navigation event you define.

Navigating to Another Program

Navigation allows you to interrupt a program at any point and start up a new program without losing your place in the original program. Once you quit the secondary program, you are returned to your original program at the point you left. The following steps explain how to set up a navigation event that starts another program.

1. **Press [CTRL]-[g] to display the Navigate menu.**
2. **Select Add a navigation action from the Navigate menu.**

   The Navigation Commands form appears.

3. **Enter the event name in the Action Code field.**

   For example, if the event initiates a program called Customer, enter customer in this field.

4. **Enter a description for the event in the Description field.**

   For example, a description of the Customer program might be "Run customer program."

5. **Enter the operating system command for the event in the Operating system command field.**

   Suppose that for the Customer program example, the event runs `i_cust.4gs/*4gi`.

   There are two ways to use navigation to run a program:

   1. Use Fitrix *Menus* to run the program.

      If you are using Fitrix *Menus* and the program you want to run is defined on a menu somewhere, you can start the program with the `mz -i` command. This method is preferred because your `$DBPATH` gets set automatically, and you do not have to specify whether to run a `*.4gi` or a `*.4ge`. Refer to the Fitrix Screen *Technical Reference* for more information on using the `mz` command.

   2. Run the program executable directly.

If the program you want to run is not on a menu, type something similar to the following on the command line:

```
cd $fg/accounting/gl.4gm/i_genjrn.4gs;fglgo *4gi
```

or

```
cd $fg/accounting/gl.4gm/i_genjrn.4gs;*4ge
```

Notice that you must first change directories to the program directory.

Deleting a Navigation Event

You can also delete defined navigation events.

To delete a navigation event:

1. **Press [CTRL]-[g] to display the Navigate menu.**
2. **Highlight the event name from the Navigate menu.**
3. **Press [CTRL]-[z] to display the Navigate Commands form.**
4. **Press the [DEL] key and at the prompt answer Y to verify deletion.**

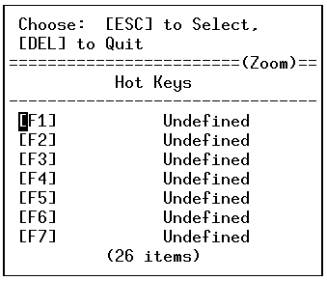
Hot Keys

As explained in the previous section, navigation events can be easily tailored to carry out internal or external events from any place within the current application. All established events can be executed by simply selecting them from the Navigate menu.

The Hot Keys feature extends the power of the Navigation feature by allowing users to assign tasks to specific keys. Instead of calling up the Navigate menu and selecting an event, you need only press the key corresponding to that event. In order to assign an event to a key, the event must be defined as a navigation event. You can only assign Hot Keys to events appearing on the Navigation menu and set up for your use.

Mapping Hot Keys

The Hot Key form lets you view Hot Key definitions and assign keys for defined navigation events. From within an application, press [CTRL]-[e] to display the Hot Key form:

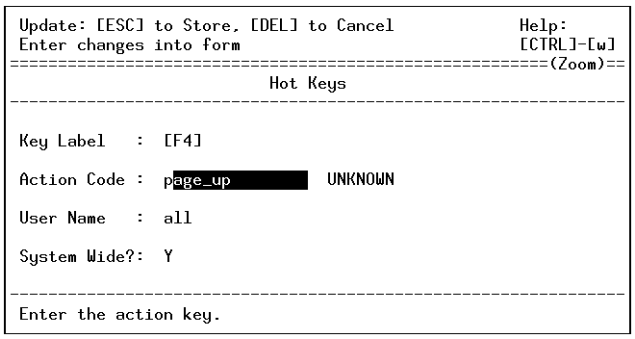


The Hot Key form, also accessible through the Program Information menu, serves as a reference for the current Hot Key settings. Use the arrow keys to scroll, or [F3] and [F4] keys to page through the current definitions.

Each row represents a key and the event with which it is currently associated. When you highlight a key definition and press [ESC], the associated event is carried out. For instance, if the user selects the row containing the event Freeform Notes, the Freeform Notes form is made current on the screen. In effect, selecting a definition

on this form is the same as Navigating to the corresponding event. Clearly, it is still faster to simply press the key or keystroke combination with which the event is associated.

To edit a Hot Key definition, highlight the definition and press [CTRL]-[z]. This action displays the Hot Keys Definition form:



The following fields appear on the Hot Keys Definition form:

Key Label: This field displays which key the Hot Key corresponds to. This is a no-entry field.

Action Code: This field stores the code representing the action to be mapped to the key displayed in the Key Label field. The action you specify in this field must be set up through the Navigate Commands form (see "The Navigation Commands Form" on page 2-3). Press [CTRL]-[z] to see a list of valid action codes.

User Name: This field stores the login name of the user for whom this Hot Key operates. You can specify "all" in this field to enable all users to use this Hot Key.

System Wide: This is a (Yes/No) field that accepts a one character entry of either Y or N. The entry determines whether the Hot Key is enabled for other programs within the general application or just for the program currently being run.

As soon as the Hot Key definition is stored, the Hot Key is available for use.

Key Mapping and Termcap

Generated code, code created with Fitrix *Screen* Code Generator or *Report* Code Generator, disassociates keys from the functions they perform. The termcap setting affects the way the terminal, and therefore, the code, recognizes and acts upon keyboard input. Depending on your termcap setting, you may not have the ability to map particular keys to events and use them as explained in this section. For additional information on termcaps, please refer to the Fitrix Screen *Technical Reference*.

If the keys on your keyboard behave abnormally, make sure your termcap is set correctly.

Defining Additional Hot Keys

If you want access to a certain key on the keyboard that INFORMIX-4GL does not have defined, you can assign it to an unused function key.

INFORMIX-4GL offers 36 function keys, and most keyboards do not have that many function keys. The following steps are necessary for defining the [SHIFT]-[F1] key on an ANSI terminal:

1. Figure out what the [SHIFT]-[F1] key sends. (ANSI example: \E[Y).
2. Pick an unused function key and define it in your termcap: (example: [F13]) (INFORMIX-4GL uses k0-k9 for [F1]-[F10], and kA-kZ for [F11]-[F36])

   ```
   [F13] ====> :kC=\E[Y:
   ```

3. Define the key in the database:

   ```
   insert into stxkeysr values(113, "[SHIFT]-[F1]")
   ```

4. If the function key that you picked is "greater than" [F15], then you need to add this "hook" to your input statements.

   ```
   on key(f18) let hotkey=118 goto event
   ```

Now the [SHIFT]-[F1] key triggers a Hot Key event. Users can now assign it to any navigation command.

Key Mapping Conventions

- [F1]-[F4] are reserved for INFORMIX-4GL (during input) and user-defined menu functions (while in menus).
- [F5]-[F12] are reserved for the real function keys [F5]-[F12].
- [F13]-[F30] can be used to map terminal keys to function keys.
- [F31]-[F36] are reserved for Fitrix *Screen* functions. Among the reserved functions are the following:

  [F34] Hard mapped to "^B" (Back Tab)

  [F35] Hard mapped to "esc" (accept)

  [F36] Hard mapped to "int" (cancel)

Example of hooking up the "real" backtab key:

1. Figure out what the [BACKTAB] key sends. (ANSI example: \E[Z)
2. Hook it up to the termcap file for [F34] (reserved for backtab): (INFORMIX-4GL uses k0-k9 for F1-F10, and kA-kZ for F11-F36)

```
[BACKTAB] =====> :kX=\E[Z:
```

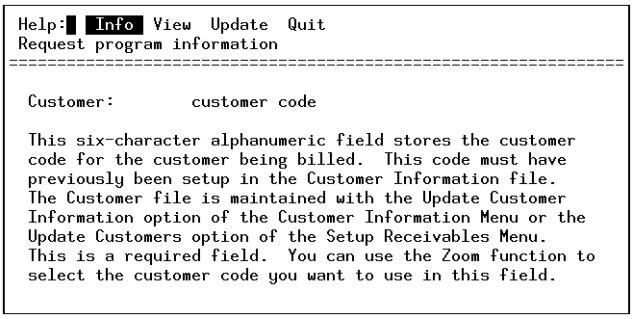
Example of hooking up an alternate "interrupt" key:

1. Figure out what the alternate key sends. (example: \E[2i)
2. Hook it up to the termcap file for [F36] (reserved for interrupt): (INFORMIX-4GL uses k0-k9 for F1-F10, and kA-kZ for F11-F36)

```
<Alternate key> =====> :kZ=\E[2i:
```

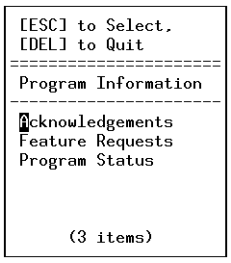
Online Help

All data-entry applications using the User Control Library contain the basic structure for online, field-specific help. Press [CTRL]-[w] to access the online help from within a data-entry form. The following form appears:



The ring menu for Help forms contains four commands: Info, View, Update, and Quit.

Info: This command leads to the Program Information menu, which contains program-specific options. The menu appears as follows:



The options found on the Program Information menu are discussed in the Fitrix Screen *Technical Reference*.

View: This command allows you to scroll through the help text displayed on the window. When you select View, the ring menu looks as follows:

```
Scroll:  [TAB], [DEL], or [ESC] to Quit
[ARROW KEYS] to Scroll, [F3] or [F4] to Page
```

Use the keys listed in the ring menu to scroll through the help text. Any of the keys listed on the top row of the ring menu return control to the data-entry form from which help was called.

Update: This command lets users update help text.

When you select Update, the ring menu looks as follows:

```
Update: [ESC] to Store, [DEL] to Cancel
Enter changes into form
```

The INFORMIX-4GL defined keys [F1] and [F2] (add a row and delete a row, respectively) are also available for use when updating help text. The [DEL] key cancels the edits and returns control to the Help ring menu. The [ESC] key stores the text for future reference.

Quit: This command returns you to the program.

Copying Help Text

A handy development tool is available for setting up help text in programs within a general application. Existing help text (in another module or program within the application) can be copied by pressing [CTRL]-[c] while at the Help ring menu. A series of prompts appear to identify the particular help text you wish to copy. The prompts begin at the module level and become more specific. The first prompt appears as follows:

```
Help:    Info  View  Update  Quit
COPY: Enter module to copy from:
```

In order to copy text you must specify the name of an existing module. For reference, the module name for any given application (generated by Fitrix *Screen*) is listed on the Technical Status form. This form is accessible through the Program Information menu. Alternatively, the section of the .per form specification file lists the module name. After entering a module name and pressing [ENTER], the prompt changes to the following:

```
Help:    Info  View  Update  Quit
COPY: Enter program name:
```

Enter the name of the program containing the help text to be copied. Again, the program name for any given application can be found by checking the Program Status, found on the Program Information menu. This information is also in the section of the .per form specification file. After entering the program name and pressing [ENTER], another prompt appears:

```
Help:    Info  View  Update  Quit
COPY: Enter Error number:
```

This is the last prompt involved in the process of copying help text. At this point, the module and program names have been entered. The unique help call number (i.e., Error number) is all that remains. Enter the number corresponding to the help text you wish to copy into the current Help form. For example, if you wish to duplicate the help text already entered for help text number 7, enter 7 at the prompt. The retrieved help text is then copied into the current help text file for storage.

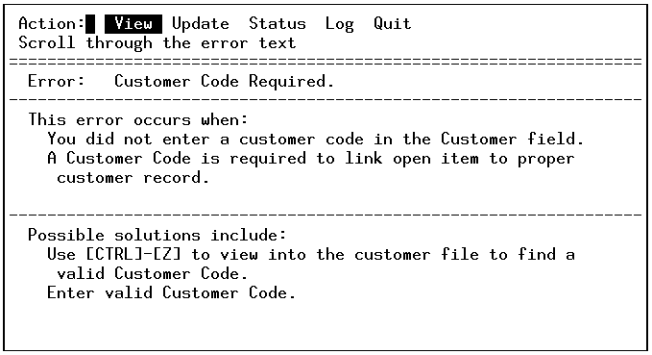
This technique is most often used to add the text for a newly-created form from which help is called. You can use this technique to copy the text from one help screen into an existing form. To do this, you must update the help text for the existing form, and then follow the copy procedure ([CTRL]-[c]) outlined above.

Online Error Text

If an error occurs, users have the option to save system status information, including the line number of the program in which the error was encountered. When logging an error, users are also given the ability to enter information that describes exactly what they were doing and how the error was encountered. Error messages, can of course, be tailored by developers to tell users what kinds of errors to log and exactly what information to enter. Error information is stored in a simple UNIX text file.

Fatal errors automatically get logged. All program status information, including error line number, is automatically logged to the error file, whether manually logged by users or not.

When an error occurs and is displayed at the bottom of the screen, you can Zoom on that error to get a problem/solution screen for the error. Pressing [CTRL]-[z] displays the following form:



Viewing Error Text

The explanation of an error appears on the form, including causes and possible solutions. The Informix scrolling keys ([F3] and [F4]) are available for movement among the lines of error text. If more text exists than is displayed, the arrow keys allow you to scroll to view the remaining text. The [TAB] key transfers the cursor from the "cause" section to the "solution" section of the error text.

Updating Error Text

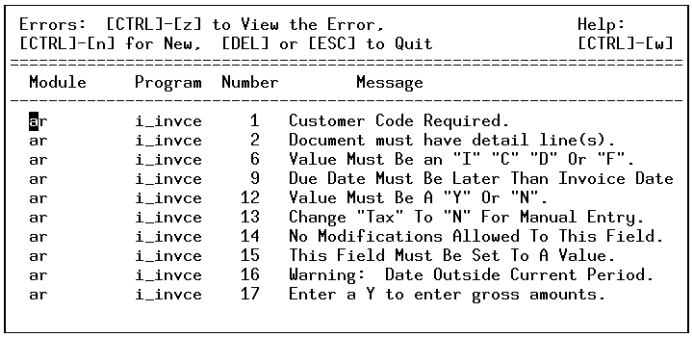
The Update command on the error detail ring menu allows the user to modify the text for a particular error call. Error messages, causes, and solutions can be modified virtually "on the fly."

The error message itself may be modified in addition to the "cause" and "solution" sections mentioned earlier. The cursor first appears in the field containing the error message title. Use the [TAB] key to switch from section to section on the error message data-entry form.

Press [ESC] to store the modifications or [DEL] to cancel them.

Adding Error Text

The User Control Library includes a navigation event that allows you to quickly create new error text. The actual error calls must be placed into the code, though the text called by the error number becomes part of the SQL database. The Edit Error Text event, which is located on the Navigate menu, displays the following form:



```
Errors:  [CTRL]-[z] to View the Error,                    Help:
[CTRL]-[n] for New,  [DEL] or [ESC] to Quit               [CTRL]-[w]
=============================================================================
  Module     Program  Number        Message
-----------------------------------------------------------------------------
  ar         i_invce     1   Customer Code Required.
  ar         i_invce     2   Document must have detail line(s).
  ar         i_invce     6   Value Must Be an "I" "C" "D" Or "F".
  ar         i_invce     9   Due Date Must Be Later Than Invoice Date
  ar         i_invce    12   Value Must Be A "Y" Or "N".
  ar         i_invce    13   Change "Tax" To "N" For Manual Entry.
  ar         i_invce    14   No Modifications Allowed To This Field.
  ar         i_invce    15   This Field Must Be Set To A Value.
  ar         i_invce    16   Warning:  Date Outside Current Period.
  ar         i_invce    17   Enter a Y to enter gross amounts.
```

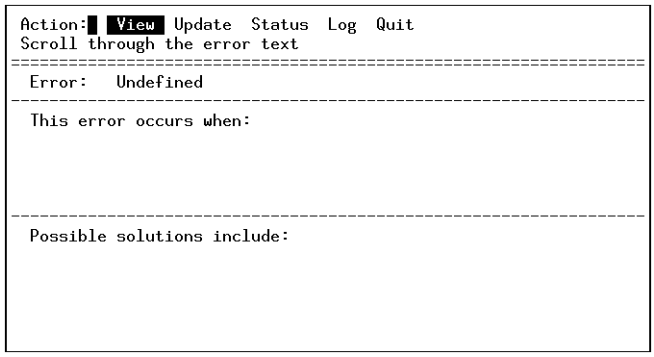
This form shows all of the errors set up for the current application. If this is a newly generated application, no errors appear on this form unless specified. The cursor appears on the first row of error text detail. The Error ring menu indicates that pressing [CTRL]-[z] displays detail information for the error. Pressing [CTRL]-[n] allows you to enter new error text. The arrow keys and the Informix-defined [F3] and [F4] paging keys are available for moving among the detail rows of this form. The data in the Number column of the form ties the error text to the actual error call found in the code.

The error text ring menu also offers the option of adding new lines of error text. For instance, after adding an error call to, say, `llh_input` within the `header.4gl` file, you want to add the corresponding error text. Previously, this meant adding the data externally, perhaps through ISQL. Code generated by Fitrix *Screen* permits this addition of information within the application itself—you can add error text for error calls without having to exit the application.

From the Errors form, press [CTRL]-[n]. A prompt appears requesting the number of the error call to be documented:

```
Enter a new error number for this module/program
or press [DEL] to quit:
```

After entering a valid error call number, the Errors Detail form appears. This form is the same as the form seen when [CTRL]-[z] is pressed:

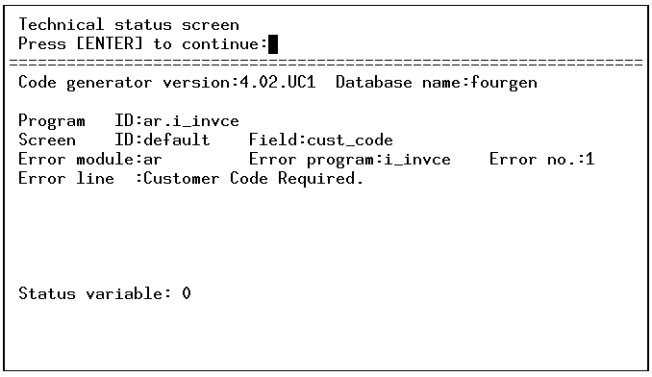


The ring menu is also identical—all commands available through [CTRL]-[z] are also available through [CTRL]-[n]. The Update command allows you to enter an error message as well as cause and solution detail. The error message appears on the screen as the error call is carried out—it is limited to one line of information.

Text is added for the detail sections as well. Use the [TAB] key to move from the error message line to the detail sections.

Viewing Program Status

The Status command on the error detail ring menu is used to display the current program status. The information appears as follows:



General information such as the database name and Code Generator version appears along with specific data regarding the error on which the cursor currently appears. At the foot of the form the Status variable value is displayed (the value 0 indicates that the program has not detected any errors).

Press [ENTER] to return to the Error Detail ring menu.

Logging Error Text

The purpose of error text is to provide additional information regarding events considered abnormal. A well-constructed application informs the user that an error has occurred. Typically, this means displaying text to the screen at the moment the error is encountered.

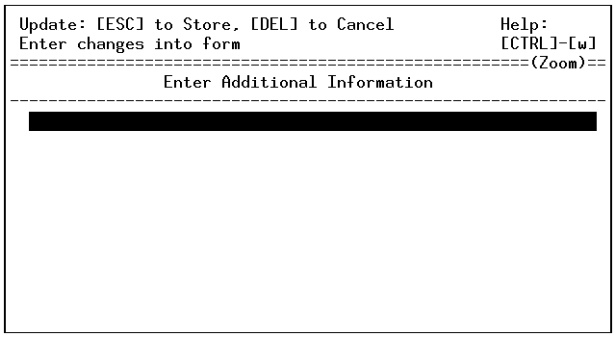
The ability to display error text is built into the code produced by Fitrix *Screen*. In addition, Fitrix *Screen* applications provide developers with a reference file to which all errors can be logged. Fatal errors are automatically logged unless they are quite severe (such as a core dump), in which case there is no opportunity to log error text prior to cessation of the program. Each application logs errors to a file titled `errlog`. The developer can periodically inspect the `errlog` file in the application directory to read information concerning errors.

Within the Error Detail form, the user can choose to log a particular error message to the `errlog` file. This is accomplished by using the Log command.

Once the Log command is selected, the user has the opportunity to add comments to the error text being logged to the local `errlog`. The following prompt appears:

```
Do you wish to add notes to this error?  (Y/N):
```

If the response is Y, a form appears for the user to fill in comments to be included with the error text. The form resembles the following:



Enter your notes into the form provided. Press [ESC] to store or [DEL] to cancel.

Error Log Zoom

The Zoom feature is provided on this form for those wishing to view or modify default (system-wide) message text. Any text specified at the default (Zoom) level automatically appears as default text when a user elects to include notes for error logging. That is, you can specify a default "testing" message on the Zoom form, which appears by default each time a user logs error text with notes.

All notes entered are appended to the `errlog` file along with the selected error text.

Once logging takes place (information is written out to the `errlog` file), the following message appears at the top of the form:

```
Error logging complete
Press [ENTER] to continue:
```

Logged error text and accompanying notes are not written to the `errlog` file until the user exits the current program. The logged text appears in the file as follows:

```
Error status log. Requested by: davidc
Codegen  version: 3.0
Database    name: stores
Program       ID: ar.i_order
Date: 07/17/90 Time: 09:38:02
Screen        ID: default
Screen     field:
Error     module: ar
Error    program: i_order
Error     number: 22
SQL error status: 0
Date: 07/17/90 Time: 09:38:02
Error text: This error occurs when:
Date: 07/17/90 Time: 09:38:02
Document already exists. Duplicate not allowed.
Date: 07/17/90 Time: 09:38:02
Possible solutions include:
Date: 07/17/90 Time: 09:38:02
You can only update this record.
Date: 07/17/90 Time: 09:38:02
End of status log
```

The duplicate Date/Time lines can be disregarded. INFORMIX-4GL automatically writes them to the `errlog` file.

The last command on the Zoom form for particular error text lines is Quit. The Quit command returns control to the basic error message form.

Copying Error Text

Error text can be copied from any module/program. The copy technique is the same as the one used to copy help text. See the earlier section on "Copying Help Text" on page 2-12.

User-Defined Fields

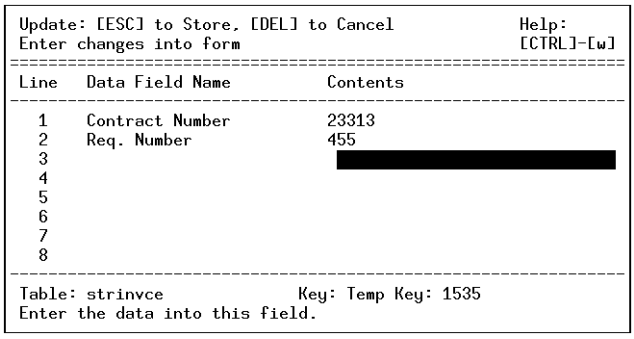
Applications generated with Fitrix *Screen* allow users to define additional fields on the fly with a special User-Defined Fields form. When users need to enter data that is not provided in current data-entry screen, they can define a field to hold that information with the User-Defined Fields form. In the future, the system administrator or programmer can determine whether such data is worth saving and integrating into a "regular" data entry screen.

Apart from those fields originally made part of the data-entry screen, up to 50 additional fields can be created and updated on User-Defined Fields form.

User-defined fields are uniquely defined for particular documents. There can be one User-Defined Fields form for each data-entry form. Once a field is defined on the User-Defined Fields form, that form is displayed automatically every time the main document is updated or a new document is added. Also, once defined, a User-Defined Field becomes a required field.

Default Hot Key access to user-defined fields is set up through [CTRL]-[f]. The User-Defined Fields form can also be accessed through the Navigate menu. Depending on the value given to the `auto_udf` variable, the User-Defined Fields form can appear automatically whenever the user stores a document (presses the [ESC] key).

The User-Defined Fields form appears as follows:



Line: This column displays the number of each row within the User-Defined Fields form. Each row constitutes a field. There are 50 fields available for use on this form.

Data Field Name: This 20-character column stores the label for each individual field. To modify the field labels on the User-Defined Fields form, press [TAB] or [CTRL]-[f].

Contents: This column stores up to 30 characters and matches up with the field label found on the same row to the left.

Note

To enter into the Data Field Name field, you must press [TAB] or [CTRL]-[f] from the Contents field.

Table: This field stores the name that forms part of the key by which the data on the User-Defined Fields form is tied to a particular document.

Key: This field stores the second part of the unique key that ties the data on this form to a particular data-entry document.

User-defined field data is maintained through the `stxaddlr` and `stxaddld` tables. For more information on tables used by the Code Generator, refer to the Fitrix Screen *Technical Reference*.

Deleting User-Defined Fields

Although any user can create user-defined fields, only programmers can remove them. User-defined fields can be removed by deleting columns from the `stxaddld` and `stxaddlr` tables using ISQL. The following shows example syntax required to delete user-defined fields:

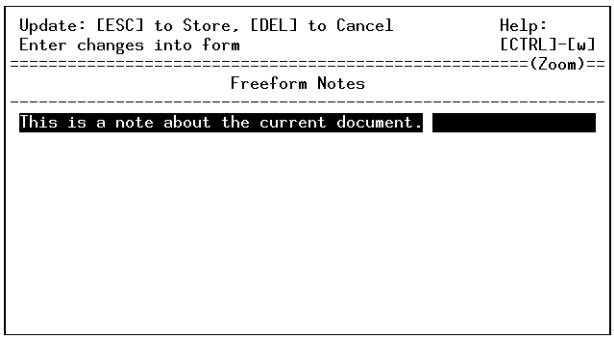
```
delete from stxaddld
  where
    stxaddld.filename = "main_table_name" and
    stxaddld.line_no = "line_number"

delete from stxaddlr
  where
    stxaddlr.filename = "main_table_name" and
    stxaddlr.line_no = "line_num"
```

Freeform Notes

With the *CASE* Tools Enhancement Toolkit, you gain the ability to attach general-purpose notes to documents displayed in your generated program.

The Freeform Notes feature is accessed by pressing [CTRL]-[n] when a document is selected.

The Freeform Notes form appears as follows:



Data-entry documents that have notes attached to them display the prompt (Notes) on the upper-right portion of the form. By simply looking at the form, users can determine whether Freeform Notes have been added for the current document.

Note

If you get the "feature not attached" error, you have not defined a unique "key" for your form. The unique key is what is used to tie secondary data such as Freeform Notes to the main table. You must go back and add a unique key to your .per form. This can be done on the Define Input Area form in the Form Painter.

The text of the Freeform Notes entered for documents within the data-entry application are stored in rows of the `stxnoted` table.

The Freeform Notes feature can only be used with header tables.

The Freeform Notes Zoom

The Freeform Notes form includes the Zoom feature, which displays the Default Freeform Notes form.

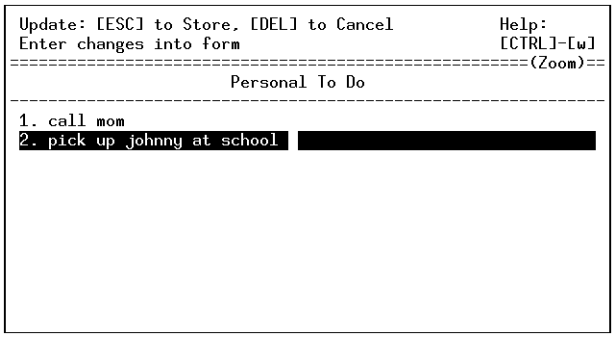
The software engineer or system administrator can use the Default Freeform Notes form to establish notes (text applicable to all documents) for initial display on forms. Default notes appear if the user accesses notes for a document that did not previously contain notes. In other words, if the user calls up notes (presses [CTRL]-[n]) for a document that previously had no notes entered, the text specified on the Default Freeform Notes form appears. The user may keep these notes, edit them, or add to them.

Personal To Do List

A common part of the data-entry process is the creation of new ideas and tasks. It is common for application users to think of tasks to carry out in the future, or to recall things that should be taken care of. While these users remain within the application, such ideas and reminders can be forgotten. Applications typically do not provide users a place for recording and reviewing personal ideas and tasks.

Applications generated with Fitrix *Screen* provide each user with a Personal To Do list. This list is part of the application, stored within an application table. Users can maintain and track this list by pressing a key, without exiting the application. The Personal To Do list operates almost identically to the Freeform Notes feature. But, rather than being form specific, the To Do list is user specific.

Access the To Do list feature by pressing [CTRL]-[t]. The Personal To Do form appears as follows:



Note that data is entered into this form in the same way it is entered on the related data-entry form.

The To Do Zoom

The Personal To Do form includes the Zoom feature, which leads to the Default To Do list.

The software engineer or system administrator can use the Default To Do list to establish tasks and reminders (applicable to all users) for initial display on forms. Default information appears when the user accesses the To Do list for the first time, or when there is no information currently saved on the Personal To Do list form. In simple terms, if the user accesses the Personal To Do list form, it displays all text previously entered onto this form by the user. If no information was previously entered, the Personal To Do list form displays the information established on the Default To Do list. The user may keep these notes, edit them, or add to them.

The text of the Personal To Do list is stored in rows of the `stxtodod` table along with the user ID.

3

Pull-Down Menus

This chapter explains how you can create pull-down menus for your application. This chapter covers:

- Pull-down menus

Pull-Down Menus Overview

The Pull-Down Menus allow you to use special ring and pull-down menus that create a much more complete and flexible data-entry environment than standard menus generated with the Fitrix *Screen* Code Generator. Pull-Down Menus allow you to use a "generic" custom menu (Mainring) which provides a number of improvements over the standard generated menus, or you can define your own specialized ring menus and pull-down menus for your individual programs. For example, the Add command on the generic custom menu calls a pull-down menu. This pull-down contains menu items that allow you to perform several different types of add. The generated version of the add command performs one task: it puts you into a blank form which you use to add a new document to your table.

Pull-Down Menus allow you to:

- Create menus by entering information into the database through a data-entry form.
- Define every aspect of menu operation by filling in screen forms.
- Create menus that offer users more options from the ring menu.
- Define how you want function keys to work when a given menu is displayed.
- Have menu items turn themselves off and on as appropriate.
- Define how you want the menus held open when items are selected.
- Create hidden menu items that respond to user keystrokes, but which don't highlight a menu item on the screen.
- Make every menu item multi-lingual so that no changes to code are required to support different languages.
- Use a "Browse" menu that allows you to use arrow keys and function keys to move through and select items from a list.
- Create ring or pull-down menu items unique to your program.

Pull-Down Menus come as a part of the Enhancement Toolkit, and programs utilizing it require the purchase of the Enhancement Toolkit for each installation. The Enhancement Toolkit need not be purchased separately if generated programs are being installed in conjunction with any Fitrix Accounting module.

Limitations:

- Up to five ring menus per application program.
- Up to 10 pull-down menus per application program.
- Up to 20 display, 20 hidden items, and 20 function keys per menu.

How It Works

Pull-Down Menus are database-driven. When you run programs using this system, menu functions go into the database to get the menu items to display for the menu the application calls. The menus use the database definition of a menu item to decide how it should behave and what functions it should call when the user selects it.

Menu items are defined by entering a description of them into the database. Two new data-entry forms are accessed from the Form Painter to allow you to define these menus. The Menu Items Definition form allows you to define generic menu items themselves. The second form, the Program Menu Definition form, allows you to make a custom version of a menu for a particular program. This second program allows you to eliminate certain commands from a standard menu, (for example, taking away the "Add" functionality) and to change the behavior of the standard menus (for example, allowing the "Update" command to update the current document instead of accessing the Update pull-down menu).

In defining generic menu items, you can define any number of different ring menus or pull-down menus. Each of these menus can call up to 20 different displayed menu items. Each menu item can be defined in up to 100 different languages. Up to 200 customized menu items can be defined for each program.

The way each menu item works is completely described in the database. This includes information about:

- whether the menu item is displayed, hidden, or in reference to a function key.
- whether or not to use this menu item based on the type of form (for example, this allows you to show some items on a header/detail form, but not a header only).

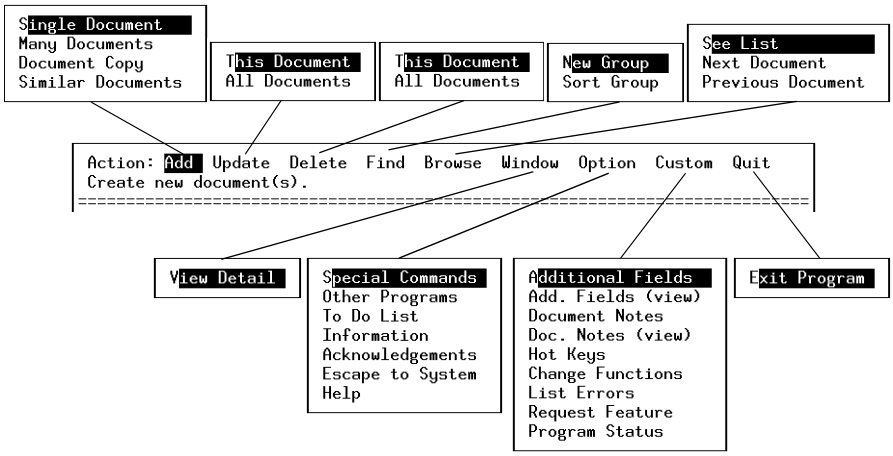
- when to turn the menu off (the user can see it, but not select it) and on (user can see and select it).

Menu items can call other ring menus, pull-down menus, or "events" that are linked to program functions. Events identify specific functions to access in either the menu function library or in your local code. The built-in events are those that duplicate existing and new file management commands, but any number of local events can also be defined.

The Pull-Down Menus come with a large number of menu items pre-defined. These include all the file management command items originally accessed by the standard ring menu system, and also a large number of new commands including all of the User Control Library commands, such as Navigate and Hot Keys. This means that many new functions can now be accessed through the ring and pull-down menu system in a generated application without any additional programming on the part of the developer. (Of course, User Control commands can still be accessed through control keys.) Among the new file management commands are "group" commands that allow users to add, update or delete entire groups of documents at once.

The default menu for all programs that use this system is called Mainring. This means that if you compile a program using the new Pull-Down Menus without specifying in the code to call another menu or defining an application specific menu, the application program calls the Mainring menu and its related pull-down menus. The pull-down menus accessed through Mainring offer access to *all* available file management commands. These pull-down menus descend from the ring commands on the initial ring menu when a command is selected from it.

The standard pull-down menu (Mainring) and related pull-downs:



Also included in the system is another menu called "Old_ring," which works, with a few improvements, just like the standard ring menu system. Finally, there is also a "Brw_ring" that allows users to use all the old browse commands. When using the "Brw_ring," however, you also are able to use the arrow keys and other function keys to move through and select items from a browse list. This is possible because you can define the way function keys work using this new menuing system.

As already mentioned, it is not necessary to use or allow access to all of these menu items from any particular application. Without defining a new menu, you can create an application specific version of a standard menuing system such as Mainring. Using the Program Menu Definition data-entry form, you can select which menu items you want to appear in your menuing system. You use this program to load a starting ring menu and its related pull-downs. You can then delete or change the function of menu items as needed in that application. This makes it possible, for instance, to eliminate unneeded menu items or change menu items so that they call certain events directly instead of going through pull-down menus.

Linking In the Pull-Down Menu System

This section explains what you need to do to take advantage of the Pull-Down Menus. In order for your programs to utilize the Pull-Down Menus the following setup steps must be taken.

To create a program using the new menus, you must:

1. Install the Form Painter and its libraries.

2. (Re)Compile programs using the advanced libraries.

Installation of the new version of the Form Painter is covered in your installation instructions, provided separately.

To compile the advanced libraries into new or existing programs, use a special version of our `make` utility.

To run the new programs, you must:

1. Install the Enhancement Toolkit on the target machine.

2. Create a custom 4GL runner on the target machine.

The Enhancement Toolkit comes as part of the Fitrix *Screen* development environment. This step is only required if you are installing the resulting program on machines other than your development platform. The Enhancement Toolkit is platform specific and must be purchased for the type of hardware the programs are going to run on.

The runner is the Informix program that knows how to run an INFORMIX-4GL program. Since the Pull-Down Menus require functions not normally part of the INFORMIX-4GL language, those functions must be linked into the runner before they can be used. Our organization provides special utilities to automate this runner creation.

Compiling Programs with Advanced Libraries

No special programming needs to be done within the program to replace the original one-dimensional ring menus with the new ring/pull-down menu system. After compiling with the advanced libraries, your programs instantly have direct menu access to a host of powerful new add, update, delete, and find options. Your programs also receive the new, more intuitive Browse ring menu. And finally, they get direct menu access to all of the powerful User Control Commands, which allow users to customize their program and access a variety of other tools that make working on a computer system easier.

To link in the new libraries, run an `fg.make` passing the library name `scr.adv` in your local directory. This adds the `scradv` libraries to the list of libraries in your local `Makefile`.

Example:

```
fg.make -L scr.adv
```

Creating a New 4GL Runner

When using our C functions, which include Pull-Down Menus, and RDS code, you have to run the finished programs with a modified runner `fglgo` and modified `fgldb`. There is an Informix utility named `cfgldb` and `cfglgo` that uses `fgiusr.c` to create these custom runners.

The creation of a custom 4GL runner requires you to link in the special C functions that these new menus require to allow users to point and pick menu items. These C functions come precompiled as part of the User Control Libraries, but you must link them into your program. This does *not* have to be done on a program by program basis. It is done only once when using the RDS (Rapid Development System) when you build a special `4glgo` program that runs the generated applications.

The `$fglibdir/lib/c_lib.4gs` directory is installed with the Enhancement Toolkit. This directory contains the following files: `mkrunners`, `README`, `fgiusr.c`, and `c_*.o`. These files are needed in order for the client to create their custom `fglgo` and `fgldb` executables.

To create a custom runner simply change directories to `$fglib-dir/lib/c_lib.4gs` and type:

`mkrunners`

It is important that `$fg/bin` comes *before* `$INFORMIXDIR/bin` in the environment variable `$PATH` setting. Again, these custom runners are only created if you develop or run programs under RDS.

Note

A common cause for the failure of `cfglgo` is that no C development system is installed.

Note

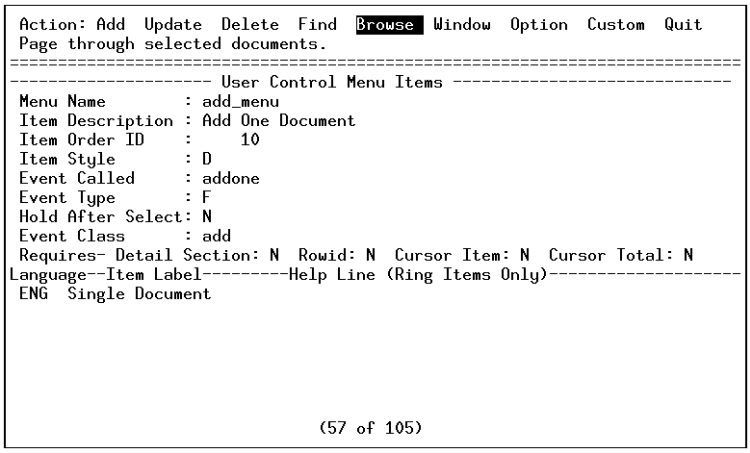
Due to the many variations of both the C compiler and the linker between all the UNIX platforms, our organization does not support any problems you may incur using these custom runners.

Creating New Menu Items

Pull-Down Menus let you customize the default ring menu, modify default pull-down menu items, or create your own entirely new menus and menu items. You can also specify different menus for different programs.

You access the program for doing this from the Form Painter using the Ring Menu Items option on the Define pull-down. This option calls up the Menu Item Definition form, which allows you to define new menus, menu items, and the way they work.

The Ring Menu Items option displays the Menu Items Definition form:



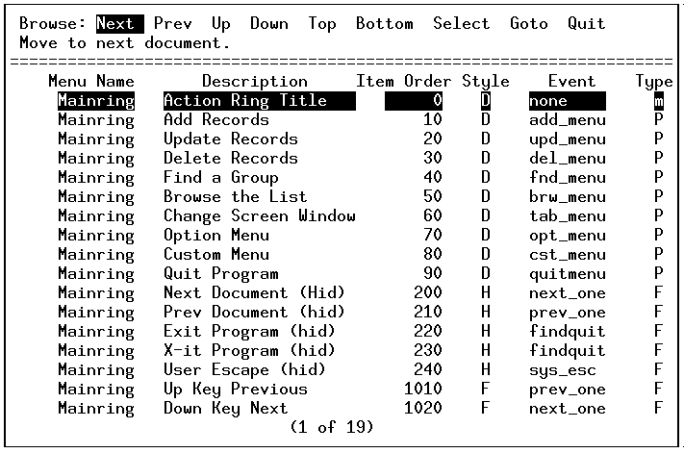
Overview of the Default Mainring Menu System

The Menu Items Definition program itself uses the default Mainring menu system, which uses all of the pre-defined menu items. This Mainring menu is the default menu created simply by linking in the advanced libraries. This ring menu can be used by all your programs by linking in the libraries. The commands in the ring menu all call pull-down menu items. By browsing through the commands and sub-menus on this menu bar, you can familiarize yourself with the functionality of the Pull-Down Menus.

You can also look at the commands (menu item descriptions) that create this menu since this is the program in which those commands are defined. To look at these commands as they are defined in the database table, use the Find command to display the Find pull-down and select the New Group option. (This type of selection is referred to as Find/New Group later in this documentation to indicate a ring menu command selection followed by a pull-down menu option selection.)

Standard (Mainring) Ring Menu Items

If you type Mainring into the Menu Name field of the Menu Items Definition form after executing the Find/New Group command and press [ESC], the system displays the first of the menu items related to the Mainring menu. To look at all the menu items on this list, use the Browse/See List options.



```
Browse: Next  Prev  Up  Down  Top  Bottom  Select  Goto  Quit
Move to next document.
=====================================================================
    Menu Name       Description      Item Order Style    Event     Type
     Mainring   Action Ring Title            0     D    none        m
     Mainring   Add Records                 10     D    add_menu    P
     Mainring   Update Records              20     D    upd_menu    P
     Mainring   Delete Records              30     D    del_menu    P
     Mainring   Find a Group                40     D    fnd_menu    P
     Mainring   Browse the List             50     D    brw_menu    P
     Mainring   Change Screen Window        60     D    tab_menu    P
     Mainring   Option Menu                 70     D    opt_menu    P
     Mainring   Custom Menu                 80     D    cst_menu    P
     Mainring   Quit Program                90     D    quitmenu    P
     Mainring   Next Document (Hid)        200     H    next_one    F
     Mainring   Prev Document (hid)        210     H    prev_one    F
     Mainring   Exit Program (hid)         220     H    findquit    F
     Mainring   X-it Program (hid)         230     H    findquit    F
     Mainring   User Escape (hid)          240     H    sys_esc     F
     Mainring   Up Key Previous           1010     F    prev_one    F
     Mainring   Down Key Next             1020     F    next_one    F
                           (1 of 19)
```

This list shows us a good selection of the various items in the Mainring menu pull-downs. In the columns of the form, we first see the name of the menu, then a description of the item, the Order/ID of the item, then the style of the item: D for display, H for hidden, and F for function keys. The Event that the menu item calls. These strings refer to either the ring menu, pull-down menu, or functional event names. The Type column refers to the type of event: P for pull-downs and F for functions.

On this menu, we see one special item, the Menu title, which is identified by an order of 0. This item is unique to ring menus and defines the string that appears before the first selectable item on the menu (such as Action: or Browse:).

There are nine selectable items that display for the Mainring. All of these items call pull-down menus. The order in which they display is indicated by the Order column. This column shows increments of ten, but that is only to facilitate adding future menu items at some other time. There are five hidden items, all of which call function events. Here, the order is unimportant and the order number simply serves as a unique ID. There are also two function key items (plus two others we don't see) all of which also call functional events. With these two, the order is also unimportant.

When a menu item is selected by the user, it can do one of two things: call another menu or call a functional event. When it calls another ring or pull-down menu, the menuing program looks for a ring or pull-down menu that uses the event name. When a menu item is selected that calls a functional event, the program places the name of that event into the global variable `scratch` and searches for it first internally, in the menuing program, then in a special function in the local program.

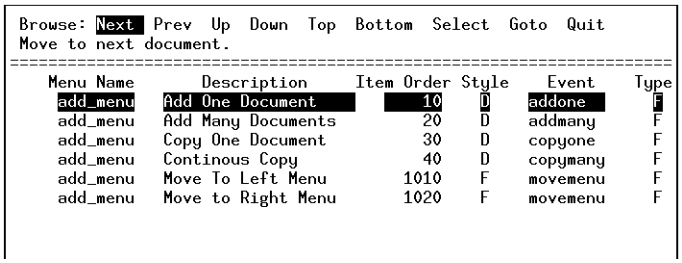
As you can see, several items can call the same event. For example, the Next Document item, which in English is invoked by pressing N for next, calls the `next_one` event. So does pressing the "Down Key" or down arrow function key. Both of these keystrokes have the exact same effect.

Typically, *displayed* menu items each call different events, but hidden and function keys call events found elsewhere in the menuing system. They can be thought of as short cuts to certain user actions. For example, the Next and Prev commands no longer display on the Mainring menu (as they do on the standard ring menu created by the Code Generator, but they do display on the `brw_menu`.

With the Pull-Down Menus, there is even a special built-in function (called `findevent`), which allows you to programmatically highlight a given menu item based upon the event it calls. This is the function used by the event `findquit` to highlight the quit command when other exiting keystrokes are used.

Standard (Mainring) Pull-Down Menu Items

To see how the items that make up a pull-down menu differ from a ring menu, execute another Find to get all of the items that make up the Add pull-down menu. Enter "add_menu" in the Menu Name field on the Find selection criteria form.



The Add pull-down is called when you use the Add command on the Mainring. As you can see, it is a simple menu. This is more or less typical of a pull-down menu. There is no "0" item, because pull-downs don't have a title line. The first four items are all display-style items. They are followed by two function key items. All of these items call functional events.

The Menu Items Definition Form

This form is displayed by selecting the Ring Menu Items option on the Define pull-down. To create new menu items to display on either ring or pull-down menus, use the Menu Items Definition form. This form gives you access to the various menu items that have been defined in the system. This program allows you to tell the menus program about new menu items you want to appear on existing menus or on

new ones and how those menu items work. This is done by simply entering the fifteen characteristics that define a menu item and how it works. The following is the Menu Items Definition form:

```
 Action: Add  Update  Delete  Find  Browse  Window  Option  Custom  Quit
 Create new document(s).
================================================================================
---------------------- Menu Items Definition -----------------------------
 Menu Name        : Brw_ring
 Item Description : Down One Page
 Item Order ID    :     40
 Item Style       : D
 Event Called     : brwsedwn
 Event Type       : F
 Hold After Select: N
 Event Class      : browse
 Requires- Detail Section: N  Rowid: N  Cursor Item: Y  Cursor Total: Y
Language--Item Label---------Help Line (Ring Items Only)--------------------
 ENG  Down                  Move down one page.









                                (5 of 105)
```

Menu Item Functionality Characteristics

The first nine of these characteristics determine how the menu items present themselves and how they run when selected.

Note

Some fields accept a value of S. This means that the value in that particular field can be determined programmatically. You determine what the value is in a field for a particular program by running the Program Menu Definition form, finding the program, then changing the value of the field for that program.

Menu Name: This character field contains the name of the menu the item being defined appears on. Each menu item belongs to a specific ring or pull-down menu.

Menus must be defined independently from defining menu items. Menus are defined in much the same way that you define a menu item. To define a menu, you need to fill out the Menu Items Definition form, and enter "P" or "R" in the Event Type field.

For example, entering "add_menu" in this field assigns this menu item to the Add pull-down menu.

Item Description: This field contains a description of the menu item. This description is for your information and helps you keep track of your menu items. It is especially helpful when browsing your menu items. The description entered in this field appears on the Browse form called from the Menu Items Definition form. This description doesn't affect how the menu item runs and does not display anywhere but in the User Control Menu programs.

For display and hidden items, it is a good idea to keep the beginning of the Description similar to the entry in the Item Label field. Another convention is to refer to hidden items with a (hid) in parentheses. For function key items, it is a good idea to say "Key" in their description.

For example, the Add pull-down menu has a menu item called "Single Document." The Item Description created for that menu item is "Add One Document."

Item Order ID: This numeric field is a unique identifier that, together with the Menu Name, creates a unique ID for the menu item. For displayed menu items, this characteristic also determines the sequential position of this menu item as it is displayed. Lower numbered items appear first in the menu.

If you think you might be adding items at a future date, leaving a lot of numeric "space" between items saves you a lot of tedious renumbering in the future. The convention is counting by tens. Convention also suggests starting Hidden Items at 200 and function keys at 1000.

For example, the Add One Document standard menu item has an Item Order of 10, while the Add Many Documents menu item has an Item Order of 20. This means that the Add One Document item appears on the Add pull-down menu before the Add Many Documents item.

Item Style: There are four possible "Styles" of items:

- **D** (display): items that are seen on the menu,
- **H** (hidden): items that work as keystrokes but are not displayed on the menu itself,
- **F** (function key): items that define how function keys are handled within the menu, and

- **S** (special item): items that only appear in some programs and not others.

For example, the Add/Single Document menu item has a D in this field, which means that item is displayed on a menu.

Event Called: This is the action that takes place when the menu item is selected by the user. Events can be of three different types (see the Event Type field) but each event is identified by a name. Different menu items on different menus can call the exact same event.

When the event is a ring or pull-down menu, the Event Called is the name of that menu. When the event is a function, it is the function name (a string of characters associated with the code for that event).

For example, a function key (the down arrow), a hidden menu item (the Next command on the Mainring menu), and the Next Document item (on the Brw_menu pull-down), all call the same "next_one" event that displays the next record on the form.

Event Type: There are three different types of events a menu can call. They can call a ring menu, a pull-down menu or a function. This field determines what kind of event the item being defined is. This one character field accepts the following:

- **R** (ring menu)
- **P** (pull-down menu)
- **F** (function)

The Add/Single Document menu item is an F for function, while the Add pull-down menu is defined as P for pull-down.

Hold After Select: This characteristic defines how the menu behaves after a user selects an item. Its meaning depends on the type of event involved. It is used to create browse-type rings that don't use the [ENTER] key internally, or hot pull-down menus that allow items to be selected without pressing [ENTER]. It can hold a pull-down menu open after an item on it has been selected and finished running.

This field accepts a **Y** or **N**. The effect of this characteristic depends on the event type being called by the menu item.

When the menu item calls a **ring menu event**, a Y or N determines the difference between a "main" ring menu and a "browse" ring menu. Y is used with main ring menus, and N is used with browse menus. In the main ring menu, the [ENTER] key can be used to select an item on the ring menu, and the user returns to the ring after selecting an item with an [ENTER]. In a "browse" menu, the [ENTER] key is used to select an item from the browse list and after doing so, the ring menu is put away. Hold After Select is defined by the event *calling* the ring menu, not by the items on the ring menu. It can also be defined when calling the `gen_menu()` function from within a program.

When the menu item calls a **pull-down menu event**, the Hold After Select characteristic indicates whether or not the menu is "hot," i.e., whether or not pressing the [ENTER] key is required to select an item from the menu. For pull-downs that don't hold after select (Hold After Select = "N"), typing the first character of any item on that menu causes the menu item to execute. This is a hot menu and the standard way pull-down menu events are called. For pull-downs that do hold after select (Hold After Select = "Y"), typing the character highlights the item on that menu, but you must press [ENTER] to select it. Once more, the Hold After Select characteristic is defined by the event *calling* the menu, not by the items on the menu. It too can be set within a program if the menu isn't called by another menu.

When a menu item calls a **function event** from a pull-down menu *only*, the Hold After Select characteristic determines what happens to the pull-down menu *after* the menu item has been selected and run. These menu items can either hold the pull-down after being selected from it (Hold After Select = "Y") or put away the pull-down after selecting a functional item (Hold After Select = "N"). The default is to put away the pull-down after selecting an item. You use the former when you are expecting another item to be selected from the same pull-down menu after executing a function. For more information on events refer to "Menu Function Events in Pull-Down Menus" on page 3-42.

Event Class: Determines the contents of the `menu_item` global variable passed to the application program. The event class allows programs to behave differently in different modes. For example, they might act one way when adding a new record but differently when updating a record. Since pull-down menus allow many different types of add and update commands, the Event Class allows existing programs to behave properly by setting the "menu_item" as a class rather than as a specific event. This field is only used with functional events.

For example, for browse events you would enter "browse," for add events you would enter "add."

Requires Detail Section: Determines whether or not this menu item is used with a header-only form. If no detail section is present, certain menu items are not used. For example, in `Old_ring`, the Tab menu item does not display if no detail section is present on the form. You can define generic menus that can be applied to either header-only or header/detail forms with the appropriate menu items selected at run time. The Requires Detail Section characteristic eliminates the need to create unique menus for every specific program.

The Requires Detail Section field accepts a Y, N, or S. An S allows you to control this option programmatically. Refer to page 3-21 for more information on controlling this option programmatically.

Menu Item Activation Characteristics

The next three items determine how menu items are turned off and on as the program runs. Deactivated menu items appear on the menu, but they are preceded by an "!" and cannot be selected by the user. They indicate that a menu item is currently unavailable, and that conditions in the program must change before it can be used. These items are:

Requires Rowid: Turns off and on (activates and deactivates) the menu item depending on the presence of a current row id. In programs generated with Fitrix *Screen*, a current row id is present whenever any document data is displayed on the current form. So, for example, if there was no current document, a command such as the Update command would be turned off because without a current document, there is nothing to update.

The Requires Rowid field accepts a Y, N, or S. An S allows you to control this option programmatically.

Requires Cursor Item: This characteristic turns off and on the menu item depending on whether or not there is a document position in the cursor or group of selected documents. Some of the group commands, such as group delete, start with the current document in the selected group of documents and continue to the end of that group. To have a group of selected documents, you must first use the Find command to select a group to work with.

This field accepts a Y, N, or S. An S allows you to control this option programmatically. Refer to page 3-21 for more information on controlling this option programmatically.

Requires Cursor Total: This characteristic turns off and on the menu item depending on whether or not there is a group of documents selected. Like the Cursor Item, this type of activation applies to group commands that act on a group of documents at one time.

This field accepts a Y, N, or S. An S allows you to control this option programmatically. Refer to page 3-21 for more information on controlling this option programmatically.

Menu Item Translation Characteristics

The last three characteristics of a menu item determine what languages are displayed. For each single menu item, there can be up to 100 different languages into which the item can be translated.

Language: This is a three-character code that indicates which language the related information is translated.

For example, you could use "ENG" for English.

Item Label: This field contains the text that is displayed on the menu. The text in this field should be in the language indicated in the Language field.

This field stores up to 20 characters.

Help Line: This field applies to ring menus only. This text field contains the help message that appears beneath the item in a ring menu. It describes the menu option.

For example, if you were defining an Add ring menu item, you would enter "Create new document(s)" into this field.

Questions About Creating New Menus and Menu Items

The following are answers to commonly asked questions about creating pull-down menus and pull-down menu items.

How do I define a ring menu?

There are seven steps to defining a ring menu:

1. **On the Menu Item Definition Form, enter a unique name for the menu.**

   By convention, ring menu names begin with a capital letter so they appear before pull-down menus and end with the word "ring."

2. **Enter the "0" in the Item Order ID field.**

   The first item on a ring menu MUST have an ID of "0." This item is the initial "label" for the menu that precedes the menu items themselves.

3. **Enter "D" in the Item Style field.**

4. **Enter the Language code.**

5. **Enter the prompt that appears on the ring menu in the Item Label field.**

   For example, the standard Mainring ring menu displays the word "Action:" before the menu items.

6. **Define the ring menu items.**

   Typically "Display" items are entered first. Then hidden items and finally function keys. These items can call other ring menus, pull-down menus, any of the "function events" built into the menuing system, or any other function event in the local `menu_extra` function defined. For more information on events, refer to "Menu Function Events in Pull-Down Menus" on page 3-42. The labels for menu items, whenever possible, should begin with unique letters. This is especially important when the ring is called with the "Hold After Select" characteristic set to "N" (browse-type menus).

7. **Define any other ring, pull-down menus, or `menu_extra` functions called by the ring menus.**

How do I define a Pull-Down Menu?

There are three steps:

1. **Enter a unique name for the menu.**

   By convention, pull-down menus begin with lowercase letters and end with the word "menu."

2. **Define the menu items that appear on the pull-down menu.**

Pull-down menus are limited to "Display" and "Function Key" types. These items can call other ring menus, pull-downs or function events. Most typically they call function events either in the built in library or in the local `menu_extra` function. A pull-down that calls another pull-down menu will be overlaid by the new menu. The labels for menu items, whenever possible, should begin with unique letters, especially when the "Hold After Select" characteristic is set to "N."

For more information on events, refer to "Menu Function Events in Pull-Down Menus" on page 3-42.

3. **Define any other ring, pull-down menus, or `menu_extra` functions called by the ring menus.**

How do I make a pull-down menu stay open so I can return to it instead of the prior ring menu?

Define the menu items on that menu with the Hold After Select characteristic set to "Y." On any given menu, some items can hold the menu, while others can automatically put it away. Typically, this feature is used when you expect the user to select another item from the same menu after selecting this item.

What is a "hot" menu?

A "hot" menu means that the menu item is executed immediately when you press the letter beginning the menu name (Item Label). Ring menus are *always* hot. Pull-down menus are set to hot when the "Hold After Select" characteristic is set to "N."

If a menu contains more than one item that begins with the same letter, can users select them by typing the beginning letter?

In ring menus, the answer is no. In pull-down menus, items can be selected (without execution) by typing the first character of the label if the "Hold After Select" characteristic is set to "Y." This highlights the first occurrence of a menu item that begins with that character. Pressing the same letter again selects the *next* menu item that begins with that letter.

Can I activate or deactivate menu items from within the 4GL program?

Yes. Before any menu is displayed, it checks to see if the menu items are active or not. "Normal" activation depends upon the "Y" or "N" and the Requires Rowid, Cursor Item, or Cursor Total characteristics. However, you can also enter "S" into any of these fields. If you do, the program calls the local function `menu_deactive` passing the menu event name, and the numbers 1, 2, or 3 depending upon which of the activation fields (1 = rowid, 2 = cursor item, 3 = cursor total) that the "S" appears in. This allows you to test each menu item against up to three local conditions. If the `menu_deactive` function returns a "true," the item is activated. If it returns a false, it deactivates the item.

Can I control whether or not a menu item is used from within a 4GL program?

Yes. Before any menu is loaded into the display array, it checks to see if the menu item should be used or not. "Normal" loading depends upon the "Y" or "N" in the Requires Detail characteristic. A "Y" value here means that the item only loads if a "detail" section is present. However, you can also enter "S" into the field for this characteristic. If you do, the program calls the local function `menu_chkput` passing to it the name of the menu, the menu type, the event name, and the screen type. If this function returns a "true," the item is put onto the menu.

Can I use the prog_ctl `ringput` and other ring functions in programs using Pull-Down Menus to control those menus?

Yes, but mostly No. Pull-Down Menus use an advanced version of the `prog_ctl` ring menu library. If you use the `scradv` library to compile your program, it must use the `scradv` library version of `lib_ring.4gl` to run Pull-Down Menus.

Currently these functions, such as `ringput`, `ringpos`, and `ringpick`, are named in the same manner as those original functions and work largely in the same way as documented. However, these libraries are evolving fairly rapidly and, because of their marriage with the database, using these functions directly will almost certainly result in ring menu emulsification. Also in the next release, these function are planned to be completely rewritten to speed their operation and conserve program size. At this point, new function names are used.

Can I use the prog_ctl `menuput` and other menu functions in programs using Pull-Down Menus?

Yes. The pull-down menus are based on this menuing system, but their functions have been completely rewritten and renamed to speed the menus and conserve memory size. The `lib_menu` and `menu_win` files in `prog_ctl` function are completely separate from the `lib_pull` `pull_win` file in `scradv`.

How do you set up pull-down menus so that the right and left arrow keys open the pull-down to the right or left of the current one?

A special function event called `movemenu` is contained in the Menu Control library. Simply set your right and left arrows as function key items on the menu and have them call the event "movement" with type "F" for function.

How do I define function keys to work a certain way within a menu?

Function keys are treated just like any other menu item. They, of course, require an Informix termcap that properly interprets the keys you want to use and this can be a problem with some special keys such as "Page Up" or "Insert," but the arrow, [ENTER], [ESC], and [DEL] keys are almost always supported.

The only differences between a Function key item and any other are the "Item Style" which is set to "F" for function keys, and the definition of the menu item's "translation" characteristics.

For function keys, the "language" is always set to "ALL." The "labels" are limited to the character strings returned by the menuing system into the `scr_funct` global variable. These strings include:

- left - for the left arrow
- right - for the right arrow
- up - for the up arrow
- down - for the down arrow
- page_up - for the page_up key

- page_down - for the page_down key
- accept - for the [ENTER] key
- cancel - for the "break" key
- escape - for the [ESC] key
- home - for the home key

Warnings about Function Keys:

[ESC] the "accept" key: You may encounter problems in attempting to redefine the "accept" key. Functions are checked before display and hidden items are processed. However, unless defined otherwise (specifically in the case of ring menus with Hold After Accept set to "N"), most menu selections also generate an "accept" value in `scr_funct`. If you have special function key "accept" processing, no other processing takes place.

[DEL] the "delete" key: In pull-down menus that are set to "hold," that is, where any menu item on them is defined with "Hold After Select" set to "Y," the only way you can "put away" these menus is to use the delete key. If you redefine "delete" within these menus, your user is unable to close them once opened.

How do I add new items to existing menus?

There are five steps:

1. **Call up the program to create new menu items.**
2. **Find all existing items on that menu.**
3. **For display-type items, decide in what position on that menus you want the new menu item to appear.**

   The easiest way is to add new menu items after existing ones, but sometimes this may not be desirable.

4. **If there is no "hole" in the existing numbering scheme in the position you want the menu item, you must renumber the "higher-ordered" menu items to make a hole.**

You do this by changing the items with a larger number in the Order/ID field first: changing the 10 to 11, the 9 to a 10, the 8 to a 9 and so on. You must do this in order not to create duplicate menu items. The system does not allow duplicates, even temporarily.

5. **Add your new menu item using an Order/ID number that positions the item where you want it.**

How do I create new function events for new menu items to call?

New function events (as opposed to events that call new ring or pull-down menus) are defined in the local function called `menu_extra`. This function contains events for any ring or pull-down menu. It doesn't matter if the event is connected with a displayed, hidden, or function key menu item. An empty function of this name is used by the menu libraries if you do not define such a function yourself locally.

Note

The event called is *not* the label that users see on the screen. The displayed label can change based on the language variable. The event name is an eight-character string defined along with the menu item.

This local "menu_extra" function should only contain a single CASE statement. This CASE statement tests for the event name in the global "scratch" variable. It then does the processing you need with that CASE statement or, more often, "points" to other functions within your program that do the processing. When a menu item is selected from either a ring or pull-down menu, the name of the event called by the menu item is placed in the scratch variable. If that event is not found in the internal event library of the menu program itself, it calls the "menu_extra" function. The "menu_extra" function should then continue testing the contents of the scratch variable and call the functions needed for those events.

For example, say you wanted to add menu items that called the events "special1" and "special2." You first go through the steps for adding menu items and, in the Event Name field, type "special1" or "special2." Be careful to pick unique, new event names that aren't used internally. In the Event Type field, enter an "F" to tell the system these were functions.

Then add a menu_extra function to your local code. This function looks like:

```
FUNCTION menu_extra()
CASE
   WHEN scratch = "special1"
       call function1()
   WHEN scratch = "special2"
       call function2()
END CASE
```

The functions referenced as "function1" and "function2" can be called anything you want. Any number of such functions or any other 4GL commands are invoked after the WHEN clause matching the event is called by the menu item. When users select one of your new menu items, they are passed to the right functions within your program by this CASE statement. For more information on events, refer to "Menu Function Events in Pull-Down Menus" on page 3-42.

What if my ring menu is wider than the data entry form on which the menu appears?

The displayed ring menu is truncated to fit in the width of your current window. If there are more menu items than can fit in a given window, an ellipse (...) appears to indicate more menu items are off the form. Users can move through the ring menus to see the other items using the [SPACE BAR] or right and left arrow keys. Menu items do not have to appear on the form in order to be selected by a single keystroke. If the user types the menu item keystroke, the command is executed and the part of the ring menu containing the last selected item is displayed.

What if my pull-down menu is longer than my current form?

Pull-down menus create their own overlapping window, so the size of the current form is irrelevant to their display.

Can I know from within other functions what the last menu item selected was?

Yes. This is the function of the "Event Class" characteristic of every menu item. This characteristic sets the value of the global variable `menu_item`. When the menu item is picked, the scratch variable is assigned the Event Name and at the same time, the `menu_item` variable is assigned the Event Class value. So each

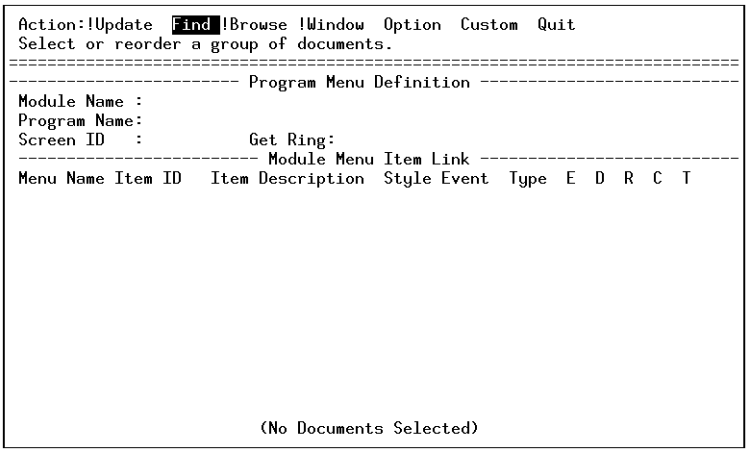
menu selection sets not one, but two different global variables that your programs can use. Of these two, however, the scratch variable is the most temporary since many different functions within the program affect its contents. However, the `menu_item` variable is expected to be changed *only* by the selection of another `menu_item`. Though obviously you can change it anytime you want within the flow of your program, the minute you do so, you lose your ability to know something about the last menu item called.

The Event Class usually is not unique to a menu item, but instead, defines a group or "class" of menu items that are functionally equivalent within the program. The most common use for this characteristic in pre-defined menus is to make new menu functions backwardly compatible with our original menuing system. In that system, when you selected the "Add" menu item, the `menu_item` variable was set to "add." However, in the new menus, there are many different commands that add new documents. To tell existing programs that all of these different functions are adding records, all of them set the "Menu Class" to "add."

Creating Custom Pull-Down Menus For Specific Programs

You can create custom versions of menus for specific programs. By selecting the Program Menu option on the Define pull-down, you can eliminate items that are not needed on a specific menu or change the way the menu functions. This displays the Program Menu Definition form. The Program Menu Definition program itself is both an example of Pull-Down Menus and a method by which you create them.

When you access this program, the following form is displayed.

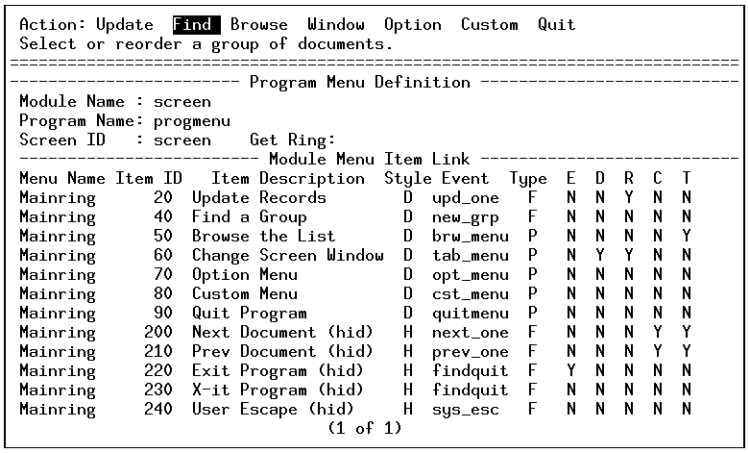


As you can see from looking at the menu line on the top of the screen, the ring menu here is very different than the standard Mainring menu. This ring menu is a variation created especially for this program because many of the options offered on the full Mainring are simply not appropriate to this menu.

Note

Whenever you customize *any* menu in a program, you must customize *all* menus in the program. This can be especially a problem with Browse menus, which aren't directly connected to the originating ring menu.

We can look at the menu items with which this special menu was created if we use the Find command to get the pre-defined menu items used by this menu. You can do this by entering "progmenu" on the Program Name line and "screen" on the Screen ID line. Pressing [ESC] displays the following screen.



```
Action: Update  Find  Browse  Window  Option  Custom  Quit
Select or reorder a group of documents.
================================================================================
---------------------- Program Menu Definition --------------------------
Module Name : screen
Program Name: progmenu
Screen ID   : screen    Get Ring:
------------------------ Module Menu Item Link ---------------------------
Menu Name Item ID   Item Description  Style Event  Type  E  D  R  C  T
Mainring      20  Update Records        D  upd_one   F   N  N  Y  N  N
Mainring      40  Find a Group          D  new_grp   F   N  N  N  N  N
Mainring      50  Browse the List       D  brw_menu  P   N  N  N  N  Y
Mainring      60  Change Screen Window  D  tab_menu  P   N  Y  Y  N  N
Mainring      70  Option Menu           D  opt_menu  P   N  N  N  N  N
Mainring      80  Custom Menu           D  cst_menu  P   N  N  N  N  N
Mainring      90  Quit Program          D  quitmenu  P   N  N  N  N  N
Mainring     200  Next Document (hid)   H  next_one  F   N  N  N  Y  Y
Mainring     210  Prev Document (hid)   H  prev_one  F   N  N  N  Y  Y
Mainring     220  Exit Program (hid)    H  findquit  F   Y  N  N  N  N
Mainring     230  X-it Program (hid)    H  findquit  F   N  N  N  N  N
Mainring     240  User Escape (hid)     H  sys_esc   F   N  N  N  N  N
                              (1 of 1)
```

Menus can be customized for specific programs simply by adding or deleting menu items found in the detail section of this form. Notice that only certain Mainring commands are displayed in the detail section for this form. The Add(10) and Delete (30) standard ring menu items from the Mainring menu have been eliminated for the progmenu program. To do this, delete the lines that contain those commands. These changes are made because there is not a need to add records in the header section of this particular program. The module, program, and screen information is created by the Form Painter when you define the screen form. The "Delete" command is also missing.

Note

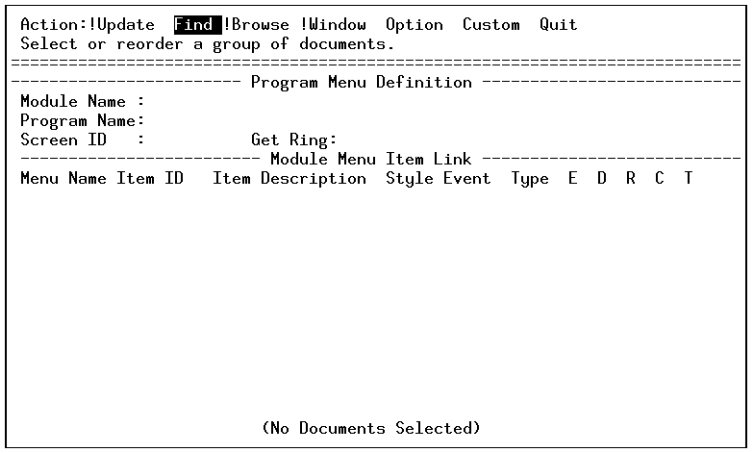
Make sure that you do not delete all of the commands in the ring menu. At least ONE ring menu command must be active when the program first loads.

The functions of certain commands have also been changed for this program. For example, the Update and Find commands normally call pull-down menus (`upd_menu` and `fnd_menu` respectively), but here their related events call functions. When the user selects "Update," they are automatically put into update on the current record. When they select "Find," they are put into a blank form for entering selection criteria. We have made these commands simpler for this particular form because the user is never working with large groups of documents. Group updates or re-sorts don't make any sense in this environment.

Other commands on the Program Menu Definition form have also been shortened. The items that begin with `brw_menu` and `cst_menu` have been changed from the standard menu. These are lines from the Browse Menu and custom pull-downs. For example, since there is no "Browse" screen for this program, there is no View List option. However, there is a Next and Previous Document option. Similarly, there is no "User Defined Fields" in the custom menu because it simply doesn't make any sense in this environment. Adding and Viewing Notes still remain because it does make sense to have these functions with this program.

The Program Menu Definition Form

The Program Menu Definition form allows you to customize your menus and menu items for each program in your application.



Module Name: This field contains the name of the module directory of the program whose menus you want to modify.

Program Name: This field contains the name of the program directory of the program whose menus you want to modify.

Screen ID: This field contains the name of the program whose menus you wish to modify. The screen ID is the name of the screen form without any filename extension.

Get Ring: This field contains the name of the menu you wish to modify.

The rest of the items on the Program Menu Definition form all deal with individual menu items. Each of these characteristics are defined on the Menu Item Definition form. They are displayed here so that you may change these characteristics on a program by program basis. For more detailed descriptions of these fields, refer to the Menu Item Definition form section.

Menu Name: This field contains the name of the menu that the menu item belongs to.

Item ID: This field contains the Item ID of the menu item.

Item Description: This field contains the description of the menu.

Style: This field contains the style of the menu item. A menu item can have the following styles: (D)isplay, (H)idden, (F)unction, (S)pecial.

Event: This field contains the action that takes place when the menu item is selected.

Type: The event can be a (R)ing menu, (P)ull-down, or (F)unction.

E: ([ENTER] key required) This field allows you to determine if you want the menu item to Hold After Select. Main ring menus generally have Y in this field, while Browse ring menus have an N.

D: (detail required) This is the Requires Detail field. Here you determine if your menu item can be used only if the screen form has a detail section. You can enter a Y, N, or S.

R: (rowid required) This is the Requires Rowid field. If you want the menu item to be deactivated when there is no current document enter a Y. You may enter Y, N, or S in this field.

C: (cursor item required) This is the Requires Cursor Item field. Enter a Y in this field if you want the menu item to be deactivated only when there is no current group of documents found by the Find command.

T: (total cursors required) This is the Requires Cursor Total field. If you want the menu item to be deactivated when there are no documents selected at all, enter a Y. N and S are also valid entries.

Defining A Custom Menu

There are only seven steps in creating a custom menu for a program. This process assumes that you have used the Form Painter to create screen forms in the program.

1. **Run the Program Menu Definition form.**

2. **Find the program who's menus need to be customized.**

   Typically, for main ring menus this is the main form that starts the program. For browse rings, it is the browse form. Execute Find, enter the module, program, and screen name to select that form.

3. **Update the Program Menu Definition form.**

   When you select the Update command from the command line, you are placed in the "Get Ring:" field. This field contains the name of the ring menu you want to customize for this program. You then press [ENTER]. When you press [TAB] or [ESC] to get down to the detail area, the detail section of the form will be filled in with the menu items defined for that ring menu.

Note

You only need to type in the ring name in the Get Ring field the *first* time you bring in a set of ring menu items. After that, you can update items by simply going onto the next step and pressing [TAB]. If you call in another ring menu, it writes over any other additions or changes you have made to this array.

4. **Press [TAB] to move down into the detail array.**

   At this point, the items for the ring menu you have specified in the Get Ring field and any dependent menus appear. You can now move through them to see all of the items as defined in the Menu Items Reference table.

5. **Delete or Insert new lines.**

   To eliminate any unwanted menu items for this particular program (module, program, screen ID), simply use the Informix Delete Line key (F2 in most Informix termcaps). You can also Insert lines, by pressing the Insert Line key (F1 in most Informix termcaps) to open a space in the array and use the Zoom command to look directly at the Menu Items Reference file and pick items from it to bring into the array. This is most useful when you delete an item you want or when you want to customize a menu not directly connected to the starting ring menu. You can bring in items from that menu into the current form, one item at a time.

6. **Change any menu characteristics.**

   Any characteristic can be changed except for the description, which always shows the original name of the menu item. However, you must make sure if you change the Menu Name and Order Items that there is an Item in the Menu Items Reference file for that specific Name and Order. If not, no language characteristics can be linked with your menu item. In general, you should only change characteristics to the right of the Description and use the Name and Order fields for Zooming. Most typically, change the event names and types called by menu items and perhaps their activation characteristics.

Note

You cannot change the order to reshuffle menu items. The Order is linked to the Menu Items Reference file and must point at the related item. It doesn't have to work in the same way, but it uses the language characteristics of that item.

7. **Change any "Special" menu items.**

   Items defined with an "S" do not appear when programs are run unless their style is changed to [D]isplay, [H]idden, or [F]unction key. These items exist in the Menu Items Reference file so you can change them here.

8. **Make sure all menus are referenced.**

   If the first ring menu is custom, all menus called in the program must appear in this custom detail table. If the menu isn't linked directly to the main ring, you can use Zoom to bring in its menu items one at a time, or, more efficiently, go to another screen in the program and call in the other ring menu. This last method is used for Browse screens. If you are using a Browse menu, you have a browse form, so select that form with the Find command and then type `Brw_menu` in the Get Ring field. This displays all the browse menu items.

   Menu items do not have to appear with the specific screen in which they are used. Any secondary screens (such as Add-Ons) can be used to create these secondary menu additions to the program.

Make sure you test all menus after defining them.

Defining a Custom Ring Menu

You can create your own unique ring menus to run within programs if you desire. However, we should note that the best programs are those that have consistently named menu items in a dependable order. It may be possible to create a new menu for every program, but it isn't necessarily a good idea.

To define a completely new ring menu, do the following:

1. **Define the ring menu items using the Menu Items Definition form.**

   Give your ring menu a unique, new name. You can use items that call functions on other ring menus and which use identical language labels. What makes a ring unique is the name you give it. What makes each item unique is the Order/ID number.

2. **Define any unique pull-down menus by defining the items on them.**

   A unique ring menu can call existing pull-downs.

   For example, you could create new ring that had its own "Custom" item on it which accesses the User Control functions by calling the "cst_menu" pull-down menu as its event. Only if the pull-down menu is new, would you have to create new items defined by a new name.

3. **After all items are defined, add the `put_scrlib` function in your `before_init` trigger.**

   The format would be:

   ```
   call put_scrlib("ringname","new_ring_name")
   ```

   where "new_ring_name" is the name you have assigned to your custom ring menu. This menu is then used to start the program instead of the default ring menu.

Linking a Custom Ring Menu (other than Mainring) into Your Program

If you create a custom ring menu, you need to take a couple of steps to link your menu into your program. If you just want to use the standard Mainring menu, then you do not have to do anything other than linking in the custom library.

To use any ring menu other than "Mainring," (the program default) you must add a `before_init` trigger that stores the name of the ring menu using the `put_scrlib` function and the scrlib key name `ringname`.

For example, if you want the program to use the ring menu "Oldring," which basically works just like the original ring menu generated by Fitrix *Screen*, add the following line to your `before_init` trigger.

```
call put_scrlib("ringname", "Oldring")
```

This stores the value "Oldring" under the key `ringname`. When the generated program runs, it uses the `get_scrlib` function to get the name of the ring menu to begin the program with.

Only if you don't store any value in "ringname" does the program use Mainring by default.

Note

If the ring menu specified in your call to the `put_scrlib` function hasn't been defined in the Menu Items Definition form, the program fails when it tries to open that form.

Calling a Ring Menu From Within a Program

You can also call different menus programmatically from within a program by using the `gen_menu()` function. This allows you to, for example, display a new screen from within a program and add its own unique ring menu to it. This function displays either ring menus or pull-down menus depending upon the `menu_type` flag.

To do this, all you have to do is put the call to the `gen_menu()` function in the trigger that occurs at the point you want the menu to appear. It is also a good idea to store the name of the menu you are calling using `put_scrlib("menuname",yourname)`, but this is not necessary for the menu to work. It is, however, helpful for people expecting to find the name of the current menu in that variable.

The syntax of the `gen_menu()` function is:

```
call gen_menu(ringname, enter_flag, menu_type).
```

ringname: a variable or string that contains the name of the ring menu you want to call.

enter_flag: is "Y" or "N" or a variable containing one of those values. This value is the same as you would use on the "Hold Upon Select" value in defining a call to a ring menu item. A ring menu that acts like the main ring menu is defined with a "Y." A ring menu that doesn't recognize the [ENTER] key like the Browse menu is defined with an "N" in this position. A pull-down menu with a "Y" waits for you to press [ENTER] to select an item. A pull-down with an "N" is hot.

menu_type: is "R" or "P" for a ring menu or a pull-down. This determines how the menu displays. There is no basic difference in the way menu items are constructed for these two different menus. You can, theoretically, display any menu as a ring menu or a pull-down depending on which menu_type flag you use. However, a ring menu does work better if you have a "O" item to show the "name" of the ring menu before the items you select and label "help" text that displays a help message under the ring menu item as it is displayed.

Example:

```
call gen_menu("Mainring","Y","R")
```

This call displays the default main ring menu at the point in the program it is called. The called menu returns control to your base program only when the new menu is exited.

Example:

```
call gen_menu("Brw_ring","N","R")
```

This call displays the browse menu. This makes sense if you first open a browse-type screen form.

Example:

```
call gen_menu("add_menu","N","P")
```

This displays the add_menu. The menu is "hot," that is, the items on it could be selected simply by pressing their first letter.

Note

Menu items that normally "hold" their menu so that they return to it after they are selected do *not* do so if called directly with the `gen_menu()` function. All menu items simply execute. Right and left arrows that usually bring up adjoining menus will also not work if they "pop-up" from a menu. They rely upon the context of a calling ring menu to tell them what "left" and "right" means.

Questions About Defining Program-Specific Menus

Can I delete menu items I don't need for a specific program?

Yes, this is one of the most common ways of modifying menus on an program-specific basis. To delete menu items, you follow four steps:

1. **Go to the Program Menu Definition form.**

   Use the Find command to find the screens in the program you are working with.

2. **Update the main screen and type the starting ring menu name into the Get Ring field.**

   This ring menu gets all the pull-down menus that it calls.

3. **Press TAB to edit the detail lines of the various menus.**

4. **Move to the menu item you wish to eliminate and press the Informix delete key. This is usually F2.**

Can I change the way a given menu item functions on a program-specific basis?

Yes. Any characteristic of a menu item except the menu item description and the language characteristics can be changed on a program by program basis.

The most common use for this feature is changing the events that a menu item calls. For example, if the various "update" options on the update pull-down menu aren't relevant, you can change the "Update" command on the Mainring menu to call the `upd_one` function directly instead of first calling the update pull-down.

To change a menu item function for a specific program, you follow four steps:

1. **Go to the Program Menu Definition form. Use the Find command to find the screens in the program you are working with.**

2. **Update the main screen, typing the starting ring menu name into the Get Menu field.**

   This ring menu gets all the pull-down menus that it calls.

3. **Press TAB to edit the detail lines of the various menus.**

4. **Move to the menu item you wish to change using the arrow key. Use the [ENTER] key to move to the field you wish to change.**

   If you are changing the Event Name, remember to also change the Event Type to match.

Can I create a new menu item for the local program alone?

Yes. This is done in two steps:

1. **First, you must define your new item in an existing menu using the Menu Items Definition form.**

   You do this the same way you add any new menu item, *but* instead of coding that Menu Style with an "D,""H," or "F" (display, hidden, function key), you code it with an "S" indicating a "special" menu item. Menu items coded with an "S" do not affect any program calling the generic menu. Using any other code when adding a menu item causes the new item to appear in all programs using the generic menu. For "S" style items, it is a good idea to reference the program it is for in your item description.

2. **Load the menu into the Program Menu Definition form and change the "S" to a usable style, that is, a "D,""H," or "F" depending on the type of menu item you want it to be.**

   If you are adding many new items or making radical changes, it may be easier to create an entirely new set of menu items for your program giving them new menu names. This can be done by copying any existing menu items. Using "S" style items only makes sense when only a few items are different.

Do I have to define all menu items in the generic file before making them program-specific?

Yes. The Menu Name and Menu Order/ID used in the program for defining program-specific menus *must* refer to an existing item in the general Menu Items table. This is necessary because only the Menu Items table has the language characteristics that are needed to allow all menu items to function. You can change any other menu characteristics while creating program-specific menus except the menu item description and these language characteristics.

Can I use two different versions of the same menu within a program?

Not in this release. Each menu can only be used in one version. If you need two different versions of the same menu, you first have to copy the existing version using a new menu name in Menu Item Definition program.

Troubleshooting Pull-Down Menus

When the menu item is selected it just blinks and doesn't do anything.

Pull-Down Menus are not finding the event called by the menu. This could be caused by a misspelling of the event name, or by mistyping an event.

Moving Pull-Down Menus to a New System

Assuming that you have developed your code and program on one system and are moving it to another, you need to understand the various components of the system and how they work together.

First, programs using Pull-Down Menus require the User Control Libraries. The User Control Libraries include the database structure and the code needed to drive different aspects of this system.

The Enhancement Toolkit includes the basic data for driving standard forms of menus such as Mainring, Brw_ring, Old_ring and the various pull-down menus, but if you create new menu items or program-specific versions of these menus, you must move the database descriptions of these items from your machine to the user's machine.

This is done fairly simply using INFORMIX-ISQL and the "load" and "unload" commands. The "unload" command saves an ASCII version of the menu data from the database into a small file. The "load" command can then load that information into the UCL table at the user's site. You would use the "Query-Language" option in ISQL and select the database before entering the following commands.

The commands for saving the complete "Menu Items" file from your development system is:

```
unload to "cgmcmndr.unl" select * from cgmcmndr
unload to "cgmcmndd.unl" select * from cgmcmndd
```

The command for saving any program-specific menus are:

```
unload to "cgmmenud.unl" select * from cgmmenud
```

These commands create three files called `cgmcmndr.unl`, `cgmcmndd.unl`, and `cgmmenud.unl`, which are then moved to the user's machine.

To load these files on the user's machine you first have to delete any existing data in the local user's version of these files. To do this, enter:

```
delete from cgmcmndr where 1 = 1
delete from cgmcmndd where 1 = 1
delete from cgmmenud where 1 = 1
```

Then load the new data for these files using the following:

```
load from cgmcmndr.unl; insert into cgmcmdnr
load from cgmcmndd.unl; insert into cgmcmndd
load from cgmmenud.unl; insert into cgmmenud
```

This completes the loading of the new files into the user's machine.

If you want, unload just part of the data from the file, for example, unload just a specific program's program menu from `cgmmenud` using the following:

```
unload to "cgmmenud.unl" select * from cgmmenud where prog = "my_program"
```

Here, "my_program" is the name of the program for which you have defined a variation on an existing ring menu. You then don't have to delete any information in the user's system. All you have to do is load the new information using:

```
load from cgmmenud.unl insert into cgmmenud
```

Note

Be sure that anytime you define new commands, you unload both the `cgmcmndr` table *and* the `cgmcmndd` table. The first table contains the definition of the menu item itself. The second contains the "language" characteristics of the item. Both are required for a new menu item to work.

Menu Function Events in Pull-Down Menus

The following "events" are encapsulated in the menuing system itself. The names of these events are what is entered into the "Event Name" field when defining a menu item for the Event Type "F," for "function" (as opposed to an event which is a menu of one kind or another). Any function event not appearing in this list but appearing in a menu item must be defined in the local function `menu_extra()`.

General Ring Events

movemenu: This event is used only from an open pull-down menu. It closes the current pull-down menu and moves to the ring command next to it. If that ring command calls a pull-down, then that pull-down displays. This event is typically used only for right and left arrow keys.

findquit: This event finds the "quit" or "exit" item on the current ring menu and calls it. It is used primarily by "break" function keys. This event works by finding the menu item that calls the "quitmenu" or "prg_quit" events.

findwind: This event finds the "window" command on the ring menu. It is used primarily by the "tab" function key.

Mainring Events

addone: This event calls the functions that allow the addition of a single document on the current form. This is the equivalent of the original menus "Add" command. In Mainring, it is called by the "Add/Single Document" option.

addmany: This event calls the functions that allow the addition of many documents on the current form. This is the equivalent of the original menus "Add" command called over and over again. In Mainring, it is called by the "Add/Many Documents" option.

copyone: This event makes a copy of the current document and puts you into the update mode on that document. It requires a "current rowid" to function. Used by the "Add/Document Copy" option in Mainring.

copymany: This event makes a copy of the current document and puts the user in update mode. When the user finishes update, a copy of that most recent version of the document is created and the user is put into update mode on that. In Mainring, it is called by the "Add/Similar Documents" option. This event requires a current row id in order to function.

upd_one: This event updates the current document on the screen. It requires a current row id. This event is called in Mainring by "Update/This Document" option.

upd_all: This puts all currently selected documents into update mode. It is called by the "Update/All Documents" option in Mainring.

del_one: This event deletes the current document. It is called by the "Delete/This Document" in Mainring.

del_all: This event goes to each currently selected document and asks if you want to delete it. This function is called by the "Delete/All Documents" option in Mainring.

new_grp: This event takes the user into a search criteria form (Query-By-Example). It calls the functions called by the original "Find" command.

sort_grp: This event allows the user to use the form to specify which data element to sort on. It is called by the "Find/Sort Group" option in Mainring.

brw_list: This event displays the browse window. It is the equivalent of the original "Browse" command.

next_one: This event displays the next document in the current group. It is the equivalent of the original "Next" command. It is now called by several different commands, including the down arrow function key from Mainring.

prev_one: This event displays the previous document in the current group. It is the equivalent of the original "Prev" command. It is now called by several different commands, including the up arrow function key from Mainring.

view_det: This event moves the cursor to the detail section of the current document and allows the user to scroll through it without being in the update mode. It is the equivalent of the old "Tab" command.

spec_cmd: This event calls the Option menu and displays it. It is the equivalent of the original "Option" command.

oth_prgm: This event calls the user defined escapes. It is called by the "Options/Other Programs" option from Mainring. Previously this function was available only as a User Control function key.

todolist: This event calls the user-definable to-do list. It is called by the "Options/To Do List" option from Mainring. Previously this function was available only as a User Control function key.

prg_info: This event calls information about the program. It is called by the "Options/Information" option from Mainring. Previously this function was available only as a User Control function key.

prg_ack: This event calls the program acknowledgement screen. It is called by the "Options/Acknowledgements" option from Mainring. Previously this function was available only as a User Control function key.

sys_esc: This event calls the operating system prompt. It is called by the "Options/Escape to System" option from Mainring. Previously this function was available only as a User Control function key.

prg_hlp: This event calls the program help. It is called by the "Options/Help" option from Mainring. Previously this function was available only as a User Control function key.

add_flds: This event calls the user-definable fields function. It is called by "Custom/Additional Fields" option in Mainring. It was previously available only as a User Control function key.

see_flds This event calls the viewing of user-definable fields function. It is called by "Custom/Add. Fields (View)" option in Mainring. It was previously available only as a User Control function key.

edt_note: This event calls the user-definable notes function. It is called by "Custom/Document Notes" option in Mainring. It was previously available only as a User Control function key.

see_note: This event calls the viewing of user-definable notes function. It is called by "Custom/Doc. Notes (view)" option in Mainring. It was previously available only as a User Control function key.

hot_keys: This event calls user-definable function keys. It is called by "Custom/Hot Keys" option in Mainring. It was previously available only as a User Control function key.

chg_func: This event calls the user-definable functions. It is called by "Custom/Change Functions" option in Mainring. It was previously available only as a User Control function key.

list_err: This event calls the error list for the current program. It is called by "Custom/List Errors" option in Mainring. It was previously available only as a User Control function key.

req_feat: This event calls the feature request form. It is called by "Custom/Request Feature" option in Mainring. It was previously available only as a User Control function key.

prg_stat: This event calls the program status form. It is called by "Custom/Program Status" option in Mainring. It was previously available only as a User Control function key.

prg_quit: This event exits from the current program. It is called by "Quit/Exit Program" option in Mainring.

4

Program Control Library

This chapter describes the functions available in the Program Control Library. The Program Control Library is included with the purchase of the Enhancement Toolkit. The Program Control Library contains a variety of useful functions designed to give you even more flexibility when creating programs. The Program Control Library consist of the following features.

- Dynamic menus
- Dynamic ring menus
- Scrolling input fields
- Warning windows
- Application's C Library

Overview of the Program Control Library

The Program Control Library contains function sets that aid you in building a friendly user interface. In general, these functions extend the features provided by the INFORMIX-4GL programming language. Usually the interface style and functionality presented by these tools requires calls to the C library, which contains a set of C language functions designed to allow for ready interaction with UNIX from an Informix program.

To utilize the Program Control Library, you need to add the library to your list of libraries in the Makefile of the program that calls any of these functions. The `$fglibdir/lib/prog_ctl.a` library should come just before `$fglibdir/lib/standard.a` in the list of libraries in the Makefile.

The *CASE* Tools Enhancement Toolkit needs to be present on any system that uses these Program Control Library functions. This means that if you incorporate these functions in your applications your customers must have *CASE* Tools Enhancement Toolkit or the Code Generator on their system.

The function sets in the Program Control Library include:

Dynamic Menus: This function set is contained in two source files: `lib_menu.4gl` and `menu_win.4gl`. Dynamic menus gives you the ability to build menus on the fly without knowing in advance how big the menu window should be or how many options are on the menu. The menu is automatically sized for optimal display size and page distribution. You can provide guidelines for the sizing/display optimizer to follow to size menus according to known parameters (these are followed unless they are unrealistic for the size and number of the menu items). You can optionally allow the user to select several menu items at once or only a single menu item. There are additional options that allow you to supply menu header lines, Zoom capability from the menu, stacking menus, plus more. The dynamic menus function set is designed much like the `textedit()` function set in the standard Fitrix *Screen* Informix library. It can be used as a replacement for "display array" in many instances.

Dynamic Ring Menus: This function set is contained in `lib_ring.4gl`. The dynamic ring menus function set is designed just like the dynamic menu function set but provides a similar capability for building dynamic, paged ring menus. This function set provides a specialized subset of the dynamic menu set. The dynamic ring menus can be used to replace the informix "menu" instruction.

Scrolling Input Fields: This function is contained in the 4GL source file `fg_getfld.4gl`. It provides a function that allows you to collect input in a "reverse" attribute field at any place in the current window. The primary function allows for input of data beyond the size of the displayed field length by scrolling the field automatically as the user types in data. You must specify the current window position, the relative position of the input field, the size of field, the maximum length of the data that can be entered, and any beginning value for that data. You get back the new data and a special code indicating what key was used to exit the scrolling field.

Warning Windows: This function set allows you to pop up a dialog box with a short message of your design sized and centered in the screen. There are two interface styles: one provides the user with a YES/NO/CANCEL response option, the other is a simple OK prompt for any key to continue. This function set is designed much like the dynamic menu set in the way it is invoked.

Dynamic Menus

This suite of menu routines is used to build, display, and select from menus that can be dynamically maintained. Typically the dynamic menu is built with a series of calls to `menuput()`, which adds a menu item to the menu for each call. Next `menupick()` is called returning the selected item in `scratch` and the index of the item selected (0 if [DEL] was pressed or an error was encountered). The menu automatically closes once a selection is made unless you make a call to `menuhold()` before calling `menupick()`. If you choose not to automatically put the menu away you must do so manually by calling the function `menuclose()`.

Once a menu is open (displayed), any successive calls to any menu function other than `menupick(), menusget(), menuget(), menunext(),` or `menuactive()` start on the next available menu (up to 10 menus can be open at a time).

An alternate way of building a menu is by letting `scratch` equal an SQL statement and calling `menusel()`, which makes the database selection, and then calling `menuput()` for each row found.

You can allow Zooms from the menu in which case the current item is returned in `scratch`, and `scr_funct` is set to "zoom". If Zoom is set for a menu you must check for `scr_funct = "zoom"` on a successful return to identify a Zoom selection from others. Usually a Zoom menu should also be a "hold" menu. Be sure to call `menuhold()`.

You can retrieve a list of all the menu items in a menu with the function `menuget()`, which starts by returning menu item one in `scratch` and then returns each menu item for each call. You may only get items from a "held" menu since `menuget()` only works on the current menu.

You can define the help text that should be made available from the menu by setting the help module, program, and number explicitly with a call to `menuhelp()`. If you do not call menuhelp then you must process help locally. When [CTRL]-[w] is pressed the `menupick()` function for the menu returns with the current menu item number and the value of `scr_funct` is "help." The menu window is not closed. If you wish to use the default help for a menu, call `menuhelp()` with a null module argument.

By calling the function `menumany()`, you mark the menu as a menu from which multiple items may be selected before exiting. Pressing [ENTER] selects and unselects items. Once done [ESC] brings up a simple "Done Selecting?" prompt. "Yes" exits the menu with the current selections while "No" and "Cancel" allow you to continue selecting.

To retrieve a list of selected items you may call `menusget()`, which returns each item in succession that was selected (in `scratch`) and returns the menu item number as long as there was another item to return. Once there are no selected items left to return it returns zero. The second return value reflects the order in which the menu items were selected.

Menu items can be activated and deactivated in order to allow or prevent selection of the menu item. The default is active. You call the function `menuactive()`, which requires an argument to specify the item to activate/deactivate and an argument to specify to activate or deactivate. By passing a null item argument the last item put onto the menu is operated on.

To build the header strings for the menu, you call the function `menuhead()`. The `menuhead()` function requires three arguments. The first is the header string. The second is how it must be positioned on the menu. That is, either centered ("center"), right justified ("right"), left justified ("left" - the default), or as a pattern ("pattern"). A pattern type heading takes the first character of the heading and repeats it for the width of the menu. The last argument is the heading attribute, which can be "high" for reverse, "dim" for blue, or "normal" for white. (A call to `menupick()` with a non-null header string automatically calls `menuhead()` for the header plus a dashed line.)

You can specify the default row/column position (over the built in default) by calling `menupos()`. If you need to be sure the window is at least a certain width you can also specify the `min_width` for the window with the `menupos()` call.

For special pull-down type menus you can call `menuwrap()` to turn left and right wrap off. In this case `scr_funct` is set to "right" or "left" and the menu returns with no item selected.

You can make a menu window current by calling `menucurrent()`. Also, any call to `menupick()` always makes the menu window current in order to allow the controlling application to switch between application windows without problem.

Each menu is divided into columns and sized for optimal display.

In order to make a menu item the current cursor position on the menu before entering the menu you can use a call to `menunext()` specifying the menu item that should be active.

To only display the first page of a menu without prompting for selection you can call `menuview()`. You must use `menuclose()` to close this menu.

Function Notes

- The maximum size menu item is 40 characters.
- The maximum number of menu items is 500.
- The maximum number of nested menus is 10.
- The maximum number of heading lines per menu is 10.

Summary of Dynamic Menu Functions

`menuput()` — add a menu item to the menu

Arguments: `menu_item` — text to appear for the menu item.

Returns: none.

Notes: If the menu item is NULL, an item line of dashes is automatically put in place of the text. The dashed line is not selectable as a menu item. If the menu item is SPACES, a blank line is used as a non-selectable menu item. If the menu item is "(see `scratch`)" then the contents of `scratch` are used for the menu item. A new menu is opened if the current one is already open.

`menuhead(`*`"header_str", "type", "attribute"`*`)` — add a header line for the menu.

Arguments: `header_str` — header text string. `type` — left, right, center, pattern. `attribute` — high, dim, normal.

Returns: none.

Notes: If the menu item is "(see `scratch`)," then the contents of `scratch` are used for the menu item. A new menu is opened if the current one is already open.

`menusel()` — add items to the menu from an SQL query.

Arguments: none. Expects an SQL query in `scratch`.

Returns: `num_items` — number of items put onto the menu.

Notes: `menusel()` expects an SQL query that is prepared and opened as a cursor putting each element into the menu and returning the number of elements put into the menu. A new menu is opened if the current one is already open.

`menupick(["`*`header_str`*`"])` — select an item from the menu.

Arguments: `header_str` — optional header text.

Returns: `item_num` — the item number selected plus the item selected in `scratch`.

Notes: If `header_str` is NULL then no automatic header is used (already set with `menuhead()`, or no header desired). If no header lines have been defined, and a header string is supplied to `menupick()` then a default header is built automatically with one or two lines to display ESC/DEL/ENTER function messages followed by a line of "=" characters. If the header line is supplied, it is added to the existing header (default of programmer defined), centered, and followed by a single dashed line. The selected menu item number is returned (0 if no selection) and the menu item text for that item is returned in `scratch`. This function does not start a new menu.

`menuhold()` — prevent the menu from closing once a selection is made.

Arguments: none

Returns: none

Notes: Only hold the menu if a selection has been made. Pressing [DEL] to exit without selection always closes the menu. A new menu is opened if the current one is already open.

`menuclose()` — close an open menu.

Arguments: none

Returns: none

Notes: Closes whatever is the current menu and its window.

`menuhelp(`*`help_module, help_program, help_number`*`)` — set the help module, program, and number for the menu

Arguments: `help_module` — help text "module" key to use for the menu. `help_program` — help text "program" key to use. `help_number` — help text "number" key to use.

Returns: none

Notes: The module, program, and number are the keys for context sensitive help ([CTRL]-[w]) from the current menu. It starts a new menu if the current one is already open. The menu returns for local help processing if `menu-help()` is not called. A null module argument causes the default help for the menu to be used.

`menuget()` — get the list of items on the menu.

Arguments: none

Returns: true/false — true if next element is in `scratch`.

Notes: Starting at one returns the next menu item in `scratch` until all elements have been returned. The true/false return value indicates whether it is done returning elements. The counter used by `menuget()` is re-initialized to one upon a return from `menupick()`. This function does not start a new menu.

`menusget()` — get the list of selected items on the menu.

Arguments: none

Returns: `item_number` — the menu item returned (0 if none). `order_number` — the relative order number of the element.

Notes: Starting at one returns the next selected menu item in `scratch` until all selected elements have been returned. The first returned value gives the menu item returned (0 if done), the second return value returns the relative order that the item was selected. The selected items are re-marked as unselected once they have been returned by `menusget()`. The counter used by `menusget()` is re-initialized to 1 upon a return from `menupick()`. This function does not start a new menu.

`menuzoom()` — set a flag to allow Zooms from this menu.

Arguments: none

Returns: none

Notes: Starts a new menu if the current one is already open.

`menupos(`*`start_row, start_col, menu_width`*`)` — set the default window coordinates.

Arguments: `start_row` — the desired upper row position of the menu. `start_col` — the desired left column position. `menu_width` — the desired menu width.

Returns: none

Notes: The starting row, column, and width are only used if a menu with those dimensions is possible given the number of items on the menu. A new menu is opened if the current one is already open.

`menumany()` — set the menu to allow for selection of multiple items.

Arguments: none

Returns: none

Notes: A new menu is opened if the current one is already open.

`menuactive([`*`menu_item`*`], `*`act_level`*`)` — activate or deactivate a menu item.

Arguments: `menu_item` — optional menu item to activate (if null activate current item.) `act_level` — activation level (-1 = non-active, -2 = non-item, 0 = active, > 0 = selected order).

Returns: none

Notes: If the menu item argument is null then the activation status is applied to the current menu item (for instance, immediately following a call to `menu-put()`).

Activation status: -2 never selectable, -1 not currently selectable, 0 currently selectable, >0 selected (value is relative selection order). This option does not start a new menu.

`menuwrap()` — turn on or off the left/right movement.

Arguments: none

Returns: none

Notes: Left and right movement does not wrap, instead it returns with `scr_funct` set to "left" or "right." It starts a new menu if the current one is already open.

`menucurrent()` — make the menu window current.

Arguments: none

Returns: none

Notes: Makes the current menu window the active window (can be used to redisplay the current menu if covered by some other window). This function does not start a new menu.

`menunext(`*`item_num`*`)` — preset the current menu item.

Arguments: `item_num` — menu item number to make active.

Returns: none

Notes: Automatically pages the menu if the requested menu item is not on the current page. This function does not start a new menu.

`menuview()` — You must use `menuclose()` to close this menu when done. A new menu is opened if the current one is already open.

Dynamic Ring Menus

This suite of ring menu routines is used to build, display, and select from ring menus that can be dynamically maintained. Typically, the ring menu is built with a series of calls to `ringput()`, which adds a ring menu command to the ring menu for each call. Next `ringpick()` is called to display the ring menu and prompt for the command selection returning the selected ring menu command in `scratch` and the index of the command selected (0 if [DEL] was pressed or an error was encountered). `ringpick()` is called each time the program prompts for a command selection from the ring menu. Up and down arrow keys are also returned from the ring menu to allow for flow control.

Basically the dynamic ring menu function set is designed to imitate the dynamic menu function set described above. Most of the function calls are structured the same, and the general methods for building and invoking the menus are the same. The ring menus are more limited in size and scope being a specialized horizontal type of menu. At most, 20 ring menu commands can be used.

Once a ring menu is opened (displayed), any successive calls to any menu function other than `ringpick()`, `ringnext()`, or `ringcurrent()` start on the next available ring menu (up to 10 ring menus can be open at a time).

You can define the help text that should be made available from the ring menu by setting the help module, program, and number explicitly with a call to `ringhelp()`.

You can specify the default row/column position (over the built in default) by calling `ringpos()`. If you need to be sure the window is a certain width you can also specify the width for the window with the `ringpos()` call.

You can make a menu window current by calling `ringcurrent()`. Also, any call to `ringpick()` always makes the menu window current in order to allow the controlling application to switch between application windows without problem.

In order to make a ring menu command the current cursor position on the ring menu before entering the menu, you can use a call to `ringnext()` specifying the ring menu command that should be active. The menu is paged as needed to display the requested command.

If all ring menu commands cannot be displayed on the screen at once, then they are paged right and left with ellipses to indicate that the menu extends in that direction as appropriate.

The ring menu always compresses the space between ring menu items to make them fit on a single menu "page." It leaves as much as three spaces (the default) between items if all can fit and adjusts down to one space in an attempt to make the items all fit. In cases when all items fit on a single menu page, you can still reduce the spacing between items to less than three. To do this call `ringspace()` with the desired spacing. Your requested spacing always gets overridden to make the menu items fit.

Summary of functions

Function Notes:

- The maximum size ring menu command is 20 characters.
- The maximum number of ring menu commands items is 20.
- The maximum number of nested ring menus is 10.

`ringput(`*`ring_item, ring_message`*`)` — add a command to the ring menu

> **Arguments:** `ring_item`—command name. `ring_message`—action message to appear on prompt line.
>
> **Returns:** none
>
> **Notes:** If the ring menu command item is "(see `scratch`)" then the contents of `scratch` are used for the command. If the action message is "(see `scratch`)" then the contents of `scratch` are used for the action message. This function starts a new ring menu if the current one is open.

`ringnext(`*`ring_item`*`)` — preset the current active ring menu command.

Arguments: `ring_item`—ring menu command number to make active.

Returns: none

Notes: Automatically pages the menu if the requested menu item is not on the current page. This does not start a new menu.

`ringpick([`*`ring_name`*`])` — activate the ring menu for command selection.

Arguments: `ring_name`—optional ring menu name.

Returns: `cmd_number`—the ring menu command number selected plus the command selected in `scratch`.

Notes: If `ring_name` is NULL then no ring menu title is used. The selected ring menu command number is returned (0 if no selection) and the command name is returned in `scratch`. This does not start a new menu.

`ringhelp(`*`help_module`*`,` *`help_program`*`,` *`help_number`*`)` — key the help text for the ring menu.

Arguments: `help_module`—help text "module" key to use for the menu. `help_program`—help text "program" key to use. `help_number`—help text "number" key to use.

Returns: none

Notes: The module, program, and number are the keys for context-sensitive help ([CTRL]-[w]) from the current menu. This starts a new menu if the current one is already open.

`ringclose()` — close the current ring menu.

Arguments: none

Returns: none

Notes: Closes whatever is the current menu and its window.

`ringpos(`*`start_row, start_col, menu_width`*`)` — position and size the ring menu (defaults).

Arguments: `start_row`—the desired upper row position of the menu. `start_col`—the desired left column position. `menu_width`—the desired menu width.

Returns: none

Notes: The starting row, column, and width are only used if a ring menu with those dimensions is possible given the number of items on the menu, the length of the ring menu commands, and the ring menu name (usually no problem). This starts a new menu if the current one is already open.

`ringcurrent()` — make the ring menu window current.

Arguments: none

Returns: none

Notes: `ringcurrent` makes the current ring menu window the active window (can be used to redisplay the current menu if covered by some other window). This function does not start a new menu.

`ringspace(`*`space_cnt`*`)` — sets a default spacing for the ring menu items.

Arguments: `space_cnt`—desired spaces between items (1, 2, or 3).

Returns: none

Notes: Requests a default spacing for the ring menu items. This function does not start a new menu.

Scrolling Input Fields

The scrolling field function is used to allow for data entry in a display space smaller than the length of data that can be entered. For example, if you have a small screen where there is only room for a data input field of 20 characters but where the data can be 30 characters, you can use the scrolling field to allow the user to view and update all 30 characters of data. The largest data field that can be input is 250 characters.

To run the function you can pass it an initial string value for the field (optional). You must specify the row and column starting position of the field relative to the window in which the scrolling field is used. You must also specify the position of the window on the terminal screen (the coordinates used with the `open_window` instruction). Although one set of coordinates would be possible, it makes the program less maintainable if your window size and position changes as you enhance and change your program. You also must specify the size of the scrolling field display space and the length of the character string to be entered.

When called, the scrolling field editor takes over until an exit action is taken. The field is displayed in reverse attribute. Within the scrolling field editor, you can move left and right with the arrow keys without erasing data. Pressing [SPACE] and [BACKSPACE] also move you right and left but erase data as you go. When you pass either end of the field with movement keys, you automatically leave the field (the function returns). While in the field you can delete to the end of the line by pressing [CTRL]-[d], you can insert a blank character at the current cursor location by pressing [CTRL]-[a], you can delete the current character with [CTRL]-[x], and you can move forward or backward a word at a time by pressing [TAB] or [BACKTAB]. [HOME] and [END] keys (when setup correctly in your termcap) take you to the beginning of the field and end of the text respectively. All other keys cause you to exit the scrolling field (function returns).

When the function returns, it returns the code number of the key stroke pressed when the field was exited. It also returns the new data in the field. If you exited the field with [DEL] to cancel, the field contents are restored to the initial data passed to the function.

In general the scrolling field is used by calling it in before field logic in an "input" statement. You must use a dummy field rather than the real data for the "input from" since the input logic truncates the data to the display size.

We recommend displaying the real data in the dummy field used in the input statement truncated with ellipses if it all won’t fit in the display space.

Since only before field logic is being used, the normal comment line at the bottom of the screen does not appear so you manually display the comment string before the call to `fg_getfield()`. You should then redisplay null after the call to clean up the comment line.

The after field logic for a scrolling field should be placed after the `fg_getfield()` call in the before field section. Scrolling fields are skipped with a next field statement after the before field, `fg_getfield()` call, and after field logic is run (You don't want to enter the dummy field after exiting the scrolling field).

Function Notes:

- Attribute reverse is always used for the field.
- The maximum length of the input data is 250.

Scrolling Fields in Input Arrays

Scrolling fields can be used with "input array." However, up and down arrows cannot take you to the previous or next row (cannot readily move up and down between rows with programmatic logic in before field).

An example of a scrolling field in an input array can be found on page 4-29.

Summary of functions

`fg_getfield(`*`fld_buffer, row_pos, col_pos, row_offset, col_offset, fld_length, data_length`*`)` — scrolling field input function.

Arguments: `fld_buffer`—initial contents of the field.

`row_pos`—row position relative to the window.

`col_pos`—column position relative to the window.

`row_offset`—window's row position.

`col_offset`—window's column position.

`fld_length`—display length of the field.

`data_length`—length of data to input into the field.

Returns: `ret_code`—return key stroke code. `fld_buffer`—contents of the field upon exit.

Notes: All function keys and control keys exit except [TAB], [BACKTAB], insert character, delete character, delete to end of line, delete line, home, and end. Special return values include:

Value	Key
13	[ENTER] (^M)
135	[ESC]
136	[DEL]
137	up arrow
138	down arrow
139	left arrow
140	right arrow
<=26	corresponding control key [^A - ^Z]
>=101	corresponding function key [F1 - F33]

Only [DEL] cancels changes to the field. If row or column coordinates are illegal, function terminates with a fatal error.

Warning Windows

The set of functions used for dynamic warning and error message boxes are structured very much like the dynamic menus and dynamic ring menus. Usually you call `warnput()` for each line of warning text you want to display in the window then call `warnhelp()` to key the context sensitive help for the warning window. Finally you call either `warnbox()` for a simple confirmation type warning window or call `warnyn()` for a warning window which requires a YES/NO/CANCEL choice.

You can store the warning messages in `stxerord` and automatically load them by using `warnread()` or `warnrd()` instead of manually building the warning window text with `warnput()`.

Warning windows cannot be nested. Each call to either `warnbox()` or `warnyn()` resets the warning window text arrays to be reloaded for the next set of messages.

These warning message windows can be used in programs that may need to be run in a non-interactive mode such as scheduled reports. If the global `auto_answer` is set with a call to `put_scrlib()`, then its value is used as the default response to the warning prompts.

Summary of functions

Function Notes:

- Maximum 10 lines of 60 characters of warning text.
- No nested warning windows.

`warnput(`*`warn_text`*`)` — put a line of text into the warning message box.

Arguments: `warn_text`—warning text line.

Returns: none

Notes: If the warning text is "(see `scratch`)" then the contents of `scratch` are used for the text.

`warnread(`*`err_module, err_program, err_number`*`)` — read warning text from error table (`stxerord`).

Arguments: `err_module`—error module key for warning text. `err_program`—error program key for warning text. `err_number`—error number key for warning text.

Returns: true/false true if some text found, otherwise false.

Notes: Prepares a selection cursor on `stxerord` using the keys and uses `warnput()` to add the text to the warning box. Returns true if at least one text line was read.

`warnrd(`*`err_number`*`)` — same as `warnread()` only doesn't need module or program.

Arguments: `err_number`—error number key for warning text.

Returns: true/false true if some text found, otherwise false.

Notes: uses progid to determine `err_module` and `err_program` and calls `warnread()`.

`warnhelp(`*`help_module, help_program, help_number`*`)` — specify the help key for the warning box (^W).

Arguments: `help_module`—help module key for warning text. `help_program`—help program key for warning text. `help_number`—help number key for warning text.

Returns: none

Notes: Module, program, and number determine key for context sensitive help from the warning box (^W).

`warnbox()` — run `warnbox` with "OK" as the only menu option.

Arguments: none

Returns: none

Notes: Any key returns — use for warning message display only uses `auto_answer` for automatic response for use with non-interactive programs (reports).

`warnyn()` — run warnbox with YES/NO/CANCEL as the menu options.

Arguments: none

Returns: true/false, YES(true), NO/CANCEL(false) (int_flag set if CANCEL).

Notes: Returns true or false — true if YES selected, false if NO or CANCEL was selected. `int_flag` is set if CANCEL is selected. [DEL] is the same as CANCEL. This function uses `auto_answer` for automatic response for use with non-interactive programs (reports).

Examples

The examples below are simplified versions of actual code and use variables not defined in the provided code and have extra logic removed to improve readability. The examples show all of the logical flow required to run and maintain the various interface items.

Dynamic Ring Menu

This example builds a simple ring menu with four options, it presets the active item before each call to `ringpick()` (in the actual code the menu item often changes during the processing of a ring menu command selection).

```
# Build the ring menu
call ringput(str.report_cmd, str.rpt_mssg)
call ringput(str.define_cmd, str.dfn_mssg)
call ringput(str.help_cmd, str.hlp_mssg)
call ringput(str.quit_cmd, str.qut_mssg)
# Set menu position and help context
call ringhelp("report", "mainmenu", 1)
call ringpos(2, 3, 76)
let cur_item = 1
# Menu loop
while true
    # Make sure the current item is set
    call ringnext(cur_item)
    # Call the ring menu
    if ringpick(str.action) then end if
    # Process the chosen item
    case
      when scratch = str.report_cmd
        let cur_item = 1
        if not report_menu() then exit while end if
      when scratch = str.define_cmd
        let cur_item = 2
        if not data_menu() then exit while end if
      when scratch = str.help_cmd
        let cur_item = 3
        if not hlp_menu() then exit while end if
      when scratch = str.quit_cmd
        let cur_item = 4
        if not quit_menu() then exit while end if
      otherwise
        let cur_item = 1
    end case
end while
```

Dynamic Menu—Pull Down Type

This example builds a small pull-down type menu, sets the activation on certain menu items, prompts for a selection, and processes the selected action including left and right flow control.

```
# It this pulldown is not active active load it
if menu_item != "m_report"
then
    # Set up the report pulldown
    call menuput(str.r_new)                  # New Report
    call menuput(str.r_load)                 # Pick a Report
    call menuput("")                         # -----------------
    call menuput(str.r_report)               # Report Definition
    call menuput(str.r_choose)               # Choose Columns
    call menuput(str.r_arrange)              # Arrange Columns
    call menuput(str.r_totals)               # Totals/Subtotals
    call menuput(str.r_format)               # View the Report
    call menuput("")                         # -----------------
    call menuput(str.r_sort)                 # Data Selection
    call menuput(str.r_print)                # Print the Report
    call menuput("")                         # -----------------
    call menuput(str.r_save)                 # Save the Report
    call menuput(str.r_exit)                 # Exit Program
    # Set up the menu control
    call menuhold()
    call menuwrap(false)
    call menupos(4, x_report + 1, 20)
    call menuhelp("report", "main.report", 1)
end if
# Activate/de-activate menuitems
# Data context required
if m_rept.tabname is null
then
    call menuactive(str.r_report, -1)
    call menuactive(str.r_choose, -1)
else
    call menuactive(str.r_report, 0)
    call menuactive(str.r_choose, 0)
end if
# At least one column must be selected
if fld_cnt = 0
then
    call menuactive(str.r_arrange, -1)
    call menuactive(str.r_print, -1)
else
    call menuactive(str.r_arrange, 0)
    call menuactive(str.r_print, 0)
end if
# Set program context variables
let menu_item = "m_report"
# Open the report pulldown menu
if menupick("") then end if
# Process the report menu selection
case
  when scr_funct = "right"
    call menuclose("")
    let scr_funct = "m_data"
  when scr_funct = "left"
```

```
    call menuclose("")
    let scr_funct = "m_quit"
  when scr_funct = "cancel"
    call menuclose("")
    let menu_item = "mainmenu"
    let scr_funct = null
  when scr_funct = "accept"
    case
      # New report
      when scratch = str.r_new
        # Define a new report
        call rept_flow(true)
      # Pick a report
      when scratch = str.r_load
        call load_rpt()
      # Report description
      when scratch = str.r_report
        call rept_flow(false)
      # Choose columns
      when scratch = str.r_choose
        call flds_flow(true)
      when scratch = str.r_arrange
        call flds_flow(false)
      # View the report
      when scratch = str.r_format
        call view_rpt()
      # Save the current report
      when scratch = str.r_save
        call rpt_save(m_rept.rptname, true) returning tmp_rpt
      # Define subtotal data
      when scratch = str.r_totals
        call subt_flow()
      # Define sort and selection criteria
      when scratch = str.r_sort
        call ssel_flow()
      # Print the report
      when scratch = str.r_print
        call print_rpt(tmp_rpt, tmp_sel)
      # Prompt to confirm exit
      when scratch = str.r_exit
        if ok_2exit()
        then
            return false
        end if
    end case
    # End of pullmenu actions
    # Return to this menu item
    let scr_funct = "m_report" # Return to command line
end case
# End of pullmenu action
```

Nested Dynamic Menus

This example builds three nested menus. The first is used just to display (not select) a report column, the first "real" menu is a special comparison operator menu depending on the type of the column to be compared (help is keyed depending on the menu built), the last is a boolean operator window to continue the comparison sentence.

```
# Open the window displaying the column name
call menuput(fldname)
call menupos(row_pos, col_pos, 0)
call menuview()
# Build the operator menu
case
  when tmp_type = "char"
    call menuput(str.c_begins)       # Begins With
    call menuput(str.c_matches)      # Matches
    call menuput(str.c_equals)       # Equals
    call menuput(str.c_list)         # Is in List
    call menuput(str.c_between)      # Between
    call menuput(str.c_contains)     # Contains
    call menuput(str.c_ends)         # Ends With
    call menuput("")                 # -------------
    call menuput(str.c_notequal)     # Doesn't Equal
    call menuput(str.c_notmatch)     # Doesn't Match
    call menuput(str.c_notlist)      # Is Not in List
    call menuhelp("report", "selector", 3)
  when tmp_type = "date"
    call menuput(str.d_equals)       # Equals
    call menuput(str.d_after)        # After
    call menuput(str.d_before)       # Before
    call menuput(str.d_between)      # Between
    call menuput(str.d_list)         # Is in List
    call menuput("")                 # -------------
    call menuput(str.d_notequal)     # Doesn't Equal
    call menuput(str.d_notbtwn)      # Not Between
    call menuput(str.d_notlist)      # Is Not in List
    call menuhelp("report", "selector", 4)
  otherwise
    call menuput(str.o_equals)       # Equals
    call menuput(str.o_between)      # Between
    call menuput(str.o_grtrthan)     # Greater Than
    call menuput(str.o_lessthan)     # Less Than
    call menuput(str.o_gtequal)      # Greater or Equal
    call menuput(str.o_ltequal)      # Less or Equal
    call menuput(str.o_list)         # Is in List
    call menuput("")                 # ----------------
    call menuput(str.o_notequal)     # Doesn't Equal
    call menuput(str.o_notlist)      # Is Not in List
    call menuhelp("report", "selector", 5)
end case
```

```
let y = row_pos + 3
let x = col_pos + tmp_len + 3
call menupos(row_pos, x, 0)
call menuhold()
# Loop until done to combine multiple selection with and/or
while true
    # Pick from the menu
    let n = menupick("")
    # Process the comparison selection
    if n > 0
    then
      case
        when scratch = str.c_begins
          call lib_prompt(str.c_begin2_pmt,str.c_begin1_pmt,y,x,"")
        when scratch = str.c_matches
          call lib_prompt(str.c_match2_pmt,str.c_match1_pmt,y,x,"")
        when scratch = str.c_equals or
          scratch = str.d_equals or
          scratch = str.o_equals
            call fld_prompt(maintab, str.equal1_pmt,
                str.equal2_pmt, y, x, tmp_type)
        when scratch = str.c_notequal or
          scratch = str.d_notequal or
          scratch = str.o_notequal
            call fld_prompt(maintab, str.notequal1_pmt,
                str.notequal2_pmt, y, x, tmp_type)
        when scratch = str.c_list or
          scratch = str.d_list or
          scratch = str.o_list
            call lib_list(str.list1_pmt, str.list2_pmt, 0, 0, "")
        when scratch = str.c_notlist or
          scratch = str.d_notlist or
          scratch = str.o_notlist
            call lib_list(str.notlist1_pmt,str.notlist2_pmt,0,0,"")
        when scratch = str.c_contains
          call lib_prompt(str.c_cont2_pmt,str.c_cont1_pmt,y,x,"")
        when scratch = str.c_ends
          call lib_prompt(str.c_end2_pmt, str.c_end1_pmt, y, x, "")
        when scratch = str.c_notmatch
          call lib_prompt(str.c_nomch2_pmt,str.c_nomch1_pmt,y,x,"")
        when scratch = str.d_after or
          scratch = str.o_gtequal
            call fld_prompt(maintab, str.o_gteq1_pmt,
                str.o_gteq2_pmt, y, x, tmp_type)
        when scratch = str.d_before or
          scratch = str.o_ltequal
            call fld_prompt(maintab, str.o_lteq1_pmt,
                str.o_lteq2_pmt, y, x, tmp_type)
        when scratch = str.d_between or
          scratch = str.o_between or
          scratch = str.c_between
            call lib_btwn(str.btwn1_pmt, str.btwn2_pmt, 0, 0, "")
                returning tmp_dat1, tmp_dat2
```

```
          when scratch = str.d_notbtwn
          when scratch = str.o_grtrthan
            call fld_prompt(maintab, str.o_grtr1_pmt, str.o_grtr2_pmt,
                 y, x, tmp_type)
          when scratch = str.o_lessthan
            call fld_prompt(maintab, str.o_less1_pmt, str.o_less2_pmt,
                 y, x, tmp_type)
          otherwise continue while
        end case
        # Build and call the and/or/done menu
        call menuput(str.x_and)          # And
        call menuput(str.x_or)           # Or
        call menuput("")                 # ----
        call menuput(str.x_done)         # Done
        call menupos(row_pos, x + 3, 0)
        call menuhelp("report", "selector", 6)
        let n = menupick("")
        # Process a cancel request
        if n = 0 then let scratch = str.x_done end if
        # Process for and/or or done
        case
          when scratch = str.x_and
            let and_or = "and"
          when scratch = str.x_or
            let and_or = "or"
          when scratch = str.x_done
            call menuclose("")   # comparison operator window
            exit while
        end case
      end if
  end while
  # Done
  call menuclose("")   # column name display window
```

Zoomable Dynamic Menu With Parallel Reference Array

This example builds a menu of tables selected from the query in `scratch`. The menu uses either table names or table descriptions depending on a `name_type` flag. The table system names are kept in a parallel list using an array (`tab_array`).

```
# Load the table menu
prepare tab_query from scratch
declare tab_cur cursor for tab_query
foreach tab_cur into tabname, tab_desc
    if name_type = "table"
    then
        call menuput(tabname)
        let tab_array[n] = tab_desc
```

```
    else
        call menuput(tab_desc)
        let tab_array[n] = tabname
    end if
end foreach
# Finish setting up the menu
call menuhold()
call menuzoom()
call menuhelp("rpt_lib", "table_pick", 2)
# Loop for selection
while true
    # Invoke the menu
    let n = menupick(str.tab_choose)
    if n > 0 then
        if scr_funct = "zoom"
        then
            call menuput(tab_array[n])
            call menuhelp("rpt_lib", "table_pick", 3)
            if menupick("") then end if
            continue while
        end if
        if name_type = "table"
        then
            let tabname = scratch
            let tab_desc = tab_array[n]
        else
            let tab_desc = scratch
            let tabname = tab_array[n]
        end if
    else
        let tabname = null
        let tab_desc = null
    end if
    exit while
end while
```

Automatic Dynamic Menu

This example uses `menusel()` to build a menu of all of the columns in a given table.

```
# Build the column selection sentence
let scratch = "select colname from systables, syscolumns ",
    "where systables.tabname = '", tab_name clipped,
    "' and syscolumns.tabid = systables.tabid"
# Load the column menu
let n = menusel()
call menuhelp("rpt_lib", "col_pick", 2)
# Invoke the picker
if menupick(str.col_choose)
then
```

```
    return true
else
    return false
end if
```

Multiple Selection Menu With Zoom and Internal Header Lines

This example builds a menu of columns from several different tables. Each group of columns has the table description as an unselectable header within the menu. This menu allows the user to select multiple columns at once and the retrieves them using `menusget()`. The "real" table and column name for each menu item is kept in a parallel array.

```
# First put the primary table heading
call tdesc_lkup(tabname) returning tmp_name, tmp_desc
let tmp_str = upshift(tmp_desc)
call menuput(tmp_str)
call menuactive("", -2)
# Set the parallel array starting index
let n = 1
# Build the column menu for the primary table
open col_nam_cur using tabname
foreach col_nam_cur into tmp_name, tmp_desc, tmp_ord
    # Store the column
    let n = n + 1
    let names[n].colname = tmp_name
    let names[n].tabname = tabname
    # Put the column description on the menu
    call menuput(tmp_desc)
end foreach
# Add all columns from related tables
foreach rel_tab_cur into tmp_tab, tmp_desc
    # Put the table name into the menu as a header
    call menuput(" ")
    let tmp_str = upshift(tmp_desc)
    call menuput(tmp_str)
    call menuactive("", -2)
    # Adjust the index for the non-active menu lines
    let n = n + 2
    # Build the column menu for the related table
    open col_nam_cur using tmp_tab
    foreach col_nam_cur into tmp_name, tmp_type, tmp_desc, tmp_ord
        # Store the column
        let n = n + 1
        let names[n].colname = tmp_name
        let names[n].tabname = tmp_tab
        # Put the column description on the menu
        call menuput(tmp_desc)
    end foreach
```

```
end foreach
# Set up the column picking menu
call menuhold()
call menuzoom()
call menumany()
call menuhelp("report", "fld_pick", 2)
# Call the menu and get the selected items
while true
    # Call the menu
    let n = menupick(head)
    if n = 0 then return false end if
    # Check for a "zoom" and build a new menu of all indirectly
    # related tables then column pickers for the selected table
    if scr_funct = "zoom"
    then
        call rel_flds(grpname, tabname)
        continue while
    end if
    # Get the selected items
    while true
        # Get the next item
        call menusget() returning n, tmp_ord
        # Check to see if we are done
        if n = 0 then exit while end if
        # Insert the item
        let tmp_desc = scratch
        insert into tmp_flds values (names[n].tabname,
            names[n].colname, tmp_desc, tmp_ord)
    end while
    # Exit when done
    if n = 0 then exit while end if
end while
# Clean up
call menuclose("")
```

Scrolling Field

With an input array you need to make the dummy field part of the `p_` array and store the "real" data in the `q_` array. You can use this strategy in both header and detail inputs. You can define the field on the screen as a formonly field, rely upon the `q_` record having the "real" field, and implement the logic to always use the real field for the `fg_getfield()` input and copy that input (properly truncated with ellipses) to the `p_` record element after the `fg_getfield()` call. Here is an example that works in both "input" and "input array."

Triggers:

```
input 1

    on_screen_record_prep
        # Prepare the scrolling fields
        call fg_elipse(q_wreptr.rpt_head1, 65)
            returning p_wreptr.tmp_head1

    before_field tmp_head1
        # Scrolling field
        call lib_before("tmp_head1")
        call str_display(str.entr_head1, 76, 22, 1, "white")
        call fg_getfield(q_wreptr.rpt_head1, 10, 11, 2, 3, 65, 256)
            returning hotkey, q_wreptr.rpt_head1
        call fg_elipse(q_wreptr.rpt_head1, 65)
                returning p_wreptr.tmp_head1
        display "" at 22, 1
        call lib_after()
        # Process other events
        if hotkey > 0
        then
            let nxt_fld = "event"
            # UP or LEFT
            if hotkey = 137 or hotkey = 139
                then let nxt_fld = "rpt_desc"
            end if
            # DOWN, RIGHT, or ENTER
            if hotkey = 138 or hotkey = 140 or hotkey = 13
                then let nxt_fld = "tmp_head2"
            end if
        end if;
```

The following takes place in the input statement or input array with applicable variable changes in the code below:

```
    input p_wreptr.* without defaults from s_wreptr.*

applied triggers:

        #_before_field tmp_head1
            # Scrolling field
            call lib_before("tmp_head1")
            call str_display(str.entr_head1, 76, 22, 1, "white")
            call fg_getfield(q_wreptr.rpt_head1, 10, 11, 2, 3, 65, 256)
                returning hotkey, q_wreptr.rpt_head1
            call fg_elipse(q_wreptr.rpt_head1, 65)
                    returning p_wreptr.tmp_head1
            display "" at 22, 1
            call lib_after()
            # Process other events
            if hotkey > 0
```

```
            then
                let nxt_fld = "event"
                # UP or LEFT
                if hotkey = 137 or hotkey = 139
                    then let nxt_fld = "rpt_desc"
                end if
                # DOWN, RIGHT, or ENTER
                if hotkey = 138 or hotkey = 140 or hotkey = 13
                    then let nxt_fld = "tmp_head2"
                end if
            end if
        #_end

    #_on_screen_record_prep
        # Prepare the scrolling fields
        call fg_elipse(q_wreptr.rpt_head1, 65)
            returning p_wreptr.tmp_head1
```

The only special issue to be aware of with using scrolling fields in input array is that an exit from the scrolling field with an up arrow or with a down arrow cannot be used to move you to the previous or next row since there is no Informix facility to allow for this kind of programmatic control (for instance no next row or prev row commands). The same applies to page up and page down ([F4] and [F3]) since you are collecting the input keystroke and cannot programmatically translate the requested item (page up or page down) to the required behavior.

Warning Box With Simple Ok (Verify) Option

This example sets the help for the warning box, loads the warning text and calls the `warnbox()` function.

```
# Check for no tables selected
if n = 0
then
    call warnhelp("report","table_pick", 10)
    call warnput(str.no_tables)
    call warnbox()
    return
end if
```

Warning Box With Yes/No/Cancel Selections

This example prompts to save or undo changes. YES throws away the changes, NO keeps the changes, and CANCEL returns you to the input loop.

```
  after input
    if int_flag
    then
```

```
        call warnhelp("report", "save_chng", 10)
        call warnput(str.cancel)
        if not warnyn() and not int_flag
        then
            let scr_funct = "accept"
        else
            if int_flag
            then
                let int_flag = 0
                next field fldname
            end if
            let scr_funct = "cancel"
        end if
    end if
    exit input
```

The Fitrix C Library

The Fitrix C Library has been incorporated as a part of the *CASE* Tools Enhancement Toolkit, and the functions documented below are only available if you have purchased and installed the *CASE* Tools Enhancement Toolkit.

Installation of the *CASE* Tools Enhancement Toolkit creates the directory `$fglibdir/lib/c_lib.4gs`, and the files within this directory are `mkrunners`, `README`, and `fgiusr.c`. These files are needed in order for the client to create their custom `fglgo` and `fgldb` executables. When using C functions and RDS code, you have to run the finished programs with the modified runner `fglgo` and modified `fgldb`. There is an Informix utility named `cfgldb` and `cfglgo` that uses `fgiusr.c` to create these custom runners. All the user has to do is go into `$fglibdir/lib/c_lib.4gs`, and run the shell script `mkrunners`. This creates these two custom runners and moves them to `$fg/bin`. It is important that `$fg/bin` comes before `$INFORMIXDIR/bin` in the environment variable `$PATH` setting. Again, these custom runners would only need to be created if you develop or run programs under RDS.

A common cause for the failure of `cfglgo` is that no C development system is installed. The surest way to build this custom runner is with the C development system installed. However, the C development system is not required. The custom `fglgo` runner can be built using `ld`. The use of `ld` varies from one platform to the next, and the `mkrunners` script tried the most common method for running `ld` to build the custom `fglgo`. The exact command used is:

```
ld /lib/crt0.o fgiusr.o c_*.o $GOLIBES -o fgldb -lc
```

The use of `ld` may vary from system to system. Your system manual may describe in more detail how the `ld` command should be built in the manual entries for `ld` or `cc`. You may need some help from your system administrator. Unfortunately our organization cannot provide support for the creation of the custom runner. We attempt to provide a list of all known variations for the proper `ld` command as part of this documentation.

Known variations for the `ld` command:

RS/6000 AIX 3.2 Users:

```
ld -H512 -T512 -bhalt:4 /lib/crt0.o fgiusr.o c_*.o
$GOLIBES -o fgldb -lc
```

Note

On RS/6000s, it may be necessary to link in the BSD library in order for `mkrunners` to find the `ftime()` function. You can do this by specifying linker flag:

```
-lbsd
```

Or you can append this flag to either of the `/bin/cc` or `/bin/ld` commands with the same affect.

The `fgiusr.c` source file, which is used by Informix utilities `cfgldb` and `cfglgo` to create the custom runners, has been included for you to modify if you wish to also include your own C functions along with these provided with Fitrix *Screen*.

Note

Due to the many variations of both the C compiler and the linker between all the UNIX platforms, we do not support any problems you may incur using these custom runners. We are only offering to you, free of cost, the ability to use the C functions that we currently use in Fitrix *Screen*.

The C functions

`c_getkey()` is a C function designed to be called from an INFORMIX-4GL program. The function reads and identifies a keystroke entered from the keyboard relying on the Informix functions `rgetkey()` and `mvcur()`. It accepts row and column coordinates as its two arguments. The row and column coordinates are relative to the current INFORMIX-4GL window and give the cursor position where the function reads a keystroke. If the row equals zero then the cursor is not positioned. The function returns a special code used to identify the keystroke and a one character string value if the keystroke was a printing character. These are the return codes:

```
character 0
control keys ^A through ^Z 1-26 respectively
function keys F1 through F36 101-136 respectively
escape 135
interrupt 136
up arrow 137
down arrow 138
left arrow 139
right arrow 140
backspace 141
page down 142
page up 143
insert character 146
delete character 147
home key 148
insert line 150
delete line 151
unknown key 152

usage: call c_getkey(y, x) returning key_code, key_string
```

`c_readfile()` is a C function designed to be called from an INFORMIX-4GL program. It accepts a path name as its only argument and returns a status flag and string read from the given file. The status flag has the value "true" if the read was successful and "false" if the read failed. If the file cannot be opened, "failed" (-1) is returned. Successive reads to the same file can be used to read the file sequentially. In order to start over reading a file from the beginning, you must first close the file for reading and then call `c_readfile()` again. The easiest way to accomplish this is by calling `c_readfile()` with a blank path_name argument.

```
usage: call c_readfile(path_name) returning stat_flag, string
```

`c_command()` is a C function designed to be called from an INFORMIX-4GL program. It accepts an OS command as its only argument and returns a status flag, a command status flag, and string read from the output of the OS command. The sta-

tus flag has the value one if the command executed successfully and some output was read; it has the value one if the command executed successfully but there is no output; otherwise it is minus one if the OS command could not be executed or exited with a non-zero exit status. The command status flag gives the exit status of the command when there is no output to read. The status flag is -1 under the following circumstances: a `chdir()` to the requested directory failed (cmd_status = 0), the command was not executable (cmd_status = 0), or the command exited with a non_zero exit status (cmd_status = the exit value ignoring signal exit values). Successive calls with the same command does not re-execute the command until all output has been read at which point a zero `stat_flag` would be returned (unless the command exits non-zero).

```
usage: call c_command(os_command) returning stat_flag,cmd_flag,string
```

Note: There should be a call `c_command("")` placed after every `c_command()` usage. If the buffer has to be retained during a period of (run) time, for instance, in a while loop, then make sure there is no `run cmd` being executed during that period. A `run cmd` messes up the buffer if it's not empty, which causes the program to hang.

`c_statfile()` is a C function designed to be called from an INFORMIX-4GL program. It accepts a path name as its only argument and returns the size of the file, the last modification time as an integer, and a string with "rwx" permissions (if one is denied it has "-" instead or the letter value).

```
usage: call c_statfile(path_name) returning f_size, m_time, perm_str
```

`c_getenv()` is a C function designed to be called from an INFORMIX-4GL program. It takes an environment variable name as an argument and returns a string containing the value.

```
usage: call c_getenv(variable_name) returning value_str
```

`c_putenv()` is a C function designed to be called from an INFORMIX-4GL program. It takes an environment variable name and value as arguments and enters the new value into the environment. If the new value is empty it removes the variable from the table altogether. It returns true if successful and false if it fails.

```
usage: call c_putenv(variable_name, value_str) returning true/false
```

`c_time()` is a C function designed to be called from an INFORMIX-4GL program. It returns the current system time as an integer that can be used to compare with the modification times on files returned by `c_statfile()` or with other integer times returned by `c_time()`.

```
usage: call c_time() returning current_time
```

`c_writeout()` is a C function designed to be called from an INFORMIX-4GL program. It accepts a path name, a string to output, and an append (a), write (w), or close (c) mode flag and returns a status flag. If the file is a new file the file is opened for "append" or "write" depending on the mode flag argument. If the mode argument is "c" for close, then the current opened file (if any) is closed and the function immediately returns. The status flag may have the value -1 if the file for output could not be opened. Otherwise the status flag has the value 1. Successive writes to the same file can be used to write more than one line to the file.

Note

When you are finished writing a file you should call `c_writeout()` with a "c" mode flag to flush the output buffer to the file and close the open file. An immediate call to `c_writeout()` with a different file name serves the same purpose.

```
usage: call c_writeout(path, str, mode) returning stat_flag
```

5

Fitrix Security

You can think of security in terms of levels. Fitrix Security defines three levels for both system users and our applications. By using a hierarchical structure, Fitrix Security establishes a permissions precedence. Once you understand the hierarchy and the logic behind Fitrix Security, you can design a security plan appropriate for your users and system components.

This section covers the following topics:

- How Security Works
- The Security Programs

How Security Works

As mentioned before, Security is based on a hierarchy. You design your security system around three levels of users. In addition, our applications are divided into three levels. The key to setting up a quality security system depends on your understanding of these levels and how they relate to each other.

User Level	Description
Individual User	This level defines system users on a unique or individual basis. All system users, in other words anyone able to log in to the system, are considered individual users. You can grant individual users explicit allow or deny permission settings.
User Group	This level is made up of a subset of system users. You define and determine the types of groups and the members of each group on your system. When you set permissions for a group, all members of the group are given that permission.
Defaults	This level is made up of all system users. It uses defaults as a keyword that signifies a user group containing every individual user. When you set permissions for defaults, you are setting permissions for all users who do not receive more specific group or individual permissions.

Application Level	Description
Module	A collection of input and output programs that compose an application product, such as General Ledger.
Program	A single program within a module. For instance, General Ledger Setup is an input program within the General Ledger module.

Application Level	Description
Event	An activity or command within a program. For example, many input programs let you Update current information. The Update command, then, is considered an event.

Security Programs

Fitrix Security is a collection of programs that let you define security permissions for each level of user and application. Security consists of five input programs. These programs work interactively. In other words, information defined in one program is used to provide information for another program.

Program Name	Description
Module and Program Information To run, type fg.modules.	This program lists the Fitrix modules and programs on your system. By default, this information comes pre-loaded in Security.
Security Events To run, type fg.events.	This program lists the events used by the modules and programs on your system. Like modules and programs, event information is pre-loaded.
Security Groups To run, type fg.groups.	This program lets you define which individual users belong to which user group.
User & Group Permissions To run, type fg.users	This program provides a complete method for identifying the users and groups on your system. In addition, it links information in the Module, and Event programs with user and group definitions, and it allows you to set explicit user and group permissions. Most of the work you do with Security is done in this program.

Program Name	Description
Group Security Control To run, type fg.gcontrol.	This program provides an easy-to-use interface for setting up group permissions on common events. It does not contain all the features and flexibility of the User & Group Permissions program, but it is a simplistic alternative.

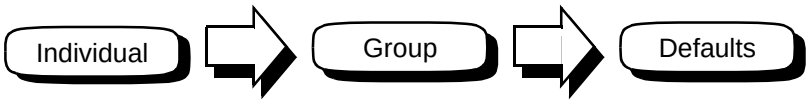
In later sections of this Guide, each program is described in more detail. This section concentrates on how Security takes and uses information supplied to the Security programs and which permission settings take precedence.

Determining Precedence

Security determines precedence in an inverted or "bottom up" manner. In other words, the most specific settings (the individual user settings and the event settings) take precedence over the more general settings.

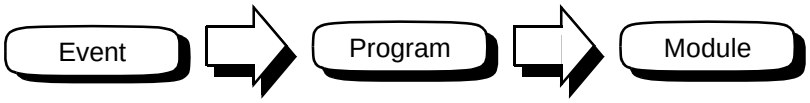
In terms of user levels, Fitrix Security searches for an allow or deny permission first on the individual level, then on the group level, and finally on the global or defaults group level.

User Level Search Order



In terms of application levels, Security looks first at the event level, then the program level, and finally the module level.

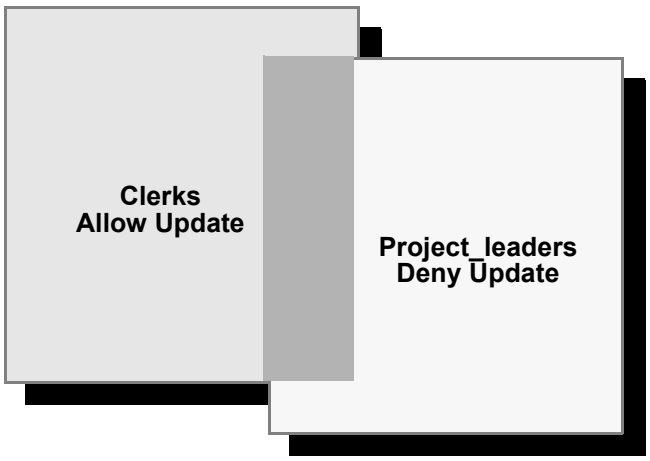
Application Level Search Order



Overlapping Group Permissions

Security is designed to meet as many custom security setups as possible. For this reason, you can place individual users into more than one user group. Sometimes, however, users belong to groups that contain conflicting permission settings otherwise known as overlapping user groups. Users that belong to overlapping groups are given allow permission.

For instance a clerk might belong to a group called `clerks` and a group called `project_leaders`. At times, `clerks` and `project_leaders` might have conflicting permission settings. For instance, `clerks` might allow the Update event and `project_leaders` might deny it.



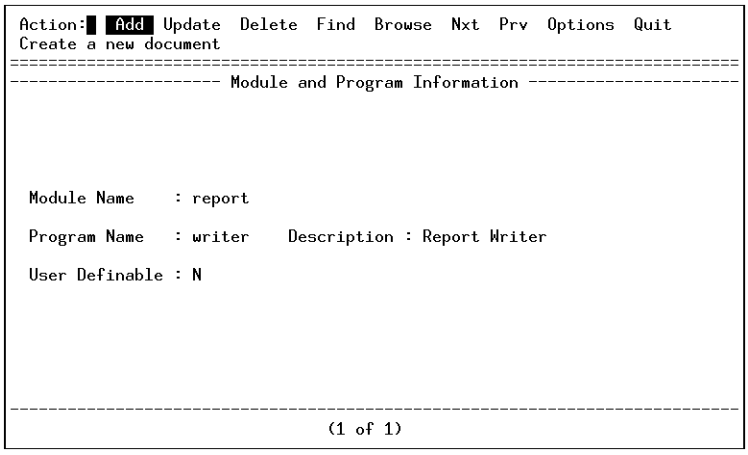
In this situation, the clerk who belongs to both groups is able to use the Update event.

The Security Programs

As mentioned earlier, Fitrix Security is a collection of five input programs. You use all of these programs to define Security on each level of user and application.

Module and Program Information

This input program lets you enter the modules and programs eligible to secure. All Fitrix modules and programs come pre-loaded. You only need to use Module and Program Information when you create custom programs or modules. The following figure shows the input screen for Module and Program Information:



Adding Custom Programs to Module and Program Information

When you create a custom application, the *Report* Code Generator automatically builds logic that Security recognizes. For example, if you create a custom report, you can add that report to Module and Program Information.

To add a custom report to Module and Program Information:

1. **Select Add from the ring menu.**
2. **In the Module Name field, enter the module directory of the custom program.**

   For example, if your custom report is in `sales.4gm`, enter `sales` in the Module Name field.

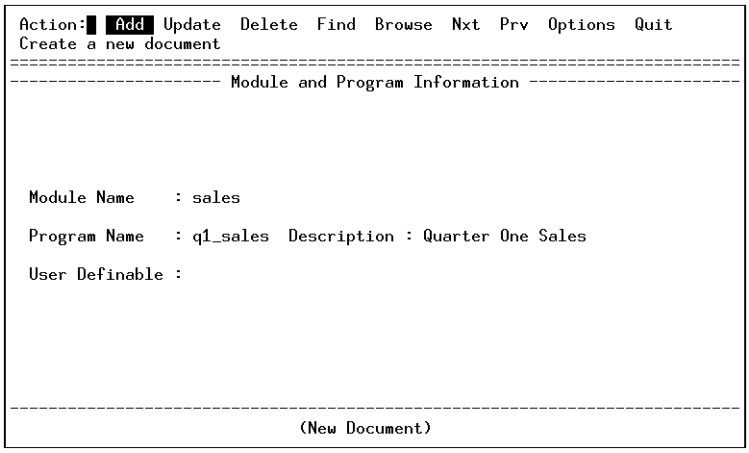
3. **In the Program Name field, enter the program directory that contains your custom report.**

   For example, if your custom report is in `q1_sales.4gs`, enter `q1_sales` in the Program Name field.

4. **Enter a description for your custom report in the Description field.**

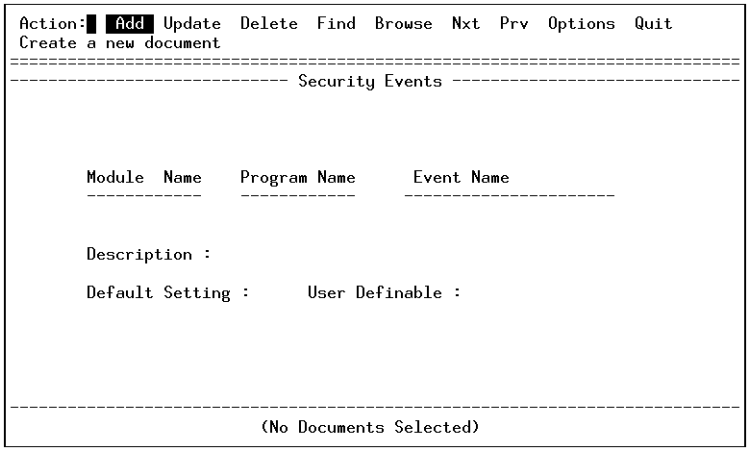
   The User Definable field is a non-entry field. At this time, you can leave this field blank.

5. **Press [ESC] to store your entry.**

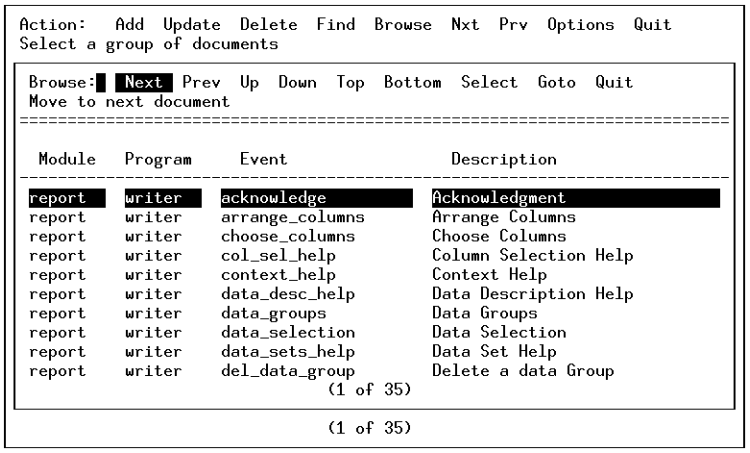


Security Events

This input program is similar to Module and Program Information. It too comes pre-loaded with events used in our programs, such as add, delete, and update. As well, Security Events lets you define custom events in custom programs. Similar to Module and Program Information, Security Events just lets you define events that are eligible to secure.



The following shows some of the 35 events associated with *Report* Writer.



Adding Custom Events to Security Events

If your application contains custom events, you can add these events to the Security Events program. Once added, you can use the User and Group Permissions program to place individual and group permissions on your custom event.

Unlike custom programs, where Security logic gets generated automatically, you must add a few lines of code at the start of your custom events for Security to be able to recognize it.

For example, suppose you create a `q1_sales` program. In `q1_sales`, you create a custom event that allows users to fax report output to company headquarters. At the start of your custom fax event, add the following lines of code:

```
# Inserted for program level security.
# Check for permission
if not security_chk("fax")
then
    call security_msg("fax")
    exit program(100)
end if
```

After you add this code to your custom event, making that event eligible to secure requires the following steps:

1. **Select Add from the ring menu.**
2. **In the Module Name field, enter the module directory of your custom program.**

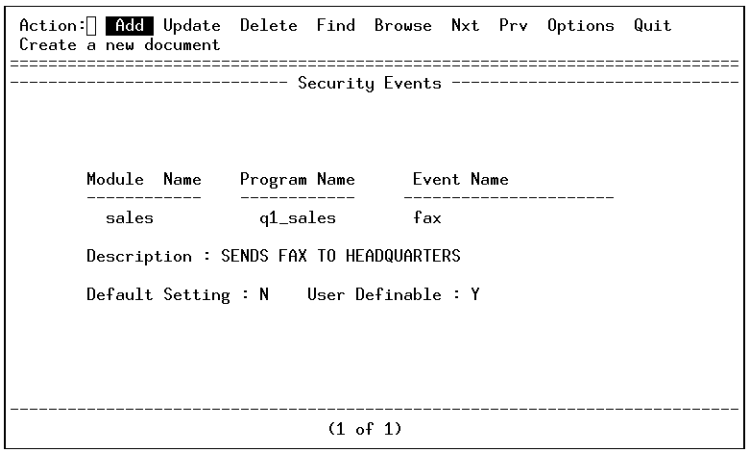
   For example, if the module directory is `sales.4gm`, enter `sales`.
3. **In the Program Name field, enter the program directory of your custom program.**

   For example, if the program directory is `q1_sales.4gs`, enter `q1_sales`.
4. **In the Event Name field, enter the name of your custom event.**

   For example, if the event name is `fax`, enter `fax`.
5. **In the Description field, enter a description of your event.**
6. **In the Default Setting field, enter the default permission for the event.**

   The User Definable field is a non-entry field.

7. Press [ESC] to store your entry.



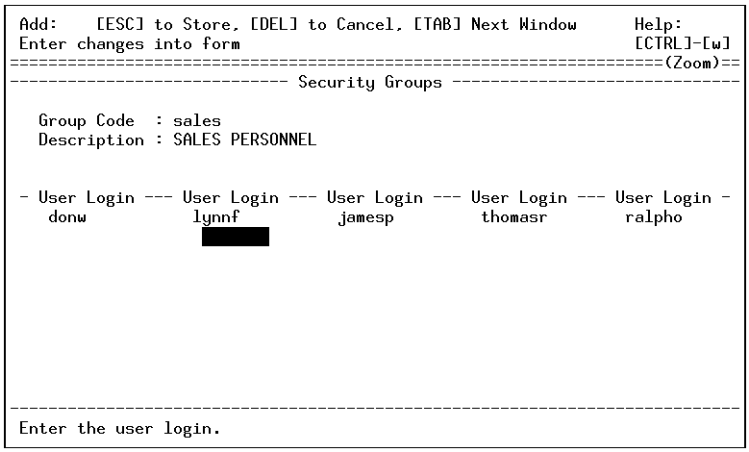
Note

If you want to set permissions for your event in all the programs in a module, leave the Program Name field blank.

Security Groups

This program lets you assign individual users to groups. By creating groups of users, from individuals users who require similar system access, you can simplify your security configuration.

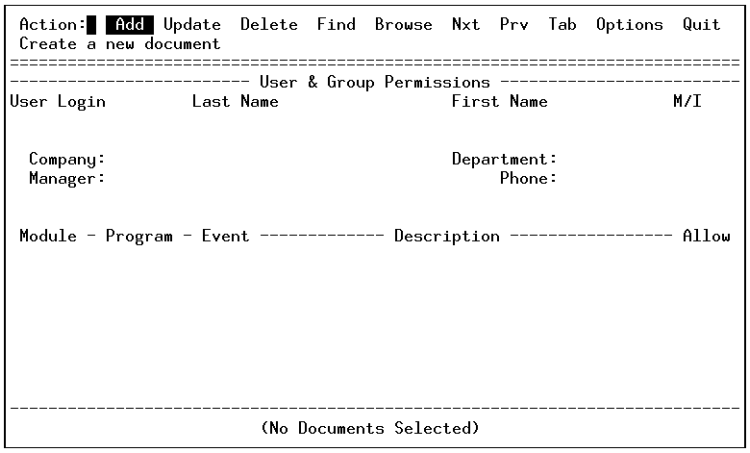
For example, you might want to assign your entire sales force to a group called `sales`. Your definition of the `sales` group might look as follows:



Once you define a security group, you can set permissions for that group in the User and Group Permissions program or in Group Security Control.

User and Group Permissions

This input program is where most of your security work gets done. It is this program that relates the information set in Module and Program Information, Security Events, and Security Groups with actual permission settings.



Setting Individual User Permissions

The most basic task of the User and Group Permissions program is setting permissions for an individual user.

To set permission for an individual user:

1. **Select Add from the ring menu.**
2. **Enter values for the User Login and Last Name fields.**

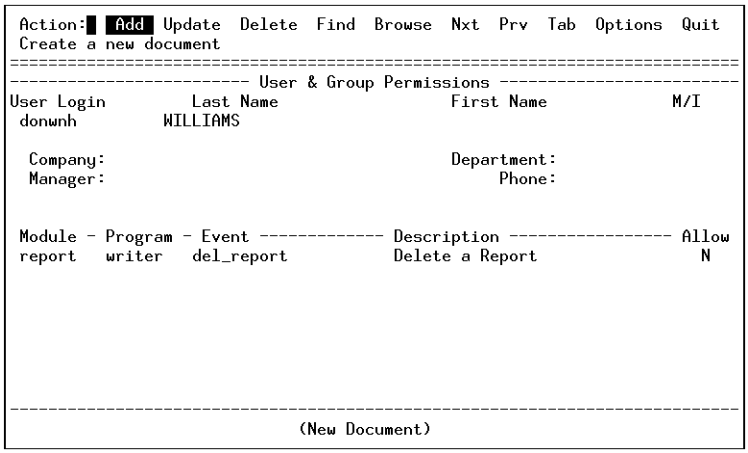
   For example, if you are setting permissions for donw, enter donw in the User Login field and donw's last name (for instance Williams) in the Last Name field.

   The User Login and Last Name fields are the only required fields. The other fields in the header section are optional, such as the Department and Phone fields.

3. Press [TAB] to move to the detail section of the program.

In the detail section you can enter the module, program, and event you want to set permissions on. You can also press [CTRL]-[z] to pick from a list of defined modules, programs, and events.

For example, suppose you want to deny `donw` the ability to delete reports:

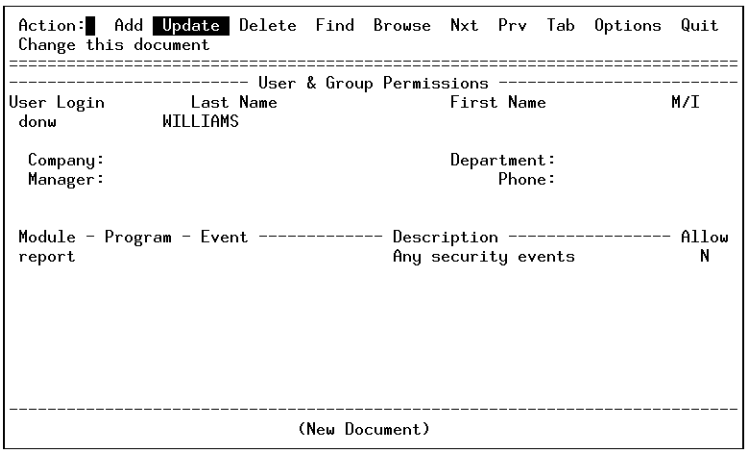


4. Once you finish entering permission data, press [ESC] to store your entry.

Setting Permission for an Entire Module

To set permissions for an entire module, only specify the module name in the detail portion of User and Group Permissions.

For example, to deny donw access to all programs in the report module, make the following entry:



In a similar sense, you can set permissions for all events in a program: specify both the module and program and leave the Event field blank.

Setting Group Permissions

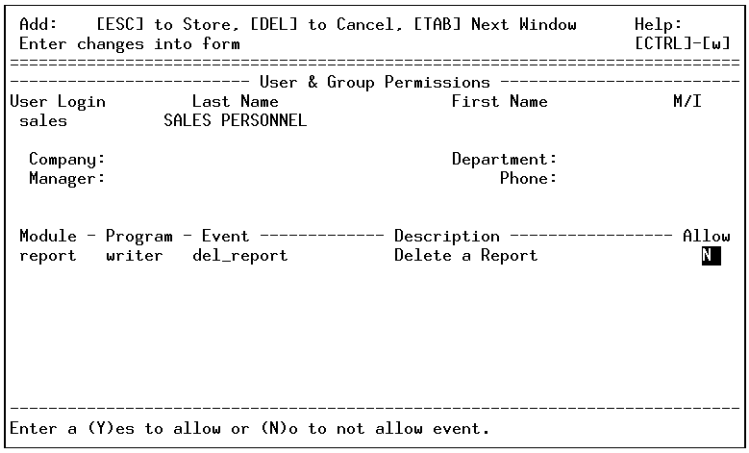
You can also set permissions for groups that you have defined in the Security Group program (see "Security Groups" on page 5-11). In the same way you set permissions for individual users, you also set permissions for groups.

To set permissions for a group:

1. **Select Add from the ring menu.**
2. **Enter the group code (i.e., group name) in the User Login field and enter a description of the group in the Last Name field.**
3. **Press [TAB] to move to the detail portion of the program.**

   In the detail section you can enter the module, program, and event you want to set permissions on. You can also press [CTRL]-[z] to pick from a list of defined modules, programs, and events.

For example, to set permissions of the `sales` group for the delete report event:



4. **Once you finish entering permission data, press [ESC] to store your entry.**

Setting Defaults Permission

The Defaults permission is a reserved permission setting. The values set for Defaults are passed to all users and groups not otherwise defined. For instance, if the user `robertc` does not belong to any groups and does not have an individual user entry, he receives the permissions set in defaults.

To set Defaults permission:

1. **Select Add from the ring menu.**
2. **Enter defaults in the User Login field and DEFAULTS in the Last Name field.**
3. **Press [TAB] to move to the detail section of the screen.**

   In the detail section, enter the module, program, and event you want to set permissions on. You can also press [CTRL]-[z] to pick from a list of defined modules, programs, and events.

4. **Once you complete setting defaults permissions, press [ESC] to store your settings.**

Note

Caution: The Defaults permission affects all users on the system. You should set Defaults permissions during a period of light system use.

Group Security Control

Group Security Control is a simplified version of the User and Group Permissions program. With Group Security Control, the most common program events are already listed. Group Security Control gives you a graphical matrix with which to assign permission settings for a defined group on a defined module.

For example, the following entry shows the permissions for the `account` group on the report module:

```
Update: [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                         [CTRL]-[w]
================================================================================
------------------------- Group Security Control -----------------------------

   Group  : account    ACCOUNTANTS GROUP
   Module : report     Report Module

 Program                          Run Add Upt Del Fnd Brw Tab Opt Bng Hot Nav
--------------------------------------------------------------------------------
 Report Image Loader               Y   Y   Y   Y   Y   Y   N   N   N   N   N
 Report Image Maker                N   N   Y   Y   Y   Y   Y   N   N   N   N
 Report Runner                         Y   Y   Y   N   N   N   N   N   N   N
 Report Writer                     Y   Y   Y   N   Y   Y   N   N   N   N   N




--------------------------------------------------------------------------------
Enter permission for adding or editing Navigation events.
```

The following describes the events available in Security Control.

Event	Description
Run	The Run event controls the use of the listed program. When the Run permission field is set to Y, members of the group can start the listed program. When set to N, the group cannot start the listed program.
Add	The Add event controls the ability to add or create new program documents. When Add is set to Y, documents can be added. When set to N, the group cannot add a document.
Upt	The Upt event specifies a group's ability to update a document. A Y in this field lets group members update a document, an N denies update permission.
Del	The Del event controls document deletion. Many times only specific users are allowed delete permission. When you set the Del event to Y, the group can delete documents. When set to N, documents cannot be deleted.
Fnd	The Fnd event controls a program's Find capabilities. When you set the Fnd event to Y, group members can conduct Query-By-Example searches for specific documents. When set to N, users cannot use the Find feature.
Brw	The Brw event controls the Browse capabilities. When you set Brw to Y, the group can use the Browse command. When set to N, browse privileges are denied.
Tab	The Tab event coincides with the Tab command. When you set the Tab field to Y, the group can use the Tab command. When set to N, group members cannot use the Tab command.
Opt	The Opt event controls access to the Options command. A Y in the Opt field grants access to the Options command, an N denies access.

Event	Description
Bng	The Bng event controls access to the operating system. In most cases, users are able to bang out (also called shell out or escape) to the operating system. When the Bng event is set to Y, the group can bang out of the program. When set to N, the group cannot escape to the operating system.
Hot	The Hot event corresponds to a program's Hot Keys. In many programs, users can define Hot Keys that serve as keyboard shortcuts to common program commands. When you set the Hot event to Y, users can alter the default Hot Key definitions. When set to N, users cannot edit the default Hot Key definitions.
Nav	The Nav event relates to a program's Navigate feature. In many Fitrix programs, users can press [CTRL]-[g] to view the Navigate pop-up menu. When you set the Nav event to Y, users gain the ability to use this menu. When set to N, users cannot use the Navigate menu.

Index

Numerics

A

B

C

D

N

O

P

Q

R

S

T

U

V

W

Z