Fitrix™
CASE Tools
4.12 New Features

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

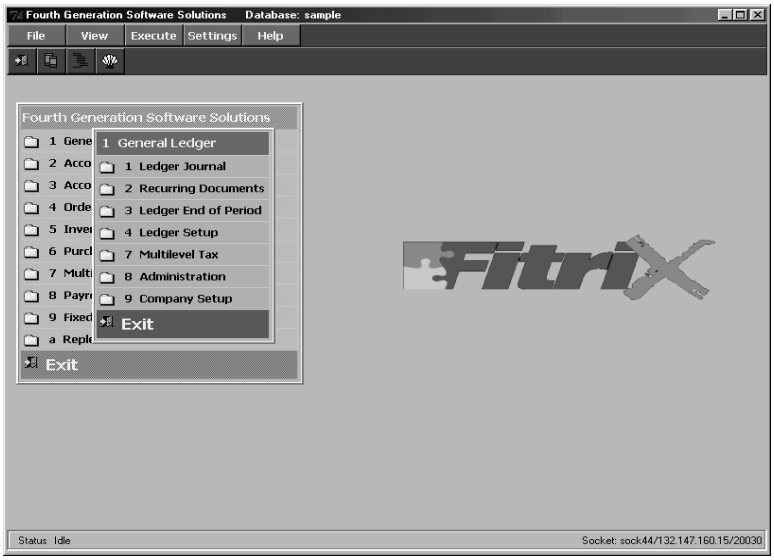
Corporate: (770) 432-7623
Fax: (770) 432-3448
E-mail: info@fitrix.com

Copyright

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

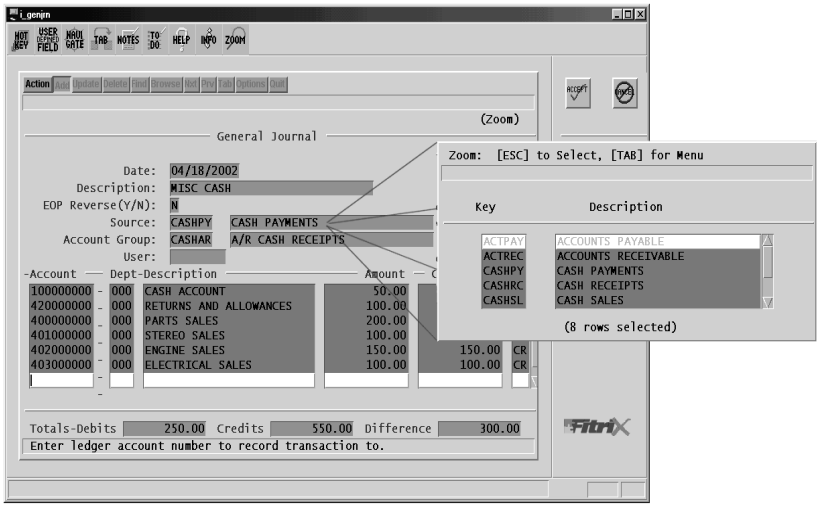
No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated.

Welcome to the Fitrix Case Tools New Features 4.12. 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

New Feature Summaries viii
Part One: Fitrix CASE Tools New Features viii
Part Two: Fitrix Screen New Features viii
Part Three: Fitrix Report New Features ix
Documentation Conventions xi

Part 1:Fitrix CASE Tools New Features

Chapter 1: CASE Tools Demo

Overview 1-2
Starting the Demonstration 1-4
The Menus Command Bar 1-4
The Demo Menu 1-5
Running the Menus Demo 1-6
Running a Screen Demo 1-9
Running the Program 1-10
Building the Program 1-11
Running a Report Demo 1-16
Running the Program 1-17
Building the Program 1-18

Part 2:Screen New Features

Chapter 2: Float Format

Overview 2-2
Setting up a Float Format 2-4
Float Format Functionality 2-5
Rounding 2-5

Acceptable Values 2-7
Applying Float Formats to Screens 2-9
The float_fmt Line 2-9
Float Format Logic 2-11
Applying Float Formats to Reports 2-12
The float_fmt Line 2-12

Chapter 3: Translating Y/N Fields

Overview 3-2
Applying Y/N Logic to Screens 3-4
Using the Form Painter 3-4
Using a Text Editor 3-5
Applying Y/N Logic to Reports 3-6

Chapter 4: Screen Hooking Logic

Overview 4-2
The socket_items Trigger 4-3
A Quick Review 4-3
Adding the socket_items Trigger 4-4

Part 3:Report New Features

Chapter 5: Report Scheduling

Report Scheduling Overview 5-2
Implementation 5-3
Three New Functions 5-3
Incorporating Selection Criteria 5-4
Scheduling Example 5-4
Scheduling Code in the Library 5-6
`flow_control` 5-6
stub function 5-6
Generated Code in midlevel.4gl 5-7

ml_schedule 5-7
`ml_put_filter` 5-8
ml_get_filter 5-9

Chapter 6: Column Aliasing

Overview 6-2
Setting up a Column Alias 6-3
The alias Line 6-3
Changing the Column Format 6-5

Chapter 7: Concurrency

Overview 7-2
Implementing Concurrency 7-5
Setting up the report.ifg File 7-5
Concurrency 7-8
Handling Concurrency Errors 7-10
Example One: Displaying a Warning Message 7-10
Example Two: Writing A Message to the Report 7-13
Code Examples 7-16
midlevel.4gl 7-16
lowlevel.4gl 7-20
Log Files A-2
Creating an Access Log File A-2
Relocating the errlog File A-3
Generator Access Variables A-5
Set Explain Support A-6
Hiding Ring Menu Options A-8
Building a Library Zoom Screen A-10

Preface

This manual provides information and insight concerning the new features and functionality of the 4.12 Fitrix *CASE* Tools. Besides maintenance fixes and other enhancements, a majority of the work that went into this release stemmed from our organization's goal to create complete, robust, and language independent code. You will also find that a lot of work has been done to the Fitrix *Report* Code Generator. Among the more salient improvements, you can now implement scheduling and concurrency logic to your report programs.

This chapter contains the following topics:

- New Feature Summaries
- Documentation Conventions

New Feature Summaries

This manual is broken into three parts. The first part discusses new features that apply to the Fitrix *CASE* Tools as a whole. The second part describes new features for the Fitrix *Screen* product, which includes the Fitrix *Screen* Code Generator and the Form Painter. The third and final part covers Fitrix *Report* new features. The following paragraphs highlight these features and serve as a good introduction to the rest of this manual.

Part One: Fitrix *CASE* Tools New Features

The biggest change to the Fitrix *CASE* Tools as a whole involves new demonstration programs and a new demonstration interface. Using the Application Development Manager (AppDev), you can step through the process of building both Fitrix *Screen* and Fitrix *Report* programs. These programs demonstrate several new features, including report scheduling and column aliasing.

In addition, new directories containing executable program files have been set up so you can skip the build phase and launch each demonstration program directly.

Part Two: Fitrix *Screen* New Features

The biggest improvements to the Fitrix *Screen* products include new float formatting package, Y/N field translation logic, and a new trigger command. Both the float formatting package and the Y/N translation logic also apply to Fitrix *Report* programs.

The Float Format Package

Using the Float Formatting package, you can define formats for the way decimal fields are displayed to the screen. The float formatting package also includes rounding logic and it performs "acceptable values" checking (in other words, it will accept a number of different symbols and convert them to display a single default symbol).

Some of the symbols you can define include the thousand separator, decimal separator, front minus, and back minus.

Y/N Field Translation Logic

Because field translation work can be a long and laborious task, this new logic gives you the ability to translate Y/N fields virtually all at once. This new logic sets up a single definition for Y/N fields in the database. Instead of defining each Y/N field individually, you simply add a line to your form specification (*.per) file that applies the Y/N translation definition to the field of your choice.

The socket_items Trigger

Each time you hook in a screen to your screen or report programs, a certain amount of overhead comes along with it. The `socket_items` trigger is intended to limit the number of functions that are linked in with each screen. You use the `socket_items` trigger in conjunction with the `switchbox_items` trigger.

For example, if you want to add a query screen to a report program, you can use both the `switchbox_items` trigger and the `socket_items` trigger. You can supply the query screen name and function name to `switchbox_items` and the keyword "query" to `socket_items`. These two triggers might appear in an extension (*.ext) file as follows:

```
#--------------------------------------------------------------------
# add the switchbox and socket items to main
#--------------------------------------------------------------------
switchbox_items
    query S_query;
socket_items
    query;
```

The `socket_items` trigger dramatically reduces the size of the resulting program. In some cases this reduction is upwards of 80K.

Part Three: Fitrix *Report* New Features

Perhaps the Fitrix *Report* Code Generator received the most attention since the 4.11 release. A lot of work has been done to increase the abilities of the Fitrix *Report* Code Generator including the addition of scheduling logic, concurrency logic, and column aliasing. In addition, the Fitrix *Report* Code Generator also supports some of the same new features as the Fitrix *Screen* Code Generator. These features include the float formatting package and Y/N Translation logic, all of which are are covered in the *Screen* part.

Report Scheduling

Using Scheduling logic, you can set a time for report execution. This ability lets you postpone the processing of large reports until night when system resources are more plentiful. Besides setting the time, Scheduling logic also lets you save selection criteria in the database until the report is run. By saving the selection criteria, you can set both the time of execution and the appropriate selection criteria long before the report actually runs.

Concurrency Logic

Concurrency gives you the ability to check each row in the header table for data integrity prior to processing the corresponding detail lines for that row. This ability helps assure that data doesn't change during report processing. In addition to checking for data integrity, concurrency also provides default locations in the code for you to handle data conflicts and lets you set up your own data integrity logic.

Column Aliasing

With column aliasing, you can now use columns that have the same name but contain different data in the same report program. Quite often different tables may contain columns with the same name. In previous releases of the Fitrix *Report* Code Generator, these columns could not be used on the same report. With the addition of column aliasing, you can assign an alias to one of the columns and use it in conjunction with the other column.

Documentation Conventions

Some information is difficult to convey in text, such as a series of keystrokes or a value you supply. This Technical Reference 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, filenames, and keystrokes from other text.

Text Format	Meaning	Example
Courier Bold	Represents command syntax in addition to variable and file definitions.	**fg.writer**
Courier Bold Italic	Represents text you should replace with the appropriate value.	**-r *report_name***

Courier	Represents commands; code; file, directory, table, and column names; and system responses.	`report.ifg`
		`Makefile`
		`standard`
		`rtmargin`

Small Courier	Represents program code or text in a file.	`output`
		`top margin 3`
		`bottom margin 3`
		`left margin 3`
		`right margin 77`
		`page length 66`

Symbol	Meaning	Example
[ ]	Represents optional command flags or arguments.	**fg.report [-f]**
{ }	Represents a mandatory choice of options.	**{one|two|three}**
|	Delimits choices.	**-y|-n**
...	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 are shown in uppercase and enclosed in brackets, for instance:

[TAB]
[F1]
[ESC]
[ENTER]

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, press [CTRL]-[d].

Some keys differ from keyboard to keyboard. This manual mentions the [ENTER] and [DEL] keys, but both may be missing from your keyboard. Hardware manufacturers give different names to keys that perform the same function.

Keys	Common Variations
[ENTER]	RETURN, RTRN, ↵
[ESC]	STORE
[DEL]	BREAK, CTRL C, CTRL BREAK

Although many similar versions of UNIX and XENIX can run INFORMIX-4GL and the Fitrix *Report* Code Generator, this manual refers to all of them with the single term of UNIX.

Part One

Fitrix CASE Tools New Features

1

CASE Tools Demo

As part of the 4.12 Fitrix *CASE* Tools, several new demonstration programs have been added. These programs give you valuable insight into our organization's development techniques and common program construction. In addition to the new demonstration programs, a new interface for accessing and building these programs was also added. This interface, which is built from Fitrix *Menus*, serves the two following purposes: It gives you an easy-to-use method for initiating the demonstration programs, and it serves as a demo in and of itself.

This chapter covers the following topics:

- Overview
- Starting the Demonstration
- Running the Fitrix *Menus* Demo
- Running a Fitrix *Screen* Demo
- Running a Fitrix *Report* Demo

Overview

The 4.12 Fitrix *CASE* Tools include a completely integrated and flexible group of demonstration programs. These programs not only show you typical Fitrix *Menus*, *Screen*, and *Report* functionality, but they also give you the opportunity to look under the hood and see how these different types of programs work.

Demonstration programs provide the following benefits:

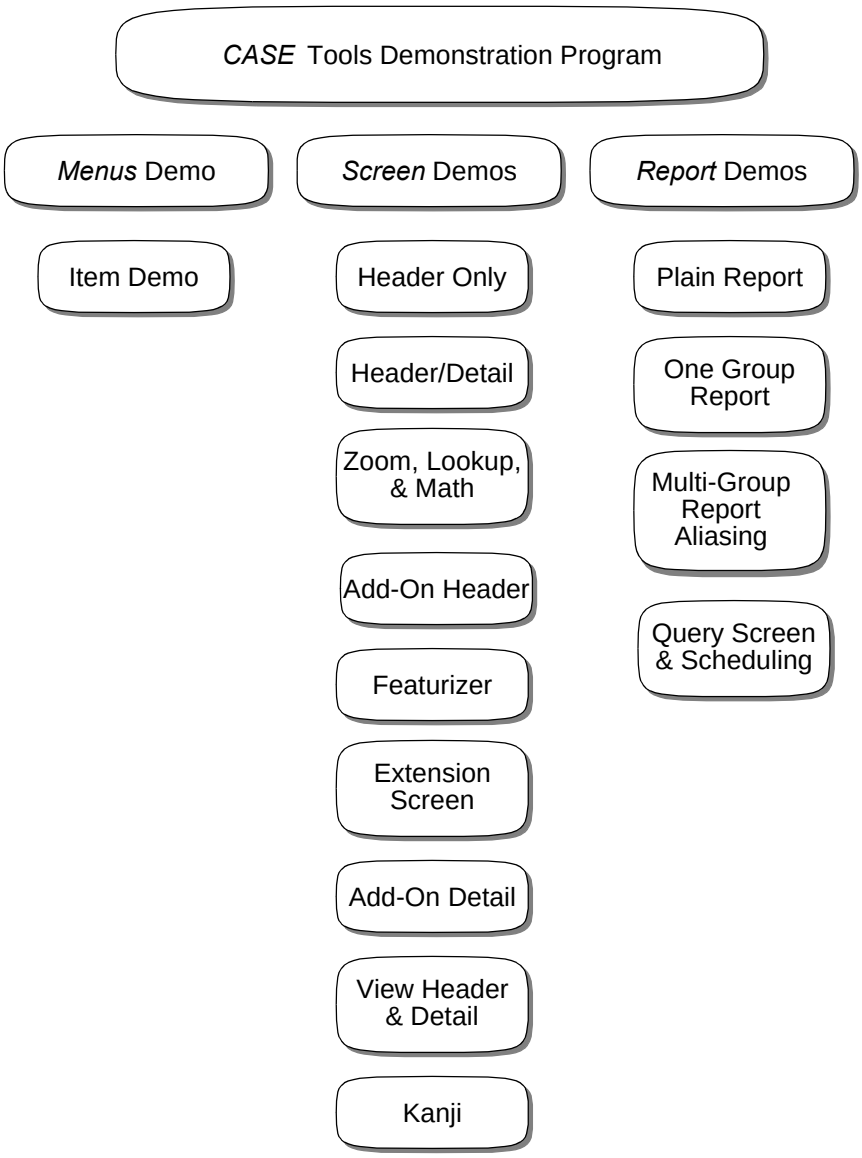
- They introduce new users to basic Fitrix techniques and development standards.
- They provide experienced users with examples of new and improved features.
- They help our developers simulate conditions that may be causing problems.

For example, if you are a new user and you want to see how a simple header screen looks and functions, you can check out screen demonstration one.

Or, being an experienced user, you may want to see how to use the *Report* Code Generator's new aliasing abilities. No problem, you can simply fire off report demonstration three.

Besides helping to answer your questions and show off new functionality, demonstration programs are handy debugging tools. They give our developers a common link between your system and our own. If, for example, you think there is a problem in the way one of your generated programs is working, you can let us know, and—more than likely—we can attempt to duplicate the problem using one of the demonstration programs.

The graphic on the following page describes each demonstration program. Following the graphic, the rest of this section introduces you to the demonstration interface and shows you how to build and run the different demonstration programs.

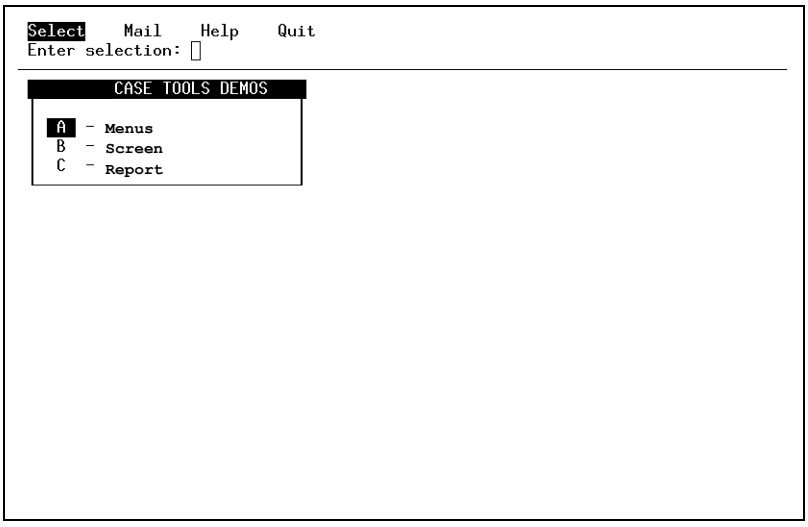

CASE Tools Demonstration Program
Menus Demo
Screen Demos
Report Demos
Item Demo
Header Only
Header/Detail
Zoom, Lookup, & Math
Add-On Header
Featurizer
Extension Screen
Add-On Detail
View Header & Detail
Kanji
Plain Report
One Group Report
Multi-Group Report Aliasing
Query Screen & Scheduling

Starting the Demonstration

To begin working with the new 4.12 Fitrix *CASE* Tools Demonstration, enter the following command at the UNIX prompt:

`fg.demo`

The following screen appears:



This screen contains both the standard Fitrix *Menus* command bar and the <YOUR COMPANY NAME>CASE TOOLS DEMOS menu. The command bar has four options: Select, Mail, Help, and Quit.

The *Menus* Command Bar

Select initiates the highlighted menu item.

Mail starts an E-mail session.

Help opens a window containing help information.

Quit exits the demo.

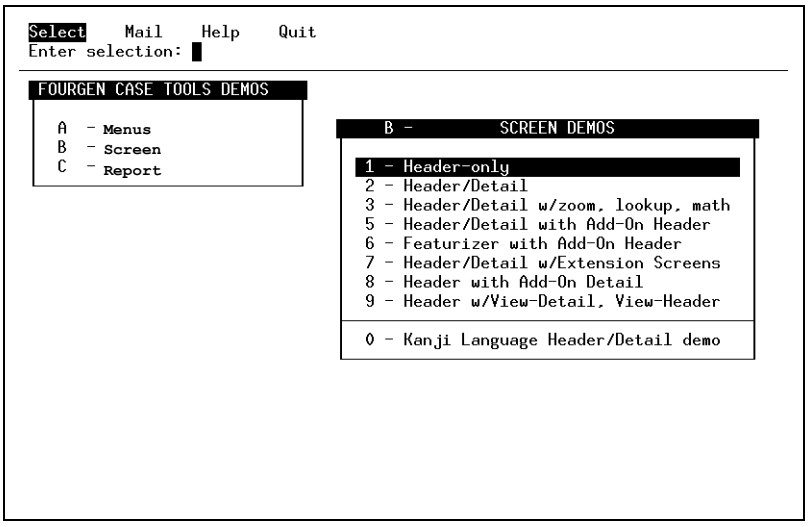
To move the highlight between options on the command bar, use the [SPACE-BAR]. Once an option is highlighted, press [ENTER].

The Demo Menu

The <YOUR COMPANY NAME>CASE TOOLS DEMOS menu contains three menu choices; one for each Fitrix *CASE* Tools product line. To initiate a menu choice, type the alphanumeric character(s) that represents the choice or highlight it and pick Select from the command bar. You can move between menu choices with the arrow keys.

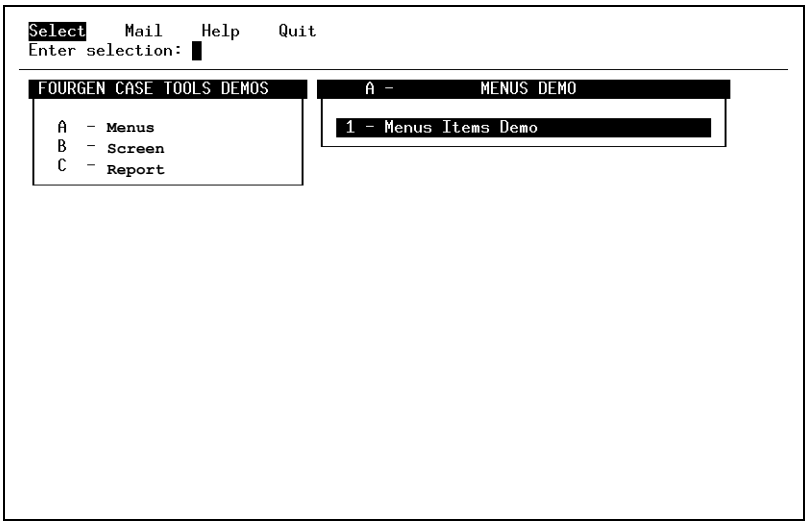
To close a submenu, press [DEL].

For each menu choice there is a submenu containing the various demonstration programs. For example, if you type B to select the *Screen* menu choice, the following submenu appears:

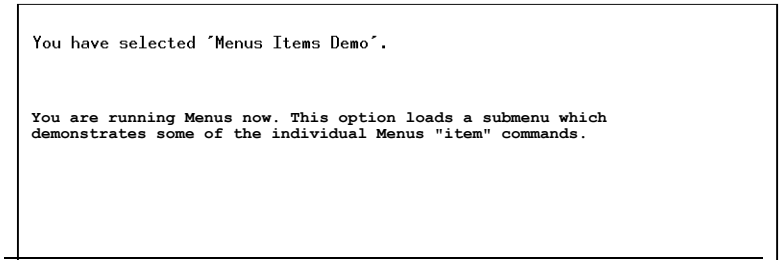


Running the *Menus* Demo

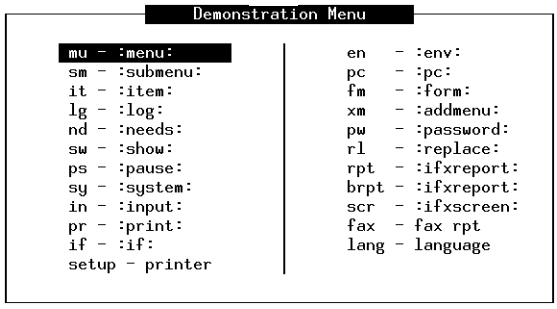
The first choice on the <YOUR COMPANY NAME> CASE TOOLS DEMOS menu is the *Menus* demo. The Fitrix *Menus* product only has one demonstration program, excluding the `fg.demo` program itself. When you type A or highlight it and choose Select from the command bar, the following submenu appears:



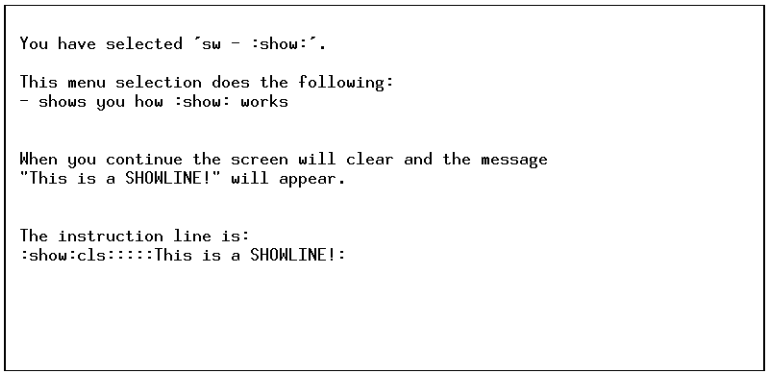
This submenu contains Menus Items Demo. To start it, type 1. As with all the demonstration programs, an information screen appears prior to running the demo. This information screen describes the purpose of the demonstration and provides some simple instructions for using the demo:



Read the information screen and then press [ENTER] to continue. The demonstration *Menu* program appears:



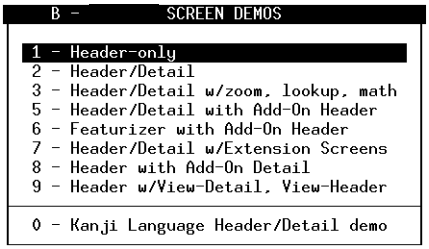
This program describes some of the most common menu item instructions and shows you how these instructions work. For example, the `show` instruction displays a line of text to the user. When you highlight and select the `sw - show` option, the following screen appears:



When you press [ENTER] to continue, the item instruction is carried out so you can see how it works. This demonstration program is very useful if you forget the syntax or how a particular item instruction works.

Running a Screen Demo

The second choice on the <YOUR COMPANY NAME>CASE TOOLS DEMOS menu opens the SCREEN DEMOS menu. This menu contains nine screen demonstration programs:

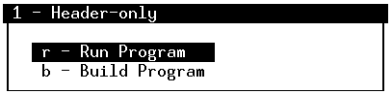


Note

It may look at first as if there are 10 screen demonstrations, but screen demo four has been purposefully left off the menu. Screen demo four is intended to be used primarily with the Form Painter.

Each screen demonstration shows a unique functionality. To run the Kanji screen demonstration you must have the multibyte version of the Fitrix *CASE* Tools, and hardware, O/S, and Informix that supports Kanji.

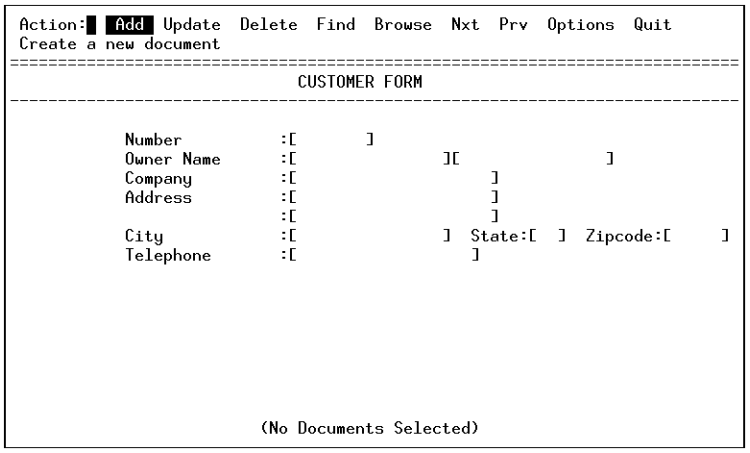
When you select a screen demonstration program, you have two choices:



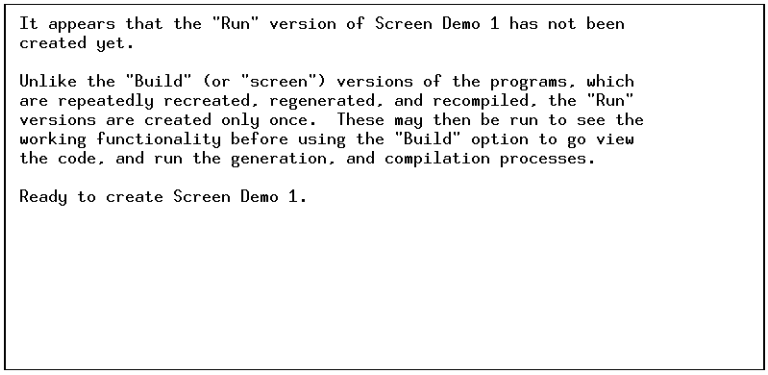
1. You can run the generated and compiled program directly.
2. You can build the program from its form specification (*.per); and, if present, trigger (*.trg) and extension (*.ext) files.

Running the Program

If you choose to run the program, an information screen appears that describes what the demonstration covers. After reading the information screen, press [ENTER] to continue and run the program. For example, if you run screen demonstration one, the following program appears:



You may encounter a case where the actual executable has yet to be created. If this occurs, you are given the choice to create it:



When you press [ENTER] to continue, the following line appears:

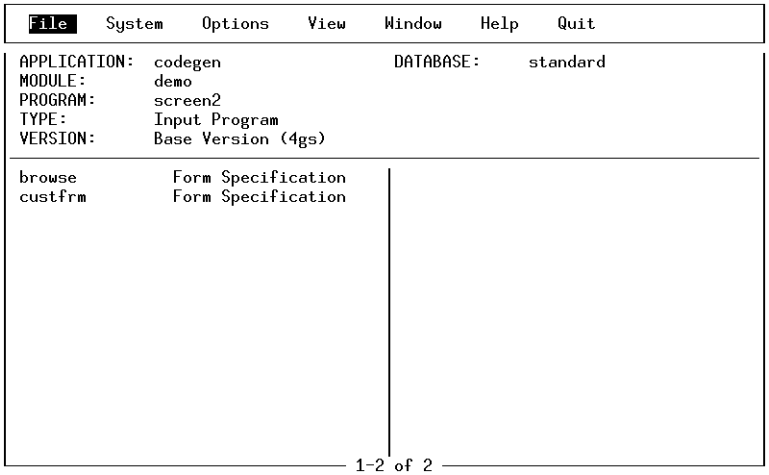
```
Enter [RETURN] for Demo 1, or [A] to prepare ALL Screen Demos:
```

This prompt gives you the option of building only the selected program or building all screen programs at once. Note that you should only encounter this circumstance once. After the executable is built, you will not have to build it again each time you select Run Program. Whichever option you select, the proper executable(s) is prepared.

Building the Program

If you choose to build the program, two information screens appear. The first screen describes the demonstration program. When you finish reading the first screen, press [ENTER] to continue. The second screen describes the Application Development Manager (AppDev). AppDev is the development tool of choice for building input programs. After you finish reading the second information screen, press [ENTER] to load AppDev.

For example, if you choose to build screen demo one, the following AppDev window appears:



Note

If you do not have AppDev, a special demo shell opens in the demonstration directory and a complete set of specification files is created for you. From this shell you can build the screen program manually.

If you are unfamiliar with AppDev, you may want to consult your Application Development Manager User Reference. Consulting the AppDev manual, however, is not necessarily required; there are a few basic AppDev functions that make building programs simple, such as the following:

- Opening a form specification file
- Running the *Screen* Code Generator
- Compiling the code
- Running the program

Opening a Form Specification File

Opening a form specification file is not required if you only want to build the program. It is useful, however, if you want to alter the program's default behavior. If you simply want to build the program, skip to "Running the Fitrix Screen Code Generator" on page 1-12.

As you can see from the graphic on the previous page, AppDev displays form specification files in the lower portion of the AppDev window. In screen demo one, there are two files, `browse` and `custfrm`.

To open a form specification file:

1. **Select the Open option from the File menu.**

   A submenu appears.

2. **Select Form Specification from the submenu.**

   A second submenu appears asking you which form specification file you want to open.

3. **Select the Form Specification file you want to open.**

For example, if you are running screen demo one, choose `custfrm`. Once you select the form specification file, AppDev runs the Form Painter using the file you specified.

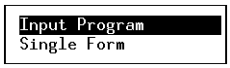
Running the Fitrix *Screen* Code Generator

The Fitrix *Screen* Code Generator builds 4GL code based off of instructions in form specification files. Since the screen demonstration programs start out with form specification files, you can run theFitrix *Screen* Code Generator directly.

To run the Fitrix *Screen* Code Generator:

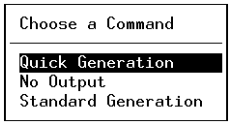
1. **Select Generate from the File menu.**

   A submenu appears asking you if you want to generate code for the entire program or for one form specification file.



2. **Select Input Program from the submenu.**

   A second submenu appears asking you which method of generation to use.



3. **Select Quick Generation.**

   This choice runs the Fitrix *Screen* Code Generator and creates all of the 4GL code necessary to compile the program.

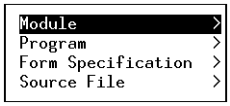
Compiling the Code

After you run the Fitrix *Screen* Code Generator, several more program files appear in lower portion of the AppDev window. To convert these files into an input program, you must run the compiler and link in the necessary library functions.

To compile the code:

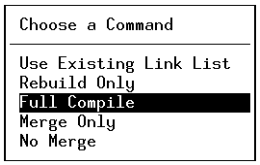
1. **Select Compile from the File menu.**

   A submenu appears asking you what type of compile to perform.



2. **Select Program from the submenu.**

   A second submenu appears requesting you to choose the compile mode.



3. **Select Full Compile from the second submenu.**

   This choice runs the compilation utility and builds a program file.

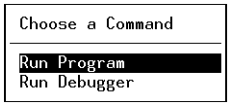
Running the Program

After compiling code, you can run the generated program.

To run the generated program:

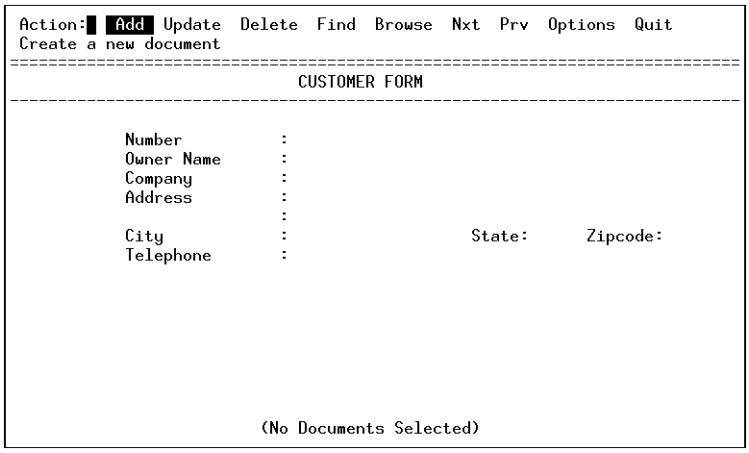
1. **Select Run from the File menu.**

A submenu appears asking you if you want to run the program directly or through the Informix Debugger.



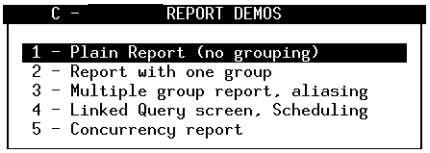
2. Select Run Program from the submenu.

This choice runs the generated program. For example, if you built screen demo one, the following program appears:



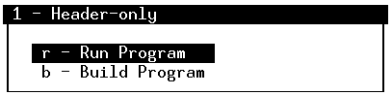
Running a Report Demo

The third choice on the <YOUR COMPANY NAME>CASE TOOLS DEMOS menu opens the REPORT DEMOS menu. This menu contains five report demonstration programs.



Each report demonstration shows a unique functionality. In fact, several of the report demonstration programs show new 4.12 Fitrix *Report* features, such as scheduling, aliasing, and concurrency.

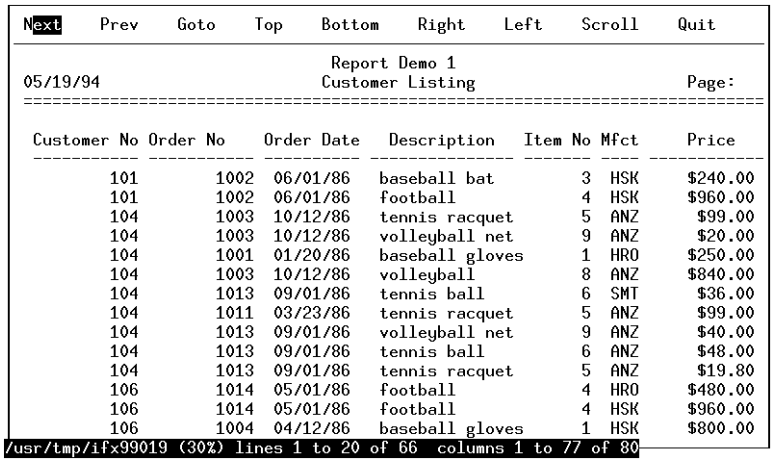
When you select a report demonstration program, you have the following two choices:



1. You can run the generated and compiled program directly.
2. You can build the program from its image (*.ifg) file and, if present, extension (*.ext) files.

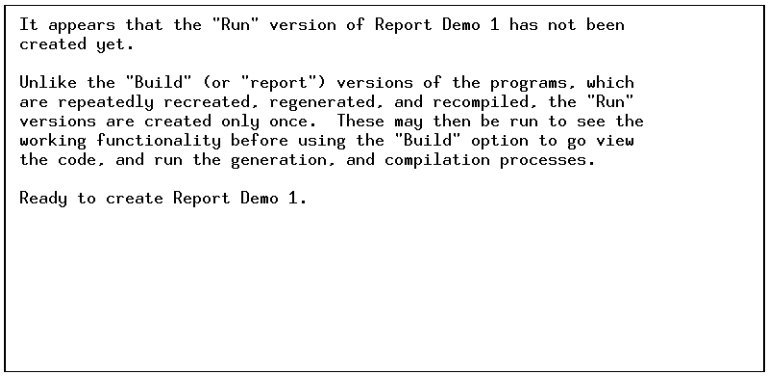
Running the Program

If you choose to run the program, an information screen appears that describes what the demonstration covers. After reading the information screen, press [ENTER] to continue and run the program. For example, if you run report demonstration one, the following program appears:



```
Next    Prev    Goto    Top    Bottom    Right    Left    Scroll    Quit

                          Report Demo 1
05/19/94                  Customer Listing                        Page:
================================================================================

 Customer No Order No    Order Date   Description   Item No Mfct     Price
 ----------- ---------- ---------- --------------- ------- ---- ------------
         101       1002  06/01/86   baseball bat          3  HSK     $240.00
         101       1002  06/01/86   football              4  HSK     $960.00
         104       1003  10/12/86   tennis racquet        5  ANZ      $99.00
         104       1003  10/12/86   volleyball net        9  ANZ      $20.00
         104       1001  01/20/86   baseball gloves       1  HRO     $250.00
         104       1003  10/12/86   volleyball            8  ANZ     $840.00
         104       1013  09/01/86   tennis ball           6  SMT      $36.00
         104       1011  03/23/86   tennis racquet        5  ANZ      $99.00
         104       1013  09/01/86   volleyball net        9  ANZ      $40.00
         104       1013  09/01/86   tennis ball           6  ANZ      $48.00
         104       1013  09/01/86   tennis racquet        5  ANZ      $19.80
         106       1014  05/01/86   football              4  HRO     $480.00
         106       1014  05/01/86   football              4  HSK     $960.00
         106       1004  04/12/86   baseball gloves       1  HSK     $800.00
/usr/tmp/ifx99019 (30%) lines 1 to 20 of 66  columns 1 to 77 of 80
```

You may encounter a case where the actual executable has yet to be created. If this occurs, you are given the choice to create it.



```
It appears that the "Run" version of Report Demo 1 has not been
created yet.

Unlike the "Build" (or "report") versions of the programs, which
are repeatedly recreated, regenerated, and recompiled, the "Run"
versions are created only once.  These may then be run to see the
working functionality before using the "Build" option to go view
the code, and run the generation, and compilation processes.

Ready to create Report Demo 1.
```

When you press [ENTER] to continue, the following line appears:

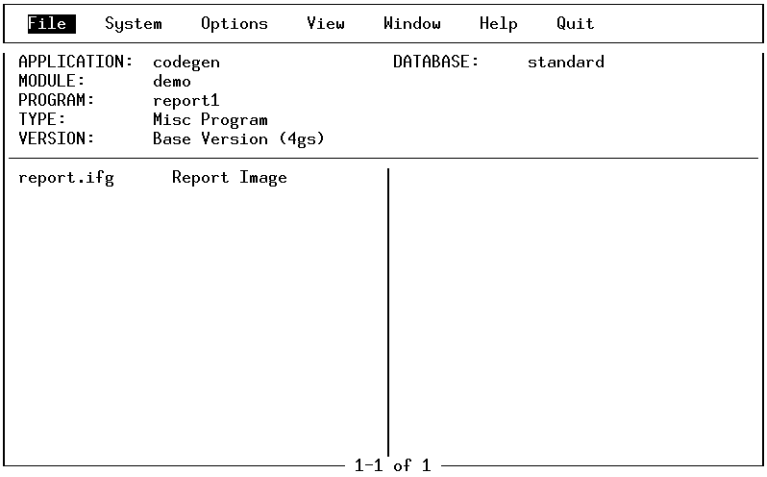
```
Enter [RETURN] for Demo 1, or [A] to prepare ALL Report Demos:
```

This prompt gives you the option of building only the selected program or building all report programs at once. Note that you should only encounter this circumstance once. After the executable is built, you will not have to build it again each time you select Run Program. Whichever option you select, the proper executable(s) is prepared.

Building the Program

If you choose to build the program, two information screens appear. The first screen describes the demonstration program. When you finish reading the first screen, press [ENTER] to continue. The second screen describes the Application Development Manager (AppDev). AppDev is the development tool of choice for building report programs. After you finish reading the second information screen, press [ENTER] to load AppDev.

For example, if you choose to build report demo one, the following AppDev window appears:



If you are unfamiliar with AppDev, you may want to consult your Application Development Manager User Reference. Consulting the AppDev manual, however, is not necessarily required; there are a few basic AppDev functions that make building programs simple, such as the following:

- Opening a report specification file
- Running the *Report* Code Generator
- Compiling the code
- Running the program

Opening a Report Specification File

Opening a report specification file is not required if you only want to build the program. It is useful, however, if you want to alter the program's default behavior. If you simply want to build the program, skip to "Running the Fitrix Report Code Generator" on page 1-19.

AppDev displays report specification files in the lower portion of the AppDev window. In report demo one, as with all report programs, there is a single report specification file: `report.ifg`.

To open a report specification file:

1. **Select the Open option from the File menu.**

   A submenu appears.

2. **Select Report Specification from the submenu.**

   A second submenu appears asking you whether you want to run the *Report* Writer or edit the report specification file directly.

Note

You cannot open `report.ifg` files contained in report demo programs with the Fitrix *Report* Writer. The Fitrix *Report* Writer can only work with a subset of reports that the Fitrix *Report* Code Generator is capable of handling.

3. **Select Edit Format file.**

After selecting Edit Format, the `report.ifg` file opens and you can edit it by hand.

Running the Fitrix *Report* Code Generator

The Fitrix *Report* Code Generator builds 4GL code based off of instructions in the report specification file. Since the report demonstration programs start out with an existing report specification file, you can run the Fitrix *Report* Code Generator directly.

To run the Fitrix *Report* Code Generator:

1. **Select Generate from the File menu.**

   The Fitrix *Report* Code Generator runs and multiple lines of code scroll across the screen.

2. **When the Generator finishes, press [ENTER] to return to AppDev.**

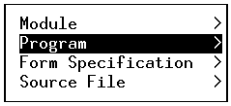
Compiling the Code

After you run the Fitrix *Report* Code Generator, several more program files appear in the lower portion of the AppDev window. To convert these files into a report program, you must run the compiler and link in the necessary library functions.

To compile the code:

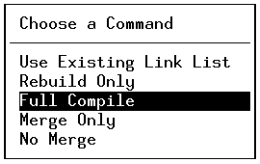
1. **Select Compile from the File menu.**

   A submenu appears asking you what type of compile to perform.



2. **Select Program from the submenu.**

A second submenu appears requesting you to choose the compile mode.



3. **Select Full Compile from the second submenu.**

   This choice runs the compilation utility and builds a program file.

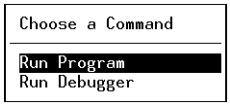
Running the Program

After compiling code, you can run the generated program.

To run the generated program:

1. **Select Run from the File menu.**

   A submenu appears asking you if you want to run the program directly or through the Informix Debugger.



2. **Select Run Program from the submenu.**

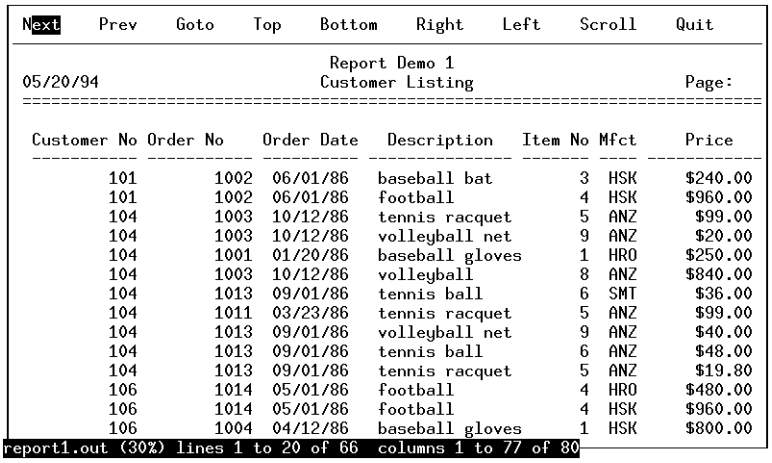
   The report runs and writes the report output into an *.out file.

3. **To see the results of the report, press [TAB] to move the highlight to App-Dev's file window.**

4. **Highlight the *.out file and press [ENTER].**

   AppDev automatically displays the report in the default pager for your system.

For example, if you built report demo one, the following report appears:



```
Next    Prev    Goto    Top    Bottom    Right    Left    Scroll    Quit

                          Report Demo 1
05/20/94                  Customer Listing                        Page:
================================================================================

 Customer No Order No    Order Date   Description   Item No Mfct     Price
 ----------- ----------- ---------- --------------- ------- ---- ------------
         101        1002  06/01/86   baseball bat          3  HSK     $240.00
         101        1002  06/01/86   football              4  HSK     $960.00
         104        1003  10/12/86   tennis racquet        5  ANZ      $99.00
         104        1003  10/12/86   volleyball net        9  ANZ      $20.00
         104        1001  01/20/86   baseball gloves       1  HRO     $250.00
         104        1003  10/12/86   volleyball            8  ANZ     $840.00
         104        1013  09/01/86   tennis ball           6  SMT      $36.00
         104        1011  03/23/86   tennis racquet        5  ANZ      $99.00
         104        1013  09/01/86   volleyball net        9  ANZ      $40.00
         104        1013  09/01/86   tennis ball           6  ANZ      $48.00
         104        1013  09/01/86   tennis racquet        5  ANZ      $19.80
         106        1014  05/01/86   football              4  HRO     $480.00
         106        1014  05/01/86   football              4  HSK     $960.00
         106        1004  04/12/86   baseball gloves       1  HSK     $800.00
report1.out (30%) lines 1 to 20 of 66  columns 1 to 77 of 80
```

Part Two

Screen New Features

2

Float Format

With the float format package you can customize the way floating point values appear on your Fitrix *Screen* and *Report* programs. This package extends the functionality of Informix's `DBFORMAT` variable. Using it, you can specify a wide range of attributes to tailor the way floating point values appear, such as a front and back symbol; a thousand separator; a decimal separator; and a positive and negative indicator. The float format package also lets you specify a precision value, which automatically rounds your floating point values.

This chapter covers the following topics:

- Overview
- Setting up a Float Format
- Float Format Functionality
- Applying Float Formats to Fitrix *Screens*
- Applying Float Formats to Fitrix *Reports*

Overview

Because floating point values are displayed differently from country to country, you may want to vary the way a floating point value looks in the applications you are developing. With the float format package, you can set up a number of float format definitions. For example, the following table shows formats for common monetary values:

Country	Positive Format	Negative Format
USA	$1,234.56	-$1,234.56
Italy	L1.234	-L1.234
Norway	kr1.234,56	kr1.234,56-
Portugal	1,234$56	-1,234$56

All float format definitions are stored in the `cgxffmtr` table. This table contains the following columns:

Column Name	Type	Description
float_format_code	char(10)	Holds a float format key value.
userdef	char(1)	Holds a Y value if row is user defined. If a Y is present, row gets preserved when dbmerge is run.
description	char(30)	Contains a short format description.
precision	smallint	Indicates how many places follow the decimal symbol. When necessary, float format rounds a value to match the specified precision.
thousand_separator	char(7)	Contains the thousand-separator symbol(s).
decimal_separator	char(7)	Holds decimal-separator symbol(s).

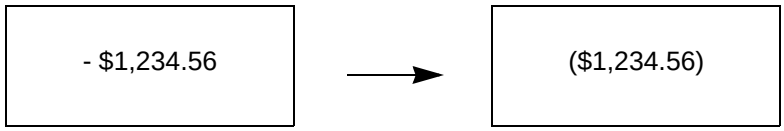
Column Name	Type	Description
front_symbol	char(7)	Contains the symbol that gets placed in front of the decimal value. Many times this is the monetary value symbol, such as a dollar sign.
front_minus	char(7)	Contains the front minus symbol(s).
front_plus	char(7)	Contains the front plus symbol(s).
back_symbol	char(7)	Contains the back symbol(s).
back_plus	char(7)	Contains the back plus symbol(s).
back_minus	char(7)	Contains the back minus symbol(s).

For example, to create the four monetary values shown previously, use the following float format definitions:

Country	float_format_code	userdef	description	precision	thousand_separator	decimal_separator	front_symbol	front_minus	front_plus	back_symbol	back_plus	back_minus
USA	USA			2	,	.	$	-				
Italy	ITL			0	.		L	-				
Norway	NOR			2	.	,	kr					-
Portugal	PRT			2	,	$		-				

Setting up a Float Format

To add a new float format definition, you must add a row to the `cgxffmtr` table. For example, you may want to make another version of the USA float format. Perhaps you want your new definition to use an left parenthesis as a `front_minus` value and a right parenthesis as a `back_minus` value.



Current Negative Format Desired Negative Format

To create this definition, add the following float format values to the `cgxffmtr` table:

float_format_code	userdef	description	precision	thousand_separator	decimal_separator	front_symbol	front_minus	front_plus	back_symbol	back_plus	back_minus
USANEW			2	,	.	$	(				)

For more information on adding rows to a table, refer to your Informix documentation set.

Float Format Functionality

Besides simply formatting the way floating point values appear on your programs, the float format package also rounds floating point values (similar to the way Informix rounds floating point values) and provides some "acceptable value" logic.

Rounding

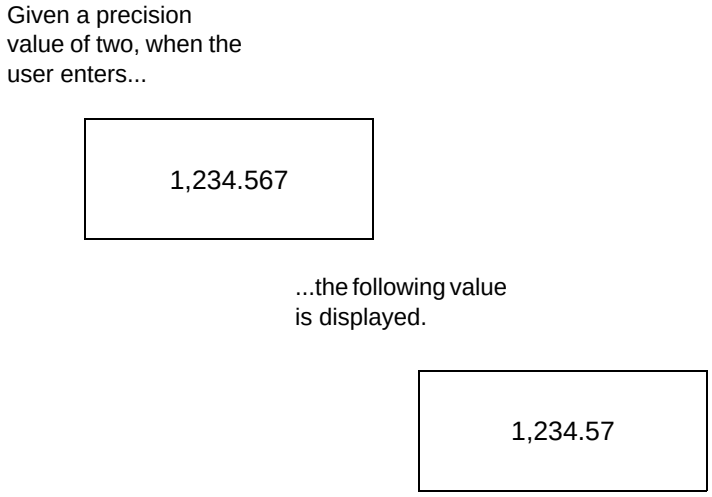
When you define a float format, you can set a precision value. By setting the precision value, you determine how many decimal places should follow a float field.

If you set the precision value to two, all float fields will evaluate to the hundredths position. Likewise, if precision is set to three, floating point values will evaluate to the thousands position.

If a user, however, enters a floating point value with more digits trailing the decimal than the precision value allows for, the float format package rounds the value to match the precision setting.

Example

Consider a field that has been defined to use a custom float format containing a precision value of two. When a program user enters a 1,234.567 in the field, the float format package rounds the value to 1,234.57.



Notes

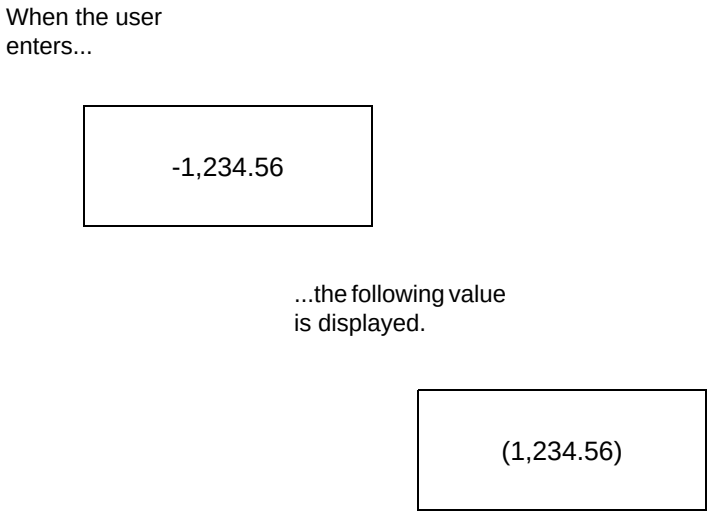
- The float format package is particularly valuable for fields representing columns defined as decimal with no scale (decimal(12)). These are floating decimals, but Informix screen I/O commands treat them as if they had a scale of 2 (decimal(12,2)). This inconsistency results in misleading screen displays and apparent rounding errors in the screen display of calculations. The float format package guarantees that what you see is what you want to see, and what you see is what you get.
- If you use the float format package with fixed decimal columns, make sure that the precision value you set matches the scale for that column. For example a decimal(12,2) column should use a float format definition that has precision set to two.

Acceptable Values

Frequently, you may want to allow program users to enter several different characters that evaluate to a single display character. This type of logic is very common in fields representing date columns. The user can enter 01-01-1994 and the value will be displayed as 01/01/94. In this case, the dash (-) character is interpreted properly and converted to a slash (/).

The float format package lets you do much of the same with float fields. This ability is known as "acceptable values." In other words, you can create formats that don’t impose a strict syntax for the user to remember.

Consider another example. Quite often, negative values are displayed in parentheses. Your user, however, may enter negative values with a minus symbol. You can set up the float format package to recognize a minus and display it as a left parenthesis.



Refer back to the `cgxffmtr` table on page 2-4. Notice that several columns are of type char(7), such as the front_minus column. All of the char(7) type columns can contain acceptable value characters except for the front_symbol and back_symbol columns.

Example

If you want the front_minus to accept both a minus sign and a left parenthesis, define the front_minus and back_minus columns as follows:

Column Name	Value	Description
front_minus	(-	Puts negative values into parentheses, but accepts both a minus sign and a left parenthesis.
back_minus	)	Puts a right parenthesis on the end of a negative number.

When the user enters a negative value preceded by a minus sign, the value is accepted and reformatted to display within parentheses.

Notes

- You can specify up to seven acceptable values for the following columns:

  thousand_separator
  decimal_separator
  front_minus
  front_plus
  back_minus
  back_plus

- You cannot use the same character in more than one definition. For example, you cannot have a comma (,) as both a decimal_separator and a thousand_separator.

- In each of these columns, only the first symbol is displayed. All the other symbols become the "acceptable value" symbols.

Applying Float Formats to Screens

Once you define a float format, you can specify the field(s) it applies to. This specification takes place within your form specification (*.per) file.

The float_fmt Line

You can apply a float format definition to any float or decimal field. To do so, you must add the `float_fmt` line to your *.per file.

Syntax

The `float_fmt` line is placed in either the input 1 or input 2 section of a .per file. You can pass the `float_format_code` value from the `cgxffmtr` table or a p_ record value. Use the following syntax:

`float_fmt = field=`*`field`*`, format_key="`*`float_format_code`*`"`

`float_fmt = field=`*`field`*`, format_key=p_`*`variable`

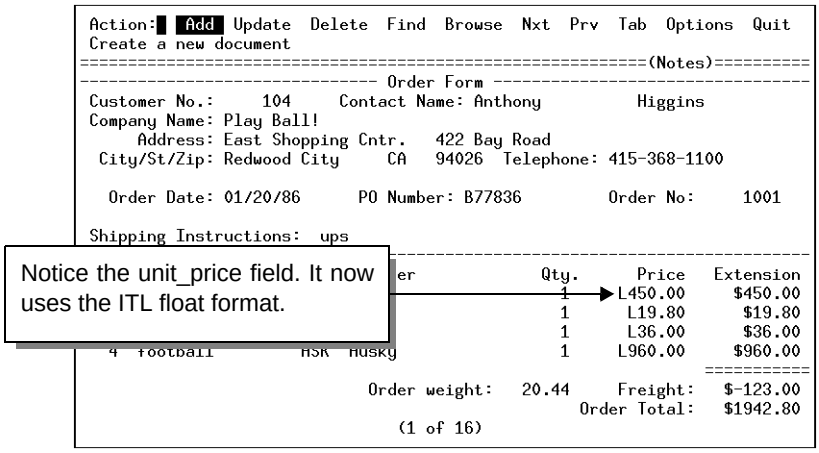
Example

This example, applied to `scr_demo 3`, customizes the `unit_price` field to use `ITL`, the Italian float format definition.

```
input 2
    table   = items
    join    = items.order_num = orders.order_num
    order   = item_num
    arr_max = 100
    autonum = item_num
    math    = total_price = quantity * unit_price
    lookup  = name=stock_num, key=stock_num, table=stock,
              filter=stock_num = $stock_num, into=description
    lookup  = name=stock_manu, key=manu_code, table=stock,
              filter=stock_num = $stock_num and manu_code = $manu_code,
              into=unit_price
    lookup  = key=manu_code, table=manufact, filter=manu_code = $manu_code
    zoom    = key=stock_num, screen=stockzm, table=stock, noautozoom
    zoom    = key=manu_code, screen=stk_mnu, table=
              filter=stock.stock_num = $stock_num
  float_fmt = field=unit_price, format_key="ITL"
```

Float Format Line

The resulting program looks as follows:



In most cases, you would want to convert every decimal field to reflect the same float format definition. In this example, however, only one field uses the ITL (Italian) definition. The other fields receive the default definition, which is defined by the `DBFORMAT` variable.

Notes

Although the float format package expands your ability to create language independent code, there are a few caveats you should consider:

- You cannot use the Form Painter to apply float format definitions to fields. You must use a text editor and add each `float_fmt` line by hand. The Form Painter, however, preserves `float_fmt` lines. So if you have modify a .per file to include float format logic, you can still open and update that file using the Form Painter.
- If you have created math logic based on a p_ record value, you must change your code so that this math logic is performed in the q_ record.
- If you have the same decimal field on a header screen and a browse screen, you must add a `float_fmt` line to both *.per files.

- If you use an invalid float format definition, decimal fields are formatted according to the `DBFORMAT` variable, which has a default precision of 2.

Float Format Logic

For every `float_fmt` line you add, the Fitrix *Screen* Code Generator creates multiple lines of decimal format logic. This logic is added to the *.4gl file that corresponds with the .per file.

Example

Building on the previous example, the Fitrix *Screen* Code Generator creates the following code for the `unit_price` field. This code is added to the `lld_display()` function in `detail.4gl`:

```
#_float_formatonly - Format the decimal
    #_fl_code_unit_price
    let fl_code = "ITL"
    #_fl_length_unit_price
    let fl_length = 10
    #_fl_attr_unit_price
    let fl_attr = ""
    #_fl_float_unit_price
    call fmt_only(
        q_items[m + n].unit_price, fl_code, fl_length, fl_attr)
              returning p_items[m + n].unit_price

        display p_items[m + n].* to s_items[n].* attribute(red)
```

As you can see, the Fitrix *Screen* Code Generator surrounds the float format logic with several block tags. These tags give you a point in the code where you can add your own custom logic via extension (*.ext) and trigger (*.trg) files.

Applying Float Formats to Reports

You can apply a float format to a Fitrix *Report* program much like you do to a Fitrix *Screen* program. When used with reports, the float format logic customizes the way a column appears on your report output. For each column you want to apply a float format to, you must create a `float_fmt` line in the `report.ifg` file.

The float_fmt Line

To format a report column, you place the `float_fmt` line in the select section of the `report.ifg` file. This line instructs the Fitrix *Report* Generator to apply the float format definition you specify to the report column.

Syntax

The `float_fmt` line is placed in the select section of the `report.ifg` file. You can either pass it the `float_format_code` value from the `cgxffmtr` table. The `float_fmt` line uses the following syntax:

`float_fmt = field=`*`table.column`*`, format_key="`*`float_format_code`*`"`

Example

This example applies the ITL float format to the `items.total_price` column.

```
select
    name   = Demo Select
    tables = customer, orders, items, stock
    join   =  stock.manu_code = items.manu_code and stock.stock_num = items.stoc
k_num and orders.customer_num = customer.customer_num and items.order_num = orde
rs.order_num
    order  = customer.customer_num
 float_fmt = field=items.total.price, format_key="ITL"

end
```

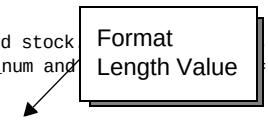
ITL Float Format Line

After adding the `float_fmt` line, you can build your report program. When you run it, the float format that you specified will appear on your report. In this example, the `total_price` column is using the ITL float format definition:

```
Next    Prev    Goto    Top    Bottom    Right    Left    Scroll    Quit

                        Customer Listing                    Page:          1
=========================================================================

         101
er No Order No    Order Date   Description   Item No Mfct     Price
----- ---------- ---------- --------------- ------- ---- ------------
  101        1002  06/01/86  football              4 HSK          L960.00
  101        1002  06/01/86  baseball bat          3 HSK          L240.00
                                 Subtotals for         101
                                                             ------------
                                                            L1,200.00

         104
er No Order No    Order Date   Description   Item No Mfct     Price
----- ---------- ---------- --------------- ------- ---- ------------
  104        1001  01/20/86  tennis ball           6 SMT           L36.00
  104        1001  01/20/86  basketball            7 HRO          L600.00
  104        1001  01/20/86  tennis racquet        5 ANZ           L19.80
  104        1001  01/20/86  baseball gloves       1 SMT          L450.00
  104        1001  01/20/86  football              4 HSK          L960.00
  104        1003  10/12/86  tennis racquet        5 ANZ           L99.00
report.out (15%) lines 1 to 21 of 132  columns 9 to 85 of 85
```

As you can see, float format logic for report programs doesn't align the formatted column correctly. To fix this, you can add a `format_length` indicator to the float format line in your `report.ifg` file.

```
select
    name   = Demo Select
    tables = customer, orders, items, stock
    join   =  stock.manu_code = items.manu_code and stock.stoc
k_num and orders.customer_num = customer.customer_num and orde
rs.order_num
    order  = customer.customer_num
 float_fmt = field=items.total.price, format_length=12, format_key="ITL"

end
```



When you set the `format_length` value to the appropriate length, the formatted column is aligned correctly.

```
Next    Prev    Goto    Top    Bottom    Right    Left    Scroll    Quit
                          Customer Listing                  Page:          1
=========================================================================

         101
er No Order No    Order Date   Description   Item No Mfct     Price
----- ---------- ---------- --------------- ------- ---- ------------
  101        1002  06/01/86  football              4 HSK       L960.00
  101        1002  06/01/86  baseball bat          3 HSK       L240.00
                                 Subtotals for          101
                                                           ------------
                                                          L1,200.00

         104
er No Order No    Order Date   Description   Item No Mfct     Price
----- ---------- ---------- --------------- ------- ---- ------------
  104        1001  01/20/86  tennis ball           6 SMT        L36.00
  104        1001  01/20/86  basketball            7 HRO       L600.00
  104        1001  01/20/86  tennis racquet        5 ANZ        L19.80
  104        1001  01/20/86  baseball gloves       1 SMT       L450.00
  104        1001  01/20/86  football              4 HSK       L960.00
  104        1003  10/12/86  tennis racquet        5 ANZ        L99.00
report.out (15%) lines 1 to 21 of 132  columns 9 to 85 of 85
```

3

Translating Y/N Fields

A powerful feature of the Fitrix *CASE* Tools is the ability to create programs that can be translated into other languages. With the 4.12 Fitrix *Screen* Code Generator, you can streamline your efforts to create translatable programs.

This section covers the following topics:

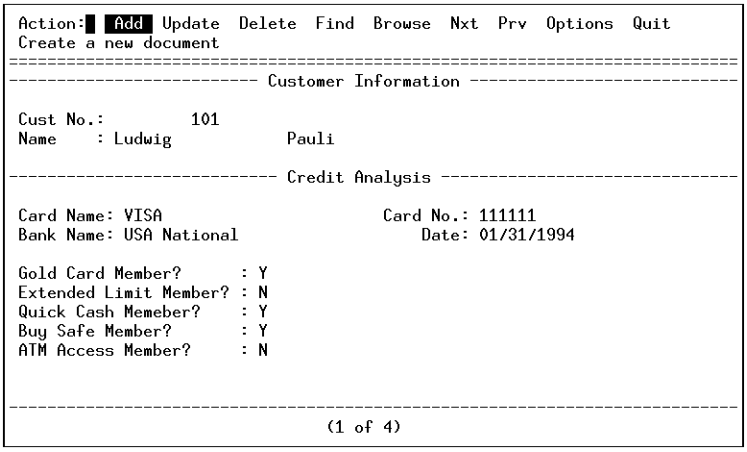
- Overview
- Applying Y/N Logic to Screens
- Applying Y/N Logic to Reports

Overview

Creating translatable programs can take some time. A big part of the process involves populating the `stxlangr` table with rows of translation strings. Many times these strings are very similar, as in the case of Y/N fields. Most Y/N fields are the same size and accept the same values.

In the past, to make a Y/N field translatable, you had to create two rows in the `stxlangr` table: one row for the Y value and one row for the N value. If your application had 10 Y/N fields, you had to create 20 rows. The 4.12 Fitrix *Screen* Code Generator simplifies this task. Instead of defining each Y/N field individually, you can define them all at once.

Consider the following form, which contains five Y/N fields:



With the 4.11 Fitrix *Screen* Generator, the `stxlangr` table would look as follows:

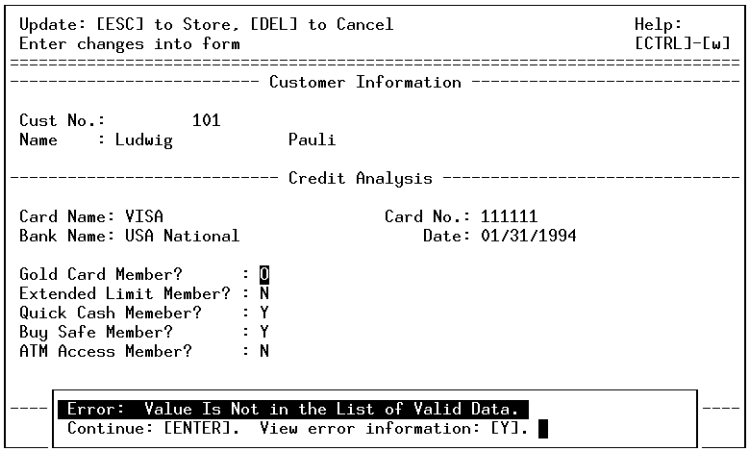
```
ENG|credit_card.gold_card|ALL|N|N|
ENG|credit_card.gold_card|ALL|Y|Y|
ENG|credit_card.ext_limit|ALL|N|N|
ENG|credit_card.ext_limit|ALL|Y|Y|
ENG|credit_card.quick_cash|ALL|N|N|
ENG|credit_card.quick_cash|ALL|Y|Y|
ENG|credit_card.buy_safe|ALL|N|N|
```

```
ENG|credit_card.buy_safe|ALL|Y|Y|
ENG|credit_card.atm_access|ALL|N|N|
ENG|credit_card.atm_access|ALL|Y|Y|
```

With the new 4.12 Fitrix *Screen* Code Generator, only two rows are required.

```
ENG|YES.NO|ALL|Y|Y|
ENG|YES.NO|ALL|N|N|
```

Besides making fields easier to translate, this new logic also automatically validates your Y/N fields. For example, if the user places a O in the field instead of a Y, the program reports an error.



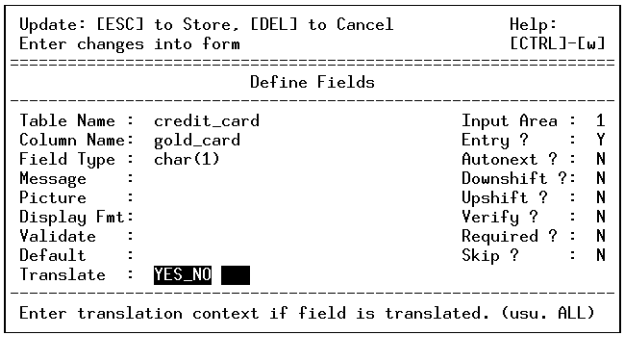
Applying Y/N Logic to Screens

You can apply the new Y/N logic using the Form Painter or directly to your *.per file with a text editor.

Using the Form Painter

The most automatic way to apply this new functionality is using the Form Painter. Begin by starting the Form Painter and loading your form. Next, select the Y/N field you want to use and press [CTRL]-[z].

In the Define Field window, place `YES_NO` in the Translate field:



Save your form, regenerate 4GL, and remake your program.

Note

Although you place `YES_NO` in the Translate field, it is not really a translation context. This value acts as a switch and uses the `ALL` translation context.

Using a Text Editor

Open your *.per file using vi or some other text editor. In the appropriate section (in this case input 1), insert the following line:

```
input 1
...
translate = field_name YES_NO
```

For example, if you want to translate the `gold_card` field, your input 1 section would look as follows:

```
input 1
    table     = credit_card
    key       = card_number
    filter    = 1=1
    lookup    = name=custlk, key=customer_num, table=customer,
      into=fname, into=lname,
      filter=customer.customer_num = $customer_num
    zoom      = key=customer_num, screen=custzm, table=customer,
      from=customer_num
    translate = gold_card YES_NO
```

As you can see, the translate line contains the field name and the YES_NO switch.

If you have multiple Y/N fields, you should add one translate line for each field:

```
input 1
    table     = credit_card
    key       = card_number
    filter    = 1=1
    lookup    = name=custlk, key=customer_num, table=customer,
      into=fname, into=lname,
      filter=customer.customer_num = $customer_num
    zoom      = key=customer_num, screen=custzm, table=customer,
      from=customer_num
    translate = gold_card YES_NO
    translate = ext_limit YES_NO
    translate = quick_cash YES_NO
    translate = buy_safe YES_NO
    translate = atm_access YES_NO
```

Applying Y/N Logic to Reports

You can also apply Y/N logic to reports. For example, you may have a one character column in your database that contains either Y values or N values. By setting up two new `stxlangr` records, you can change the way these values appear on your reports.

Syntax

To translate a Y/N report column, you must add a new section to your `report.ifg` file called language. This section uses the following syntax:

```
language
    translate = table.column YES_NO
end
```

Typically, the language section follows the `select` section within the `report.ifg` file. If you wanted to translate multiple fields, the syntax is as follows:

```
language
    translate = table.column YES_NO
    translate = table.column2 YES_NO
    ...
end
```

Example

Suppose you want a Y/N column to appear in its Italian equivalent on a report. You could define the following records in `stxlangr`:

```
|ITL|YES.NO|ALL|Y|S|
|ITL|YES.NO|ALL|N|N|
```

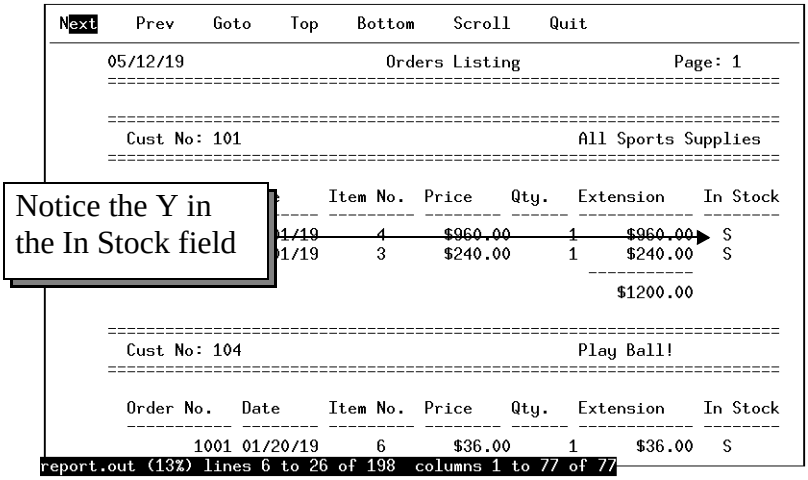
In your `report.ifg` file, set up a new language section:

```
language
    translate = stock.in_stock YES_NO
end
```

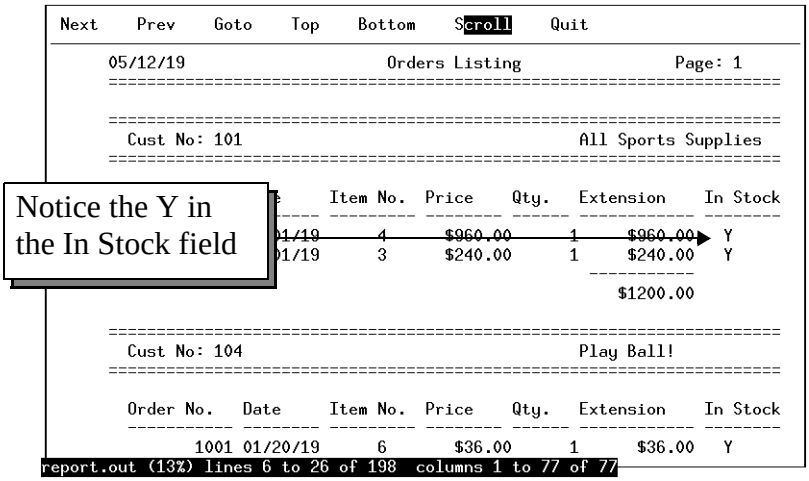
Generate and compile your report, then at runtime, specify:

```
fglgo *4gi -l ITL
```

Your report runs and you see “S’s” in place of “Y’s” in the in_stock column.



When you run fglgo *4gi without the -l ITL language flag:



4

Screen Hooking Logic

The 4.11 version of the Fitrix *CASE* Tools provided a new function to handle screen switching logic. This function, called `socketManager()`, established a new method for hooking different screen types to Fitrix *Screen* and *Report* programs.

The `socketManager()` function created a standard method for hooking screens into an input program. However calling `socketManager()` in a report program pulled in a number of unnecessary functions. To reduce this overhead, a new library file called `sktSwtch.4gl` and a new trigger called `socket_items` were created.

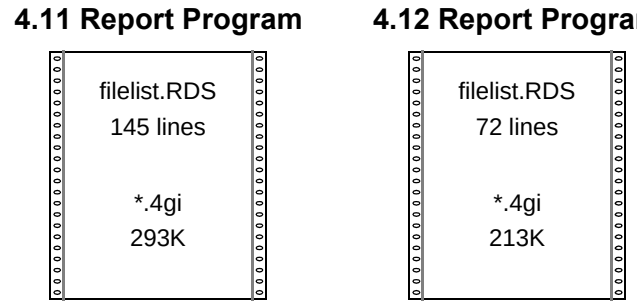
This chapter covers the following topics:

- Overview
- The `socket_items` Trigger

Overview

To understand the need for a new function and trigger, consider the case of a simple report program that includes a query screen. Before the report is run, the query screen appears allowing users to enter selection criteria for the report.

A query screen is hooked to a report program via the `socketManager()` function. This function links in all of the screen handling logic necessary to run the query screen. The `socketManager()` function, however, does not stop there. It also pulls in screen handling logic for other screen types, none of which are being used in the report program. A simple report program can end up being much larger than necessary.



Using new screen-hooking logic, you can reduce the size of your programs.

To simplify the amount of code linked in by the `socketManager()` function, `socketManager()`'s screen switching mechanism was extracted and placed in the new `sktSwtch.4gl` file. The functions in this file evaluate the screen type, which you specify using the new `socket_items` trigger. Once the screen type is known, only the appropriate functions for that screen type are linked in.

The socket_items Trigger

If you want, you can look at how the functions in `sktSwtch.4gl` work using the Informix debugger or some other method. This file is located in `$fg/lib/scr.4gs`. For the most part, however, you can ignore what goes on "behind the scenes." Most important to you is understanding how to incorporate the new screen-hooking logic into your programs.

A Quick Review

If you are unfamiliar with hooking in screens to your programs, you might want to consult the 4.11 *Screen* Technical Reference. For those who are familiar with this process, a quick review is in order.

In all, the Fitrix *Screen* Code Generator recognizes nine different screen types. These screen types consist of main screens, known as header and header/detail screens. For each program, you create a single main screen. In addition to main screens, there are secondary screens that you hook to your main screen. These secondary screens include zoom, browse, add-on header, add-on detail, extension, view header, and view detail.

Of all the secondary screens, browse screens are hooked in automatically and zoom screens and hooked in via the Form Painter. All other screen types, however, must be manually hooked to your program via an extension or trigger file. In general, you use the following steps to build and hook in secondary screens:

1. Build the screen image using either the Form Painter or a text editor to create the form specification (*.per) file.
2. Specify the program condition that initiates the secondary screen and create 4GL logic in an extension or trigger file that evaluates for this condition.
3. When the above condition is met, place a call to the `socketManager()` function specifying the screen name, type, and flow.
4. Add the screen and function names to the `switchbox_items` trigger in either the default section of your trigger file or apply it to `main.4gl` in the extension file.

Adding the socket_items Trigger

The `socket_items` trigger specifies the screen type of the screen you are hooking in. For every screen identified in the `switchbox_items` trigger, you should create a corresponding `socket_items` trigger. Both are placed in the default section of a trigger file or, if more appropriate, they can be applied to `main.4gl` in an extension file.

The `socket_items` trigger cannot be used alone; it must accompany the `switchbox_items` trigger:

Syntax

Use the following syntax to add the `socket_items` trigger to a trigger or extension file:

```
default

switchbox_items
    scr_name function
    [scr_name function...];

socket_items
    scr_type
    [scr_type...];
```

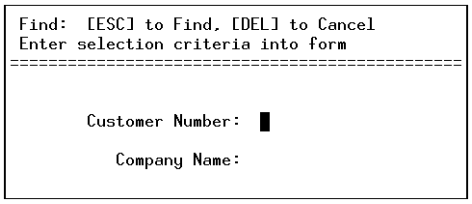
Use the following syntax if you use `socket_items` in an extension file:

```
start file "main.4gl"

switchbox_items
    scr_name function
    [scr_name function...];

socket_items
    scr_type
    [scr_type...];
```

Example

Consider again the simple report program mentioned previously. If you are familiar with the Fitrix *CASE* Tools demonstration programs, you may also want to start `rpt_demo 4`.

This is a simple report program built from the customer table, which is part of your demonstration data. Prior to running, this report program displays a query screen that lets you build the selection criteria for the report:



This query screen is hooked by the following extension file:

```
#---------------------------------------------------------------------
# add the switchbox and socket items to main
#---------------------------------------------------------------------
switchbox_items
    query S_query;
socket_items
    query;

#---------------------------------------------------------------------
# add the scr library to the Makefile
#---------------------------------------------------------------------
start file "Makefile"
    libraries
        $(fg)/lib/scr.a;

#---------------------------------------------------------------------
# add some working counter variables to midlevel.
#---------------------------------------------------------------------
start file "midlevel.4gl"
    function define ml_filter
        m smallint,
        n smallint;

#---------------------------------------------------------------------
# call the query function and get the filter clause back.
#---------------------------------------------------------------------
after block ml_filter sel_filter
    while true
        call socketManager("query", "query", "default")
        let n = fgStack_pop()
        # n = 0 means <DEL> was hit - nothing returned
        if n = 0
        then
```

```
            let int_flag = true
            call ct_int_exit()
            # if we return from ct_int_exit, that means the user wanted to continue
            continue while
        else
            let sel_filter = sel_filter clipped, " and ("
            for m = 1 to n
                let sel_filter = sel_filter clipped, fgStack_pop()
            end for
            let sel_filter = sel_filter clipped, ")"
            exit while
        end if
    end while ;
```

Note that the name of the screen in this case is query. So query is both the screen name and screen type.

Once you add this extension file to the `base.set` file and run `fg.make` to merge and compile the code, screen handling logic is linked into your program.

Notes

Although the `socket_items` trigger eliminates a number of unnecessary function calls, it is not a required trigger as is the `switchbox_items` trigger. You can continue to hook in screens using the 4.11 method. If you do decide to use the `socket_items` trigger, keep the following items in mind.

- In all, the `socket_items` trigger recognizes nine different item types:

  zoom
  query
  add-on header
  add-on detail
  extension
  view header
  view detail
  single_function
  custom

- The `single_function` item, lets you pass a function to the switchbox.
- The custom item lets you build custom screen types and link in your own screen handling logic.
- The Form Painter also supports the `socket_items` trigger.

Part Three

Report New Features

5

Report Scheduling

Scheduling allows you to preset a desired runtime, and thus can be used to free system resources during peak hours.

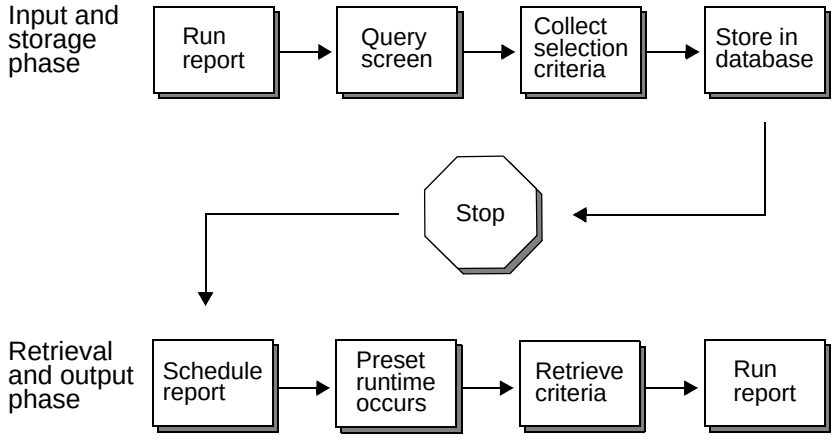
This section covers the following topics:

- Overview
- Implementation
- Scheduling Code in the Library
- Generated Code in `midlevel.4gl`

Report Scheduling Overview

Scheduling is a tool that allows you to predetermine when report output will be generated. The normal process for generating a `report.out` file is as follows: create a `report.ifg` file, generate code, compile code, and run. With scheduling, you can delay program execution by specifying a desired runtime.

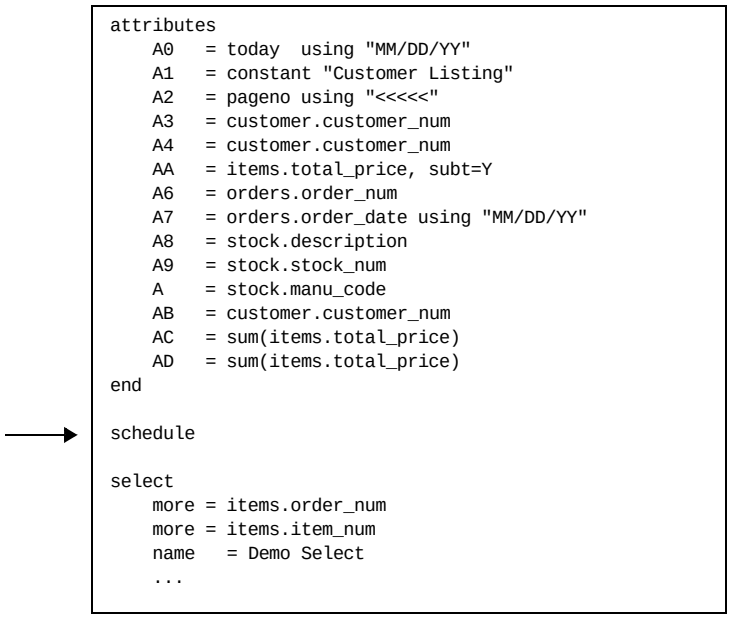
The scheduling process is shown in the following diagram:



Report scheduling is most useful when report generation would be a burden to your system. Since some reports may be highly system-intensive, a better time for report generation may be after peak hours. Report scheduling is the tool by which to accomplish this.

Implementation

To implement the scheduling feature, simply enter the word "schedule" in your `report.ifg` file. You may type `schedule` anywhere in the file, provided it's only placed between other sections. The following piece of code from a sample `report.ifg` file shows how this may be done:



```
attributes
    A0   = today  using "MM/DD/YY"
    A1   = constant "Customer Listing"
    A2   = pageno using "<<<<<"
    A3   = customer.customer_num
    A4   = customer.customer_num
    AA   = items.total_price, subt=Y
    A6   = orders.order_num
    A7   = orders.order_date using "MM/DD/YY"
    A8   = stock.description
    A9   = stock.stock_num
    A    = stock.manu_code
    AB   = customer.customer_num
    AC   = sum(items.total_price)
    AD   = sum(items.total_price)
end

schedule

select
    more = items.order_num
    more = items.item_num
    name   = Demo Select
    ...
```

Three New Functions

When scheduling is implemented, the Report Generator will create three new functions in the `midlevel.4gl` file:

`function ml_schedule()`

The function `ml_schedule` has two primary responsibilities: it gathers and stores selection criteria for the report, and later retrieves the stored criteria and prepares the report.

First `ml_schedule` gathers selection criteria by calling the `ml_filter` function (discussed in next section), also located in `midlevel.4gl`. Then `ml_schedule` calls the `ml_put_filter` function to store this criteria into the `stxfiltr` table. At this point, program execution stops. This concludes the storage phase.

When it is time to retrieve selection criteria back from the `stxfiltr` table, `ml_schedule` calls the function `ml_get_filter` to accomplish this task. Once criteria has been retrieved, the report is ready to run and output is generated.

`function ml_put_filter(job_id)`

Stores the selection criteria into the `stxfiltr` table. This way, selection criteria can be used again when the program is run at a later time.

`function ml_get_filter(job_id)`

Retrieves selection filter information stored in `stxfiltr` when the report is running in background mode.

Incorporating Selection Criteria

Selection criteria screens are often used for the manual input of criteria at runtime. In reports that do not have some type of query screen, selection criteria is found in `ml_filter`, which contains the filter criteria provided by the `report.ifg` file at the time that code was generated. Any code that calls a criteria screen should be placed within this function.

For information on using scheduling with Menus, refer to the *Menus User Reference*.

Scheduling Example

After generating code and compiling, you will need to store selection filter information to `stxfiltr`. Type the following line at the UNIX prompt to accomplish this:

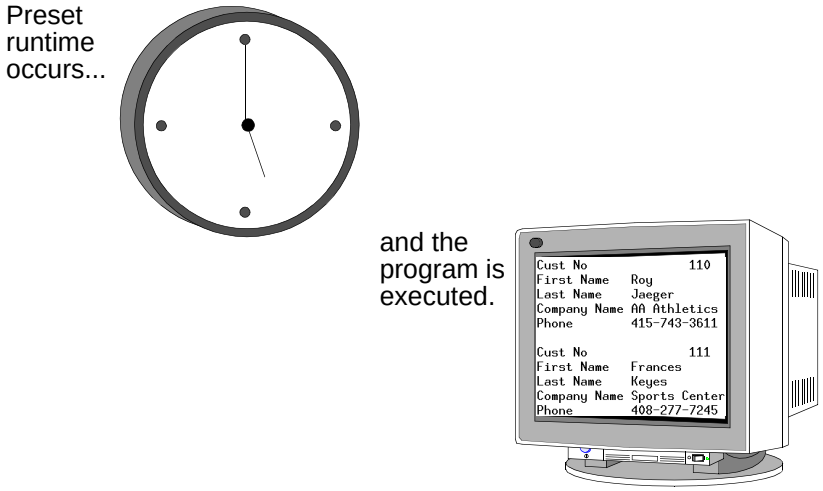
`fglgo /path/filename.4gi -s job_id_code`

The -s flag specifies that selection filter information will be stored in `stxfiltr`, and program execution doesn't occur at this time. You may select any job_id_code you wish, provided that it is less than 16 characters in length. Remember the job_id_code, as you will need to use it again when passing the `-b` flag to execute the program.

When it is time to run your report, selection filter information must be retrieved back from `stxfiltr`. To do this, try using the UNIX `echo` and `at` commands:

```
echo "fglgo /path/filename.4gi -b job_id_code" | at time
```

When the scheduled report is run at the chosen time, the `-b` flag is passed along with the same job_id_code that was used during the storage phase of the report. The result will be program execution, and your newly generated report.out file (if you've chosen this as your destination).



Scheduling Code in the Library

In the report library, scheduling code gets called from flow control.

flow_control

The `flow_control` function, in `$fg/lib/report.4gs/flow.4gl`, is as follows:

```
if ct.sel_filter is null
then
    if not ml_schedule()
    then
        call ml_filter()
    end if
end if
```

If scheduling has been added to your program, the generated `ml_schedule` returns a value of true. If scheduling has not been added to your program, then a stub function (discussed next) links into your program and a value of false is returned. This causes the call to `ml_filter` to get executed and the program runs as it would without scheduling being implemented.

stub function

In `$fg/lib/stubs.4gs/ml_sched.4gl`, a stub function has been added for `ml_schedule`

```
##########################################################################
function ml_schedule()
# returning false
##########################################################################
# This is a stub function for ml_schedule(). It will always return
# false. It will cause ml_filter() to be called in the normal way
# for those report programs that do not use the new Generated Report
# Scheduling provided by the locally generated ml_schedule().
#
 #_define_var - Define local variables
 #_ret - Returning value
 return false
end function
# ml_schedule()
```

Generated Code in midlevel.4gl

As explained in the Three New Functions section, three functions have been added to the `midlevel.4gl` file. Each function and its included code is listed below.

ml_schedule

```
########################################################################
function ml_schedule()
########################################################################

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        sel_flag smallint,   # (boolean) Running in 'select only' mode?
        bg_flag smallint,    # (boolean) Running in the background?
        job_id like stxfiltr.unique_id

    #_init - Initialize
    let sel_flag = false
    let bg_flag = false

    #_set_select - Set job_id if running in select only mode
    let job_id = get_argument("-s")

    #_check_for_mode - Not null job_id means running in select only mode
    if job_id is not null
    then
        #_set_sel_flag - Set sel_flag for select only mode run
        let sel_flag = true
    else
        #_set_background - Set job_id if running in background
        let job_id = get_argument("-b")

        #_check_background - Not null job_id means running in background
        if job_id is not null
        then
            #_set_bg_flag - Set bg_flag for background run
            let bg_flag = true
        end if

    end if

    #_check_load - Load the data from the stxfiltr table or from the
    # input screen depending on if we're running in the background.
    if bg_flag
    then
        #_get_filter - Load filter data from stxfiltr table
```

```
        call ml_get_filter(job_id)

        #_no_error_prompt - Set no error prompting for background run
        call err_hand_prompt_off()
    else
        #_else_load - Call ml_filter() to build selection filter
        call ml_filter()
 end if

    #_check_save - Save the selection criteria to the sel_filter or
    # to the disk if running in select only mode.
    if sel_flag
    then
        #_put_filter - Save filter data in stxfiltr table
        call ml_put_filter(job_id)

        #_exit_program - Exit program after saving filter data
        call exit_program(0)
    end if

    #_ret - Returning value
    return true

end function
# ml_schedule()
```

ml_put_filter

```
##############################################################################
function ml_put_filter(job_id)
##############################################################################
# This function saves the selection criteria out to the stxfiltr
# table for running the program at a later time.
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        job_id like stxfiltr.unique_id,
        tmpStr char(200),   # Working string
        filt_len smallint,  # Length of selection filter (sel_filter)
        n smallint          # Working number

    #_init - Initialize
    let filt_len = length(sel_filter)

    #_cleanup - Delete any existing data in stxfiltr
    delete from stxfiltr where stxfiltr.unique_id = job_id

    #_save_data - Store selection filter data into stxfiltr
    for n = 1 to filt_len step 200
```

```
        #_set_data - Grab 200 characters of filter at a time
        let tmpStr = sel_filter[n, n+199]

        #_insert_data - Insert data into table
        insert into stxfiltr values (job_id, n, tmpStr)
    end for

end function
# ml_put_filter()
```

ml_get_filter

```
########################################################################
function ml_get_filter(job_id)
########################################################################
# This function loads selection filter information stored in stxfiltr
# when the report is run in background mode.
#
    #_define_var - Define local variables
    define
        #_local_var - Local variables
        job_id like stxfiltr.unique_id,
        tmpStr char(200),   # Working string
        n smallint          # Generic number

    #_init - Initialize

    #_build_curs - Build the cursor on the filter table
    let tmpStr =
        "select ",
            "seq_no, ",
            "sel_filter ",
          "from stxfiltr ",
          "where stxfiltr.unique_id = ? ",
          "order by seq_no "
   #_prep_curs - Prepare the string for execution
    prepare filt_prep from tmpStr

    #_declare_curs - Declare cursor from the string
    declare sel_curs cursor for filt_prep

    #_open_curs - Open cursor for retrieving data from stxfiltr
    open sel_curs using job_id

    #_read_data - Read the rows from the filter table
    while true
        #_fetch - Fetch the data
        fetch next sel_curs into n, tmpStr

        #_notfound - No more rows found
        if sqlca.sqlcode = NOTFOUND then exit while end if
```

```
        #_build_filter - Build selection filter with retrieved data
        let sel_filter[n,n+199] = tmpStr
    end while

    #_close_curs - Close the cursor
    #_cleanup - Delete stxfiltr rows
    delete from stxfiltr where stxfiltr.unique_id = job_id

end function
# ml_get_filter()
```

6

Column Aliasing

Previous versions of the Fitrix *Report* Code Generator required you to use unique column names in your report specification (*.ifg) files. Using the 4.12 Fitrix *Report* Code Generator, unique column names are no longer necessary. If you want to use two columns from different tables that have the same name, you can give one column a unique alias in the `report.ifg` file.

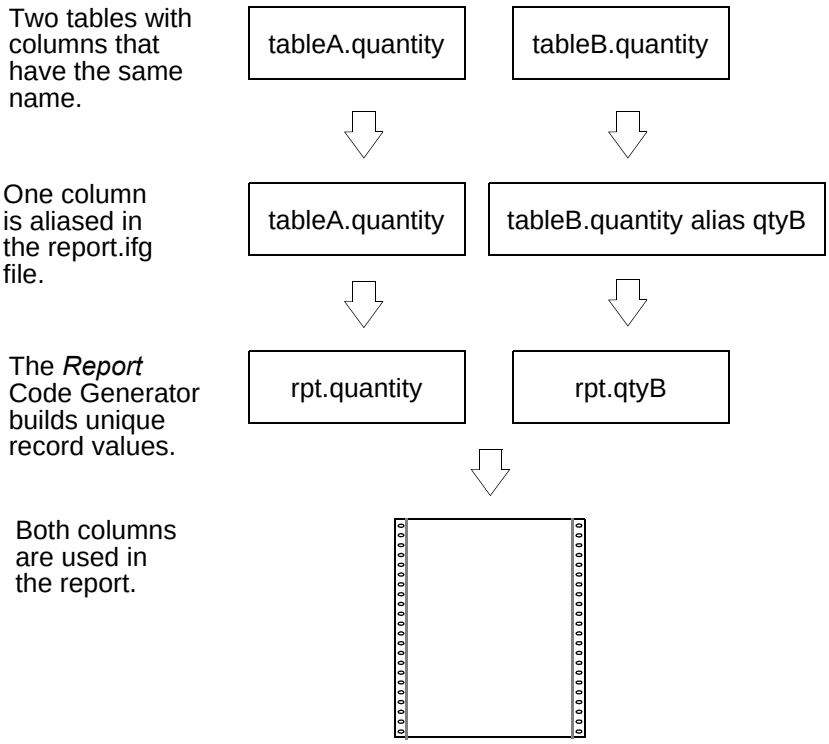
This chapter covers the following topics:

- Overview
- Setting up a Column Alias
- Changing the Column Format

Overview

You may encounter a situation in which you want to build a report from two different columns that have the same name but contain different data. Or you may want to use the same column twice and on the second use assign a different "using" format. By employing column aliasing, you can do both.

Column aliasing simply lets you assign an alias to a column name. For example, you may have two tables (call them table A and table B) that both contain a column named quantity. In order to show this column from both tables, you must assign an alias to one of them.



Setting up a Column Alias

You can assign a column alias to any column in your `report.ifg` file. When you create an alias, the alias name is used in the appropriate `usg`, `rpt`, `curs`, or `curs_next` record instead of the actual column name.

The alias Line

To apply an alias to a column you must specify an alias in the `report.ifg` file. In the attributes section of this file, you can follow a column definition with the alias keyword and the alias name you want to use.

Syntax

When setting up an alias, use the following syntax:

table.column **alias** ***alias_name***

Example

Suppose you have a report that shows the quantity of your stock at three different stores. The attributes section of the `report.ifg` file may look as follows:

```
attributes
    A0   = today
    A1   = constant "Orders"
    A2   = pageno using "<<<<<"
    A3   = orders.customer_num using "<<<<<<<<<<<"
    A5   = orders.order_date
    A6   = stock.stock_num
    A7   = stock.description
    A8   = stock.unit_price
    AA   = storeA.quantity
    AB   = storeB.quantity
    AC   = storeC.quantity
end
```

As you can see, there are three tables that have a quantity column. For at least two of these tables, you must assign an alias. In this case, you might want to assign an alias to all three to make the resulting code easier to read. For example, you might want the following aliases: qtyA, qtyB, and qtyC.

The resulting attributes section should look as follows:

```
attributes
    A0   = today
    A1   = constant "Orders"
    A2   = pageno using "<<<<<"
    A3   = orders.customer_num using "<<<<<<<<<<"
    A5   = orders.order_date
    A6   = stock.stock_num
    A7   = stock.description
    A8   = stock.unit_price
    AA   = storeA.quantity alias qtyA
    AB   = storeB.quantity alias qtyB
    AC   = storeC.quantity alias qtyC
end
```

And the resulting `rpt` record definition in `globals.4gl` should look as follows:

```
rpt record
    customer_num like orders.customer_num,
    order_date like orders.order_date,
    stock_num like stock.stock_num,
    description like stock.description,
    unit_price like stock.unit_price,
    qtyA like storeA.quantity,
    qtyB like storeB.quantity,
    qtyC like storeC.quantity
end record,
```

Notes

Keep the following issues in mind when you create a column alias.

- Alias names follow the same naming conventions as column names. For example, alias names should not exceed 18 characters.
- You cannot create an alias for a formonly field.
- If you need to refer to the aliased column in a different part of the `report.ifg` file, you should use the actual table and column name. For example, if you want to sum an aliased column, the attribute definition should read:

```
AA   = storeA.quantity alias qtyA
AB   = storeB.quantity alias qtyB
AC   = storeC.quantity alias qtyC
AD   = sum(storeA.quantity)
```

Changing the Column Format

Besides showing values for columns with the same name, you can apply Column Aliasing in another manner. Suppose you want to show the same column in separate locations on your report. In one location you want to format the column flush left and in the second location you want to use the default formatting. You can use column aliasing to accomplish this task.

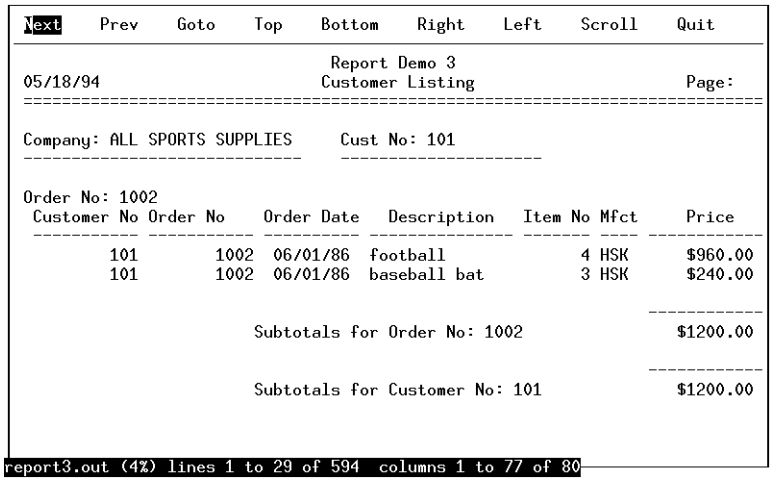
Note

The example shown below is from report demonstration three. If you want to see this scenario played out, type `rpt_demo 3`. Examine the `report.ifg` file then run the Fitrix *Report* Code Generator and `fg.make`. Finally, run the report then view the `report.out` file using `fg.pager` or a text editor.

The following lines show the `attributes` section of a `report.ifg` file. Notice how the `customer.customer_num` and the `order.order_num` columns are listed twice. On the first listing, an alias is used to format the columns flush left. On the second listing, no using string is applied.

```
attributes
    A0   = today  using "MM/DD/YY"
    A1   = constant "Customer Listing"
    A2   = pageno using "<<<<<"
    A3   = customer.customer_num alias customer_alias using "<<<<<"
    B3   = customer.customer_num
    A6   = orders.order_num alias order_alias using "<<<<<"
    B6   = orders.order_num
    B7   = customer.company upshift
    A7   = orders.order_date using "MM/DD/YY"
    A8   = stock.description
    AC   = sum(items.total_price)
    AD   = sum(items.total_price)
    AE   = sum(items.total_price)
    A    = stock.manu_code
    A9   = stock.stock_num
    AA   = items.total_price
    C1   = constant "Report Demo 3"
end
```

When this `report.ifg` file is built and the report is run, the resulting report looks as follows:



Notice how the Customer No and Order No columns are flush left in some places (using the alias format) and flush right in other places (using the default format).

7

Concurrency

Concurrency checks stored data for integrity before processing that data into a report. With concurrency, report program users can be sure that the data selected for processing the report is as current and accurate as possible.

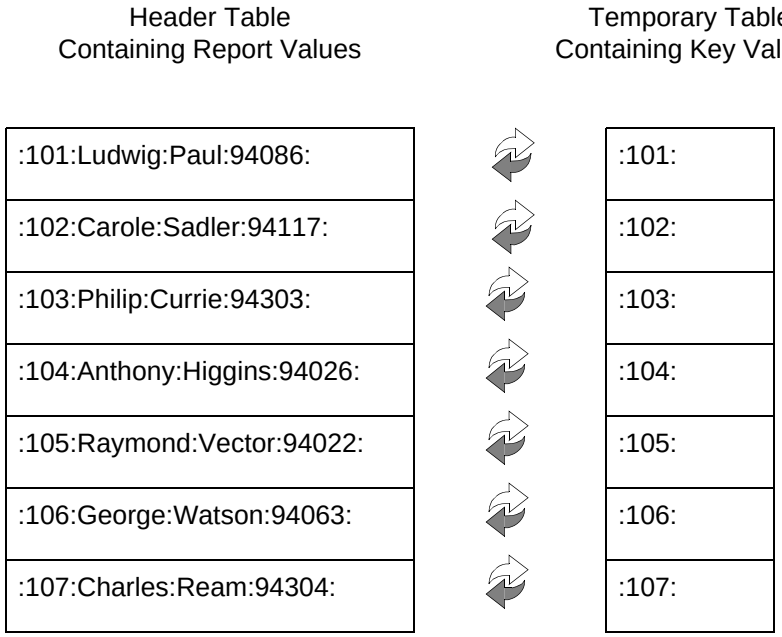
This section covers the following topics:

- Overview
- Implementing Concurrency
- Handling Concurrency Errors
- Code Examples

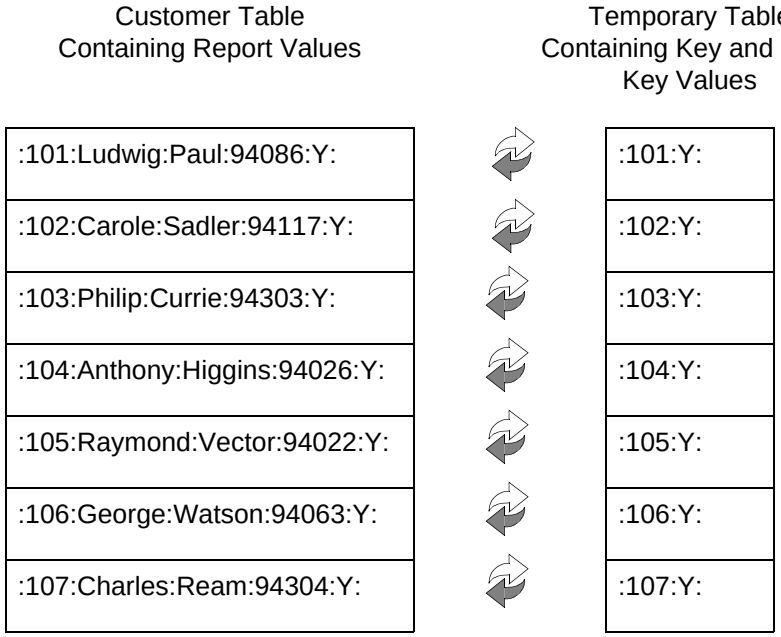
Overview

If you work with large report programs (ones that take several hours to complete), you know how data can change while the report is running. For example, during report processing, a user might delete data that appears on the report. Concurrency addresses this problem. With concurrency, you gain the ability to check each header row of a document before the data is printed. If the row is valid, the data is printed and the program processes the next row. If there is an error, the program sets a flag that you can handle programmatically.

In order to perform each check, concurrency builds and sorts a temporary table prior to processing the report. This temporary table is base on the *key column* of the header table. The key column uniquely identifies each row of the header table. Next, using rowid, concurrency sequentially locks each row in the header table and compares the current value with the temporary table value. Consider the following graphic showing a simple case:



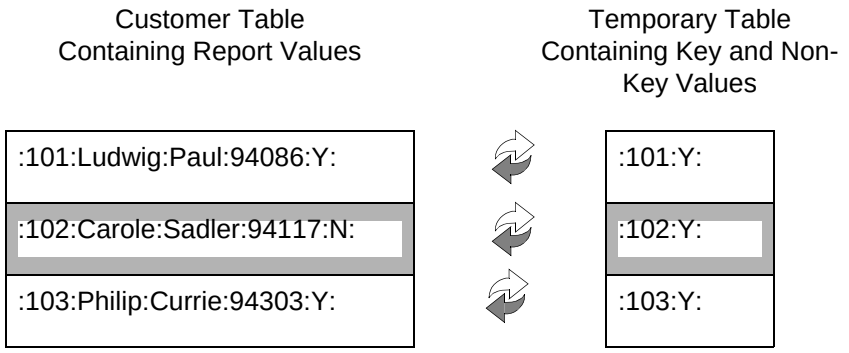
Although a key column value is required, you can also instruct concurrency to check additional column values. For example, posting programs often evaluate an `ok_to_post` column before posting a document. The following graphic expands on the previous one by showing a concurrency report that evaluates both key and non-key column values:



During concurrency's locking and comparison logic, three errors can occur:

1. The document may be locked, meaning someone is altering the document at the time of processing.
2. The document may be missing, meaning it was deleted.
3. The document may contain a value that has changed.

Consider the same example again. This time, however, notice how the ok_to_post non-key value for Customer 102 has changed.



When a concurrency error occurs, a flag, known as the bypass flag, is set. It is up to you to determine how you want to handle the error. For example, if the bypass flag is set to indicate a value has changed, you may want to print a note on the report.

Within the concurrency code, there are different points with which you can merge your own error handling logic (refer to "Handling Concurrency Errors" on page 7-10 for more information).

The following steps summarize how concurrency logic works:

1. Select the key column(s) of the report into a temporary table.
2. Sort the rows in the temporary table using the order-by column.
3. Use rowid to lock the first row of the header table.

   If the row is in transaction or missing, set the bypass flag.
4. Check for data integrity between the header table and the temporary table using both key and non-key columns (if specified).

   If the data does not match, set the bypass flag.
5. Process the detail lines for the header row.
6. Repeat the process for the next row in the header table.

Implementing Concurrency

Applying concurrency to a report program is a two-part process:

1. Set up the `report.ifg` file.

   To set up the `report.ifg` file, you must alter the `select` section and add a new section called `concurrency`.

2. Create custom code to handle concurrency errors.

   Once the report program detects a concurrency error, it must know how to process the error. Because each user handles errors differently, you must create and merge your own custom logic to process them.

Setting up the report.ifg File

The `report.ifg` file contains all of the instructions necessary to build a report. Because concurrency is a new feature, you must add a few more lines of instructions to the `report.ifg` file. These lines instruct the Fitrix *Report* Code Generator on how to build the 4GL code necessary to employ concurrency. As mentioned above, to set up the `report.ifg` file you must add two new lines to the select section and create a new concurrency section. If you are not familiar with `report.ifg` files, consult your *Report Code Generator Technical Reference*.

Select

If you have experience creating `report.ifg` files, you already are aware of the `select` section. This section contains information about how to build the report such as table names, joins, and filters. A standard `select` section uses the following syntax:

```
select
     more = table.column
    [more = table.column]
    table = table_name [, table_name, ...]
     join = table.column = table.column [...]
   filter = sel_criteria
    order = table.column [, table.column, ...]
end
```

Each line in the `select` section provides a different instruction to the *Report* Code Generator:

select: Designates this section in the `report.ifg` file. The `select` line must always appear first.

more: Specifies a column that is required for processing but does not appear on the report. Each column must have its own `more` line. For example, if your report contains three columns that are required for processing but do not appear on the report, you must create three separate `more` lines.

A common example is the `ok_to_post` column again. This column must be referenced to check for a posting flag, but it does not need to appear on the final report.

tables: Specifies the tables used in the report. Unlike the `more` line, the `tables` line does not have to be unique. You can reference multiple tables on the same line.

join: Specifies the columns that join the tables on the report.

order: Specifies the order that columns are fetched and processed.

end: Designates the conclusion of the `select` section.

Here is a typical `select` section for a simple report:

```
select
    more   = items.item_num
    tables = customer, orders, items, stock
    join   =  stock.manu_code = items.manu_code and stock.stock_num = items.stoc
k_num and orders.customer_num = customer.customer_num and items.order_num = orde
rs.order_num
    filter = customer.customer_num > 104
   order   = customer.customer_num,orders.order_num,items.item_num
end
```

When you implement concurrency instructions, the `select` section changes a bit and several lines have a slightly different meaning. Perhaps the biggest change involves separating out header table information from detail table information. With concurrency, only header table information goes in the `select` section. Detail table information is contained in the new `concurrency` section (described later).

A select section that contains concurrency instructions uses the following syntax:

```
select
```

```
     more  = table.column
    [more  = table.column]
    table  = table_name [, table_name, ...]
     join  = table.column = table.column [...]
   filter  = sel_criteria
    order  = table.column [, table.column, ...]
  notfound = table.column [, table.column, ...]
     [save = table.column [, table.column, ...]]
end
```

While the `select`, `more`, and `end` lines mean the same, the `table`, `join`, `filter`, and `order` lines take on a slightly new meaning. In addition, two new lines (`notfound` and `save`) are added:

tables: Specifies the header table and reference tables used by the header table. You now specify detail tables in the `concurrency` section.

join: Specifies joins between the header table and the tables referenced by the header table. Again, you now specify joins involving detail tables in the `concurrency` section.

order: Specifies the order header columns are fetched and processed.

notfound: Specifies the key column. In order to apply concurrency you must specify a key column that uniquely identifies each row in the header table. The `notfound` line holds this value.

save: Specifies non-key columns that are selected to the temporary table. Unlike the `notfound` line, the `save` line is optional.

The following code shows a `select` section modified to accommodate concurrency:

```
select
    more     = items.item_num
    tables   = customer
    filter   = customer.customer_num > 104
    order    = customer.customer_num
    notfound = customer.customer_num
    save     = customer.company
end
```

In the case of this example, the `join` line is not necessary because there are no reference tables used by the header table (`customer`). The only joins that exist are between detail tables, and these joins are specified in the `concurrency` section. Also look at the `notfound` and `save` lines. These lines contain the header table values that are selected into the temporary table before the report is processed.

Concurrency

Unlike the `select` section, which you may have been familiar with previously, the `concurrency` section is a new section as of the 4.12 Fitrix *Report* Code Generator. This section passes instructions to the Generator concerning the detail table. A standard `concurrency` section uses the following syntax:

```
concurrency
    cursor = detail_curs
    tables = table_name
    filter = table.column = ?
    filler = table.column
    join   = table.column = table.column [...]
   order   = table.column [, table.column, ...]
end
```

concurrency: Designates this section in your `report.ifg` file. The `concurrency` line must appear first.

cursor: Names the detail cursor. For now, the detail cursor must always be set to `detail_curs`. This is a required line.

tables: Specifies both the detail table and the reference tables that are used by the detail table.

filter: Specifies which columns to fetch from the detail table(s). The question mark (?) is used because this value changes as key header values change. The question mark represents a dynamic and changing value.

filler: Specifies the value of the header column passed to the question mark (?) in the `filter` line.

join: Specifies the joins between the detail tables and the tables referenced by the detail tables.

order: Specifies the order in which detail rows are fetched and processed.

Building on the previous example, the `concurrency` section takes on the following values:

```
concurrency
    cursor = detail_curs
    tables = orders, items, stock
    filter = orders.customer_num = ?
    filler = customer.customer_num
```

```
    join   = stock.manu_code = items.manu_code and stock.stock_num = items.stock_num
and items.order_num = orders.order_num
   order   = orders.order_num, items.item_num
end
```

You should note, however, that the 4.12 Fitrix *Report* Code Generator can read a syntax for the filter and join lines. Instead of building the join clause entirely on one line, you can now break it up across several lines, for example consider the above concurrency section again:

```
concurrency
    cursor = detail_curs
    tables = orders, items, stock
    filter = orders.customer_num = ?
    filler = customer.customer_num
    join   = stock.manu_code = items.manu_code and
    join   = stock.stock_num = items.stock_num and
    join   = items.order_num = orders.order_num
   order   = orders.order_num, items.item_num
end
```

Notice how two `join` lines have been added to make the join statement easier to read. Also notice how the "and" clauses remain between each join. In reality, you are not changing any of the information within the join line, you are simply formatting this information in a more readable manner. This same syntax applies to a long `filter` lines as well.

Perhaps the most confusing lines in the `concurrency` section are the `filter` and `filler` lines. These lines work together. The `filter` line specifies the many side of the one-to-many relationship and the `filler` line specifies the one side. The question mark acts as a dynamic value; it takes on the value of the each header row, which is specified in the `filler` line.

When you finish adding concurrency instructions to your `report.ifg` file, you must determine how to handle concurrency errors and the bypass flag. The next section covers handling concurrency errors and provides a few error-handling examples.

Handling Concurrency Errors

Properly setting up your `report.ifg` file is only part of employing concurrency. You still need to create custom logic to handle concurrency errors. A concurrency error is defined as any condition that sets the bypass flag. Typically, there are three such conditions:

1. The document may be locked, meaning someone is altering the document at the time of processing.
2. The document may be missing, meaning it was deleted.
3. The document may contain a value that has changed.

You can also create your own custom logic to handle additional conditions.

The next three examples illustrate some common ways to handle each type of concurrency error mentioned above.

Example One: Displaying a Warning Message

During the concurrency locking logic, a condition may arise where a selected document is already "in transaction" (i.e., the document is locked because a system user is updating it). One way to handle this condition is to display a warning message to the screen. Besides the warning message, however, you also have to reinitialize the `rpt` record values so that the final report does not show data from this document.

The following extension file shows how to call a warning message and reset the `rpt` record values. This extension file is built for `rpt_demo 5`.

Note

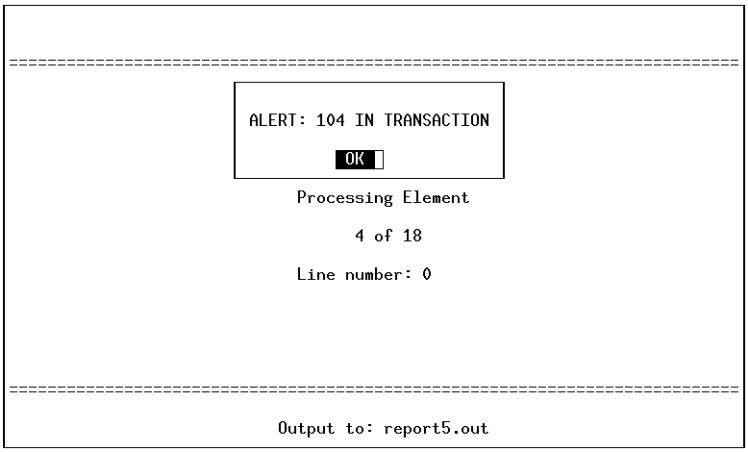
The `warnbox` function used in this example is part of the `prog_ctl` library, which comes with the Enhancement Toolkit. This library contains compiled C functions. If you are developing in an INFORMIX-RDS environment, you need to run `mkrunners`. This script creates a custom pseudo-code runner (consult your Enhancement Toolkit documentation).

```
#########################################################################
```

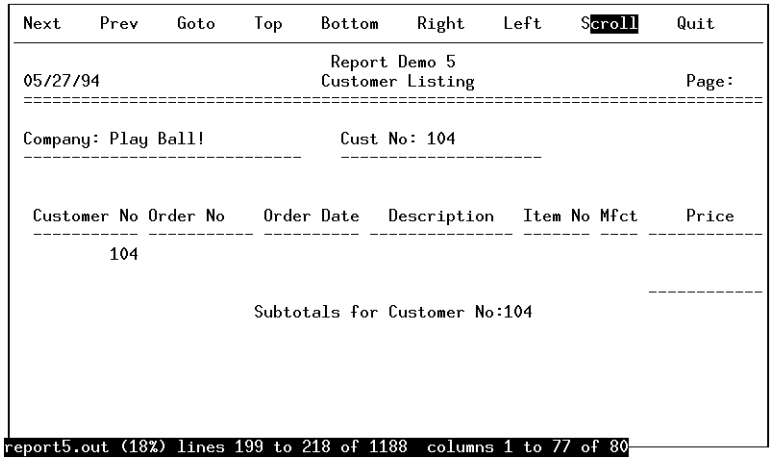
```
start file "Makefile"
##############################################################################
    libraries
        $fg/lib/prog_ctl.a
;


##############################################################################
start file "globals.4gl"
##############################################################################

define
    tmpStr char(40)
;

##############################################################################
start file "lowlevel.4gl"
##############################################################################

after block b_g_customer_num doc_locked_error
    let tmpStr="ALERT: ",
        curs.customer_num using "###",
        " IN TRANSACTION"
    call warnput(tmpStr)
    call warnbox()
;

after block on_detail on_bypass
    let rpt.order_date = null
    let rpt.description = null
    let rpt.order_num = null
    let rpt.manu_code = null
    let rpt.item_num = null
    let rpt.stock_num = null
    let rpt.total_price = null
;

after block b_g_customer_num init_bypass
    let tmpStr = null
;
```

The first block statement in this extension file simply adds the `prog_ctl` library to the `Makefile`. The second block statement defines a variable. The third statement builds the message text and calls the `warnbox()` and `warnput()` functions. The fourth block statement sets the detail table values in the `rpt` record to `null` and the fifth block statement sets the temporary message variable to `null`.

When you use this logic with `rpt_demo 5`, a message similar to the following one appears when a document is *in transaction*:



In this case, the document with `customer_num = 104` is being updated. Besides showing this message, the detail lines associated with 104 are excluded from the report:



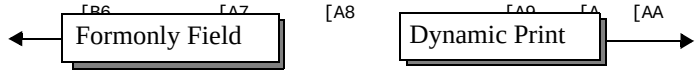
Example Two: Writing A Message to the Report

Instead of the warning box, some report users would rather see a note on the report itself when a concurrency error occurs. In this example, if a document has been deleted or is currently being updated, a note appears on the report.

Add a formonly Field

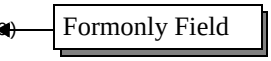
To create a line for the note, a formonly field is added to the `report.ifg` file. This field is placed on a row by itself with a dynamic print symbol at the end of the line. The following code shows the `on every row` section of the `report.ifg` file. The E1 tag signifies the formonly field. The [* symbols instruct the report to only print this line when a value exists for it.

```
on every row
{
 [B3          [B6          [A7      [A8          [A9    [A   [AA          [*
 [E1
}
```



The formonly field is defined as a character field. When an error occurs, an error message will be written to the formonly field. The following code shows the `attributes` section of the `report.ifg` file. Notice attribute E1. This attribute defines the formonly field.

```
attributes
    A0   = today  using "MM/DD/YY"
    A1   = constant "Customer Listing"
    A2   = pageno using "<<<<<"
    A3   = customer.customer_num using "<<<<<"
    B3   = customer.customer_num
    B6   = orders.order_num
    B7   = customer.company
    A7   = orders.order_date using "MM/DD/YY"
    A8   = stock.description
    AC   = sum(items.total_price)
    AD   = sum(items.total_price)
    A    = stock.manu_code
    A9   = stock.stock_num
    AA   = items.total_price
    C1   = constant "Report Demo 5"
    E1   = formonly.error type char(40)
```



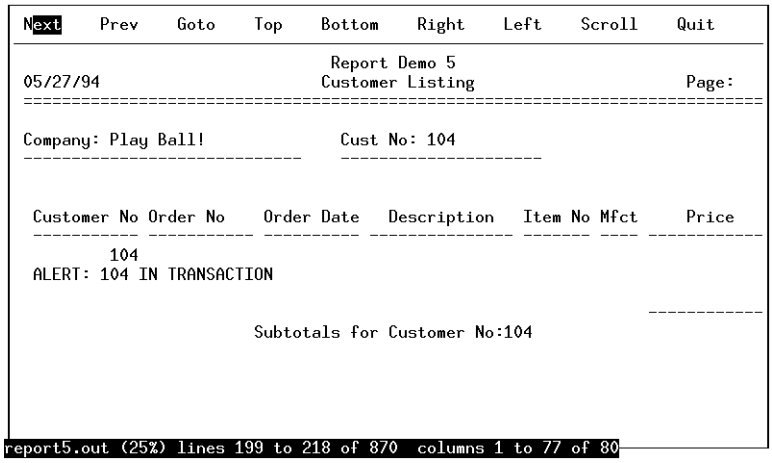
Create an Extension File

Once you have altered the `report.ifg` file, you must create an extension (*.ext) file. The extension file contains custom logic to handle the error and write the message to the formonly field. The following code shows an example extension file that can be used with report demonstration 5.

```
########################################################################
start file "globals.4gl"
########################################################################

define
    tmpStr char(40)
;

########################################################################
start file "lowlevel.4gl"
########################################################################

before block b_g_customer_num init_bypass
    let rpt.error = null;

after block b_g_customer_num doc_locked_error
    let tmpStr="ALERT: ", curs.customer_num using "###", " IN TRANSACTION"
    let rpt.error = tmpStr
;

after block b_g_customer_num doc_deleted_error
    let tmpStr="ALERT: ", curs.customer_num using "###", " HAS BEEN DELETED"
    let rpt.error = tmpStr
;

after block on_detail on_bypass
    let rpt.order_date = null
    let rpt.description = null
    let rpt.order_num = null
    let rpt.manu_code = null
    let rpt.item_num = null
    let rpt.stock_num = null
    let rpt.total_price = null
;

after block b_g_customer_num init_bypass
    let tmpStr = null
;
```

Generate and Compile

After you alter the `report.ifg` file and create the above extension file, you must generate (`fg.report`) and compile (`fg.make`) the report program. When the program runs and encounters a concurrency error, the following message appears:



In this case, the message indicates that document 104 is in transaction.

Code Examples

The following code is taken from the `midlevel.4gl` and `lowlevel.4gl` files in report demonstration five. Callout boxes have been added to show you good locations to merge your own custom logic and the purpose of different functions throughout the code.

midlevel.4gl

```
########################################################################
function ml_join()
########################################################################
#

    #_define_var - Define local variables

    #_err - Trap fatal errors
    whenever error call error_handler

    #_sel_join - Set the join criteria
    let sel_join =
        " 1=1"

end function
# ml_join()

########################################################################
function ml_filter()
########################################################################
#

    #_define_var - Define local variables

    #_sel_filter - Set the filter criteri
    let sel_filter =
        " 1=1"

end function
# ml_filter()

########################################################################
function ml_order()
########################################################################
#

    #_define_var - Define local variables
```

This line sets the join criteria. In this case, no specific join has been defined, so the default is simply " 1=1".

This line sets the filter criteria. In this case, no specific filter has been defined, so the default is simply " 1=1".

```
    #_sel_order - Set the order criteria
    let sel_order =
        " customer_num"

end function
# ml_order()

#########################################
function ml_getcount()
#########################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        syn_no smallint,           # Synonym number
        n smallint,                # Synonym counter
        sel_stmt char(4096)        # Sele

    #_create_temp - Create the temp table
    select
        customer.rowid h_row,
        customer.company,
        customer.customer_num
      from
        customer
      where customer.rowid = 0
      into temp curs_temp with no log

    #_insert_temp - Select data into temp table
    let sel_stmt =
    "insert into curs_temp ",
          "select ",
              "customer.rowid, ",
            "customer.company, ",
            "customer.customer_num ",
          "from ",
            "customer "

        #_chk_translation - logic for translated fields
        if is_translated is not null
        then
            for syn_no = 1 to num_trans - 1
                #_build_synon - build synonyms
                let sel_stmt =
                    sel_stmt clipped, ", stxlangr t", syn_no using "<<"
            end for
            #_write_last - write last synonym
            let sel_stmt =
                sel_stmt clipped, ", stxlangr t",  num_trans using "<<"
        end if

        #_cont_getcount - Continue building getcount select
```

This line sets the specifies the order in which header table rows are processed. In this case, `customer_num` is used.

This select statement builds the temporary table. The columns are taken from the `order`, `not-found`, and `save` lines in the `report.ifg` file.

This insert statement populates the temporary table with data from the actual header table.

```
        let sel_stmt = sel_stmt clipped," ",
              "where ",
                "(", sel_join clipped, ") and ",
                "(", sel_filter clipped, ")"

    #_set_ct_sel_stmt - Set the ct.sel_stmt variable for
    #                   display during error handling
    let ct.sel_stmt = sel_stmt clipped

    #_count_cursor - Prepare and execute the cursor

        #_prep_curs - Prepare the string for execution
        prepare get_count from sel_stmt

        #_execute_curs - Execute the string
        execute get_count

        #_set_row_count - Set number of row to process
        let ct.num_rows = sqlca.sqlerrd[3]

end function
# ml_getcount()

#########################################################################
function ml_define_cur()
#########################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        syn_no smallint,           # synonym counter
        n smallint,                # synonym counter
        sel_stmt char(4096)        # Selection statement

    #_fetch_temp - Fetch data from the temp table
    let sel_stmt =
        "select ",
            "h_row, ",
            "company, ",
            "customer_num ",
          "from ",
            "curs_temp"

        #_chk_translation - logic for translated fields
        if is_translated is not null
        then
            for syn_no = 1 to num_trans - 1
                #_build_synon - build synonyms
                let sel_stmt =
                    sel_stmt clipped, ", stxlangr t", syn_no using "<<"
            end for
            #_write_last - write last synonym
```

This line builds the `rpt_cursor` to fetch data from the temporary table.

```
            let sel_stmt =

                sel_stmt clipped, ", stxlangr t",  num_trans using "<<"
        end if

    #_include_order - Include any valid c
    if sel_order is not null
    then
        let sel_stmt = sel_stmt clipped,
            " order by ", sel_order clipped
    end if

    #_set_ct_sel_stmt - Set the ct.sel_stmt variable for
    #                   display during error handling
    let ct.sel_stmt = sel_stmt clipped

    #_rpt_cursor - Prepare and execute the cursor

        #_prep_curs - Prepare the string for execution
        prepare get_curs from sel_stmt

        #_declare_curs - Declare cursor from the string
        declare rpt_cursor cursor with hold for get_curs

        #_read_data - Read the data
        open rpt_cursor

end function
# ml_define_cur()

#########################################################################
function ml_fetch()
#########################################################################
#

    #_define_var - Define local variables

    #_fetch_cursor
    fetch rpt_cursor
      into
        #_fetch_list
        curs_next.h_row,
        curs_next.company,
        curs_next.customer_num

end function
# ml_fetch()
```

If an order by column has been specified, it is applied here.

Flow control calls the `ml_fetch()` function to get next row from the temp. table. These rows are put into the `curs_next` record.

lowlevel.4gl

```
globals "globals.4gl"

#_local_static - Local (static) variable definition
define
    #_misc_static - Misc static variables
    line_display smallint,          # bool
    sv_old record                   # colu
        #_sv_old_columns - columns used f
        customer_num char(18),
        company char(18)
    end record,
    line_no_pos smallint,           # position to print count at
    lineStrDis char(60)             # string used in printing count


########################################################################
function before_group(group_key)
########################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        group_key char(20)          # group identification

    #_err - Trap fatal errors
    whenever error call error_handler

    #_first_row - Check for first row
    if group_key = "first_row"
    then
        #_call_first_row - Call function for processing
        call b_g_first_row()
    end if

end function
# before_group()

########################################################################
function b_g_first_row()
########################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        sel_stmt char(4096)         # Selection statement

    #_err - Trap fatal errors
    whenever error call error_handler
```

Saves the columns to check for changed values. These are the columns from the `notfound` and `save` lines in the *.ifg file.

```
#_b_first_row - Before first row processing

#_after_first_row - After first row processing

#_init_messages

#_line_number - String for displaying line number
let lineStrDis = fg_message("standard","concurr",1)

#_set_line_display - True if ok to display line count
if downshift(ct.destin) != "screen" and ct.quiet = 1
then
    let line_display = true
else
    let line_display = false
end if

# display the "line number" line, centered.  Note:  the actual
# number is displayed with up to eight characters, but this will
# center based on the assumption of two characters.
#_chk_line_display
if line_display
then
    # 76 is window width.  "3" is "space & line_no".  ".5" is
    # fudge that starts it in the right spot regardless if
    # the result is odd or even.
    #_set_line_no_pos
    let line_no_pos =
        (76 / 2) - ((length(lineStrDis) + 3) / 2) + .5
    #_display_mssg - Display line number message
    display lineStrDis at 14, line_no_pos

    # now set line_no_pos to the actual print position
    # of the line number.
    #_set_line_no_pos
    let line_no_pos = line_no_pos + length(lineStrDis) + 1
end if

#_build_main_curs - Build the main cursor
let sel_stmt =
    "select ",
        "customer.customer_num, ",
        "customer.company ",
      "from ",
        "customer ",
      "where ",
        "customer.rowid = ? ",
      "for update"

#_prep_main_curs - Prepare the main c
prepare s_main_curs from sel_stmt
```

When `line_display` is set to true, record number is displayed to screen during processing.

This statement builds the header cursor from the header table.

Locks header rows report is processing so values in both header and detail lines does not change during processing.

```
    #_declare_main_curs - Declare the main cursor
    declare main_curs cursor for s_main_curs

    #_build_detail_curs - Build the detail cursor
    let sel_stmt =
        "select ",
            "orders.order_date, ",
            "stock.description, ",
            "orders.order_num, ",
            "stock.manu_code, ",
            "items.item_num, ",
            "stock.stock_num, ",
            "items.total_price ",
          "from ",
            "orders, items, stock ",
          "where ",
            "(orders.customer_num = ?) and (stock.manu_code ",
            "= items.manu_code and stock.stock_num = items.stock_num ",
            "and items.order_num = orders.order_num) ",
          "order by ",
            "orders.order_num, items.item_num"

    #_prep_detail_curs - Prepare the detail cursor
    prepare s_detail_curs from sel_stmt

    #_declare_detail_curs - Declare the detail cursor
    declare detail_curs cursor for s_detail_curs

    #_prepare_others - Build & prepare other statements

end function
# b_g_first_row()

########################################
function b_g_customer_num()
########################################
#

    #_define_var - Define local variables

    #_err - Trap fatal errors
    whenever error call error_handler

    #_init_bypass - Initialize bypass flag
    let rpt.bypass_doc = false

    #_check_line_number - Display line number if true
    if line_display
    then
        #_clear_line_number - Clear line number displa
        display "0       " at 14, line_no_pos
    end if
```

After header row is locked, detail cursor is built.

For this report, this function is the lowest level before group.

The bypass flag is initialized to false. This flag is evaluated before processing detail rows.

```
#_save_customer_num - save original data for customer_num
let sv_old.customer_num = curs.customer_num

#_save_company - save original data for company
let sv_old.company = curs.company

#_lock_header = retrieve header infor and lock the

# begin work and fetch the header.  This locks the header row.
begin work

#_open_main_curs - Open the main cursor
open main_curs using curs.h_row

#_error_continue - Continue if error on fetch
whenever error continue

#_fetch - Fetch next row from main_curs
fetch main_curs into
    #_fetch_columns - Columns cursor fetched i
    curs.customer_num,
    curs.company

#_error_handler - Reset after fetch to handler
whenever error call error_handler

#_check_for_errors - Check for retrie
case
  #_doc_locked
  when sqlca.sqlcode < 0
    #_doc_locked_do - Row is locked
    let rpt.bypass_doc = true
    let rpt.customer_num = sv_old.customer_num
    let rpt.company = sv_old.company

    #_doc_locked_error - Error handling

    #_doc_locked_ret - Return, can't do anythi
    return

  #_doc_deleted
  when sqlca.sqlcode = notfound
      or curs.customer_num != sv_old.customer_num
    #_doc_deleted_do - Row has been deleted
    let rpt.bypass_doc = true
    let rpt.customer_num = sv_old.customer_num
    let rpt.company = sv_old.company

    #_doc_deleted_error - Error handling

    #_doc_deleted_ret - Return, can't do anyth
    return
```

These lines save the original data to temp. table for comparison.

Fetches original data from header table.

Checks to see if document is locked. If it is locked, the bypass flag is set to true.

This block is where you insert custom error handling logic for locked document errors.

Checks to see if document is still current.

This block is where you insert custom error handling logic for in case document is missing.

```
      #_other_when - Tag for additional when statements

    end case

    #_b_customer_num - Before group processing

    #_company - Before group processing for compan
    let rpt.company = curs.company

    #_customer_num - Before group processing for c
    let rpt.customer_num = curs.customer_num

    #_a_customer_num - Post before group processing

end function
# b_g_customer_num()

################################################
function on_detail()
################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        detailRow smallint         # Detail line count

    #_init - Initialize
    let detailRow = 0

    #_check_bypass_doc - Is bypass_doc set to true
    if rpt.bypass_doc
    then
        #_on_bypass - Do this on bypass
        #_exit_bypass - Exit because we are bypassing doc
        return
    end if

    #_open_detail_curs - Open the detail cursor
    open detail_curs using
         curs.customer_num

    #_initial_fetch - Do the initial fetch
    fetch detail_curs into
        curs.order_date,
        curs.description,
        curs.order_num,
        curs.manu_code,
        curs.item_num,
        curs.stock_num,
        curs.total_price

    #_check_no_detail - Are there any detail rows
```

This block is where you insert custom logic to evaluate other error conditions. For example, if you want to evaluate a posting flag or some other table value.

This function fetches each detail row from the detail table.

Checks bypass flag before processing.

```
    if sqlca.sqlcode = notfound
    then
        #_on_no_detail - Do this for no detail
        #_exit_no_detail - Exit because there is no detail
        return
    end if

    #_process_detail - Process the detail
    while true

        #_increment_count - Increment the detail row count
        let detailRow = detailRow + 1

        #_check_display - Display row count if true
        if line_display
        then
            #_display_row - Display the row count
            display detailRow using "<<<<<<<<"
                at 14, line_no_pos
        end if

        #_process_row - Do row processing
        call on_every_row()

        #_fetch_detail - Fetch more detail rows
        fetch detail_curs into
            curs.order_date,
            curs.description,
            curs.order_num,
            curs.manu_code,
            curs.item_num,
            curs.stock_num,
            curs.total_price

        #_check_no_more_rows - Do if there are no more row
        if sqlca.sqlcode = notfound
        then
            #_on_no_more_rows - Do this for no more rows
            #_exit_no_more_rows - Exit.  No more rows
            exit while
        end if

        #_call_ml_output - Call for each row except last
        call ml_output()

    end while

end function
# on_detail()

###################################################
function on_every_row()
###################################################
```

Process and display each row.

This function does all of the assignments to the rpt record for the detail rows.

```
# This function prepares the report record from the
# cursor record and other data.
#

    #_define_var - Define local variables

    #_err - Trap fatal errors
    whenever error call error_handler

    #_before_every_row - Before on every row assignments

    #_item_num - On every row processing for item_num
    let rpt.item_num = curs.item_num

    #_order_date - On every row processing for order_date
    let rpt.order_date = curs.order_date

    #_order_num - On every row processing for order_num
    let rpt.order_num = curs.order_num

    #_description - On every row processing for description
    let rpt.description = curs.description

    #_total_price - On every row processing for total_price
    let rpt.total_price = curs.total_price

    #_stock_num - On every row processing for stock_num
    let rpt.stock_num = curs.stock_num

    #_manu_code - On every row processing for manu_code
    let rpt.manu_code = curs.manu_code


    #_after_every_row - After on every row assignments

end function
# on_every_row()

################################################
function a_g_customer_num()
################################################
#

    #_define_var - Define local variables

    #_err - Trap fatal errors
    whenever error call error_handler

    #_b_customer_num - After group processing for

    #_customer_num - After group processing for customer_num
    let rpt.customer_num = curs.customer_num
```

Does after-group processing for group by column.

```
    #_a_customer_num - After group processing for

    #_wrap_up_work - Commit or rollback work
    call wrap_up_work()

end function
# a_g_customer_num()

##########################################################################
function after_group(group_key)
##########################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        group_key char(20)         # group identification

    #_last_row - Check for last row
    if group_key = "last_row"
    then
        #_call_last_row - Call function for processing
        call a_g_last_row()
    end if

end function
# after_group()

##########################################################################
function a_g_last_row()
##########################################################################
#

    #_define_var - Define local variables

    #_err - Trap fatal errors
    whenever error call error_handler

    #_b_last_row - Before last row processing

    #_a_last_row - After last row processing

end function
# a_g_last_row()

################################################
function wrap_up_work()
################################################
# Commit or rollback the work for the document.
#

    #_define_var - Define local variables
```

Final after group logic. Does a commit or rollback work depending on bypass flag.

```
    #_before_work - Before commit/rollback processing

    #_check_bypass_doc - Rollback work if true
    if rpt.bypass_doc
    then
        #_do_rollback - Rollback work
        rollback work
    else
        #_do_commit
        commit work
    end if

    #_after_work - After commit/rollback processing

end function
# wrap_up_work()
```

A

Improvements and Notes

Besides the features already mentioned, there are a few improvements and other notes you should be aware of. In this chapter, you can find information on a new access log file, you can see how to change where errlog files are created, and you can learn the new method for hiding ring menu options. In addition, a new flag has been added to the Fitrix *Screen* Code Generator. This flag prevents the Generator from creating a `Makefile`, which can be useful when you are building a custom library containing zoom screens.

This chapter covers the following topics:

- Log Files
- Generator Access Variables
- Set Explain Support
- Hiding Ring Menu Options
- Building a Library Zoom Screen

Log Files

With the 4.12 Fitrix *CASE* Tools, you can create a new access log file and change the default behavior of the errlog file. These abilities were brought about by the introduction of three new library files in `standard.4gs`:

- `logStart.4gl`
- `setAcc.4gl`
- `setErr.4gl`

The `logStart.4gl` file contains a new function called `logStart()`. This new function has replaced the traditional call to `startlog("errlog")` in `main.4gl`. The `logStart()` function accepts two arguments. The first specifies an environment variable pointing to the access log file. The second argument specifies an environment variable pointing to the `errlog` file.

When an argument is passed to the `logStart()` function, program flow sends that argument through `logStart.4gl` and then to the appropriate `set*.4gl` file.

The `setAcc.4gl` and `setErr.4gl` files each contain a single function that handles access log and errlog creation. These functions were set apart in their own file so you can modify and move them to a custom library more easily.

Creating an Access Log File

An access log file stores the time, name, and user login each time a program is run. For example the following lines show you sample output for an access log file created by screen demo five:

```
Date: 06/02/94    Time: 12:52:49
Program ID: demo.screen5  Login: brianh
Program Started
```

You can use one of three methods to create an access log file:

1. You can pass the `-accesslog` flag on the command line. For example:

   ```
   fglgo screen5.4gi -accesslog ./logfile
   ```

This example creates an access log file called `logfile` in the `screen5.4gs` program directory.

2. You can pass an environment variable to the `logStart()` function. The variable you pass must point to a specific file in the filesystem.

   For example, you can create the following environment variable:

   ```
   progAcs=$fg/logfile ; export progAcs
   ```

   Then merge an extension file to replace the `logStart()` function:

   ```
   #------------------------------------------------------------------------
   start file "main.4gl"
   #------------------------------------------------------------------------

   replace block main start_error_log
       call logStart("progAcs", "")
   ;
   ```

3. You can set `accesslog`, a global environment variable, to point to a specific file in the filesystem:

   ```
   accesslog=$fg/logfile ; export accesslog
   ```

The command line flag takes precedence over the other two methods and the extension file method takes precedence over the global variable method.

Relocating the errlog File

Along with the ability to create an access log file, you can change where the program errlog file is created. By default, an errlog file is created in the local program directory. This default behavior hasn't changed but now you can override this behavior and create an errlog file anywhere on the filesystem. Like the access log file, there are three ways to create an errlog file:

1. You can pass the `-errlog` flag on the command line. For example:

   ```
   fglgo screen5.4gi -errlog $fg/errlog
   ```

   This example creates an errlog file in the FourGen directory (`$fg`).

2. You can pass an environment variable to the `logStart()` function. The variable you pass must point to a specific file in the filesystem.

For example, you can create the following environment variable:

```
progErr=$fg/errors/errlog ; export progErr
```

Then merge an extension file to replace the `logStart()` function:

```
#-------------------------------------------------------------------------
start file "main.4gl"
#-------------------------------------------------------------------------

replace block main start_error_log
    call logStart("", "progErr")
;
```

3. You can set `errlog`, a global environment variable, to point to a specific file in the filesystem:

   ```
   errlog=$fg/errors/errlog ; export errlog
   ```

The command line flag takes precedence over the other two methods and the extension file method takes precedence over the global variable method.

You can give the errlog file any name you want. For example, you can set the errlog environment variable in the following manner:

```
errlog=$fg/errors/progerrs ; export errlog
```

This example builds a file named `progerrs`. Whenever a program fails, error text is written to this file and not the local errlog file.

Generator Access Variables

Similar to the access log and error log capabilities discussed earlier in this chapter, you can also set up access log and error log files for both the *Screen* and *Report* generators.

By setting up an access log file for these programs, you can see when the <Your Company Name> development tools were used to generate new applications or to regenerate existing applications. By setting up an error log file, you can keep track of all the errors that occur during program generation in a single file.

To implement an access log or error log file associated with the Fitrix *Report* Generator, set the `rptgenaccess` and `rptgenerrlog` variables to point to a specific file in your filesystem, for example:

```
rptgenaccess=$fg/rptacclog ; export rptgenaccess
rptgenerrlog=$fg/rpterrlog ; export rptgenerrlog
```

To implement an access log or error log file associated with the Fitrix *Screen* Generator, set the `scrgenaccess` and `scrgenerrlog` variables to point to a specific file in your filesystem, for instance:

```
scrgenaccess=$fg/scracclog ; export rptgenaccess
scrgenerrlog=$fg/screrrlog ; export rptgenerrlog
```

Set Explain Support

Informix database engines have a method for reporting the decisions made by the engine query optimizer. The optimizer is the intelligence in the engine that interprets requests and determines the best method for carrying them out. Its decisions are based on the existence of indexes, the number of rows in the various tables, and even the distribution of values, in the 6.0 and later releases.

When your program issues `set explain on`, the engine optimizer writes its query plan to a file called `sqexplain.out`, in your current directory. Or, if you are using I-Star, it writes this file in your home directory on the machine where your database server actually resides. This query plan includes the order of table access, how filters are applied, and what if any indexes are used in processing the query. It lets you know if any temporary tables will be created to handle order by sorting.

You can now make use of `set explain` in generated programs without having to recompile your programs. All you must do is pass the `-explain` flag at the command line when you run a program. For example:

```
fglgo report.4gi -explain
```

- or -

```
report.4ge -explain
```

The `init()` function in the `standard` library checks for the `-explain` flag. If this flag is present, the `set explain on` command is issued, and a file called `sqexplain.out` file is created in the program directory. By using the `set explain` statement, you can gain insight on how the database is being accessed and whether changing indexes may improve the decisions of the optimizer.

For example, if your queries seem to be taking longer than necessary, you may choose to change your indexing method. In a complex query, it may be difficult for you to know the order of actions taken by the optimizer, which in turn makes it difficult for you to determine what indexes should be added or dropped.

You might find you can prevent the creation of a temporary table by modifying your order by clause to use indexed columns, or, conversely, by creating an index to match your order by.

You can make use of the `set explain` statement from within the debugger by typing:

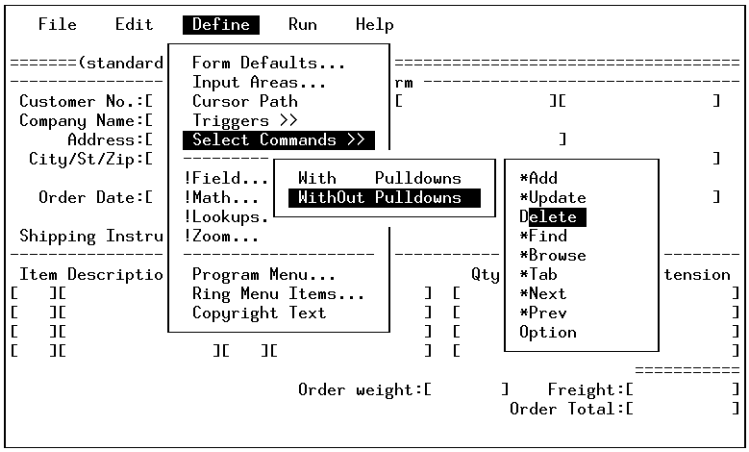
```
run -explain
```

at the debugger prompt.

Hiding Ring Menu Options

A problem introduced in the 4.11 Fitrix *CASE* Tools release disabled one aspect of the Select Commands option in the Form Painter. The 4.12 Fitrix CASE Tools fix this problem and improve on the previous functionality.

If you want to hide a ring menu option for a generated input program, you can load the corresponding form specification (*.per) file into the Form Painter. Then select the Define menu and choose Select Commands followed by WithOut Pulldowns:



Menu options preceded by an asterisk appear on the generated program. Those lacking an asterisk are removed from the program. You can hide/remove an option by selecting it to add/remove the asterisk.

For each menu option you hide, an extra line is written to `main.4gl`. For example, if you remove the Delete and Option, `main.4gl` contains the following lines:

```
#_hide - Hide options on the main ringMenu
call ringMenu_setOpt("HIDE", "Delete")
call ringMenu_setOpt("HIDE", "Option")
```

These lines represent new functionality that has been added to the upper level library main ring menu function. The first argument tells whether to hide or show the menu option. The second argument is the actual option. The available options for the second argument include:

- Add
- Update
- Delete
- Find
- Browse
- Next
- Prev
- Tab
- Option
- Quit

If you want to avoid using the Form Painter, you can hide any menu option using the `after_init` trigger. For example, to hide Delete and Option in screen demo five, you can create and merge the following `order.trg` trigger file:

```
defaults
    after_init
        call ringMenu_setOpt("HIDE", "Delete")
        call ringMenu_setOpt("HIDE", "Option");
```

Building a Library Zoom Screen

Quite commonly, zoom screens are used by more than one program, so a good location to place zoom screens is in a custom library where they may be accessed by multiple programs.

Placing zoom screens in a library is not much different than building them in the program directory. Here is a general outline of the steps you should follow when you are building a library of zoom screens:

1. Create the form specification (*.per) file for the zoom screen.

   You can create the specification file using the Form Painter or by hand with a text editor. If you use the Form Painter, don't forget to specify a returning value for the zoom screen.

2. Build a custom library `Makefile`.

   A library `Makefile` differs slightly from a program `Makefile`. The following library `Makefile` contains one zoom screen (`cust_zm`):

```
# Makefile for an Informix function library

TYPE = library

LIBFILES =  $(LIB)(cust_zm.o)

FORMS = cust_zm.frm

LIB=../lib_zm.a

#---------------------------------------------------------------------

all:
        @echo "make: Cannot use make.  Use fg.make to compile."
```

3. Once the *.per file and `Makefile` are built, run the *Screen* Code Generator to compile the form and create code for the zoom screen, type:

   `fg.screen -M`

   Because the Fitrix *Screen* Code Generator automatically creates a program `Makefile`, you want to pass the `-M` flag when you run this command. The `-M` flag is a new flag that prevents the Generator from creating a `Makefile` and thus overwriting the custom library `Makefile`.

4. Build and merge an extension (*.ext) file to set the form path.

   An extension file is necessary so programs can locate your zoom screen. Here is an example extension (*.ext) file for the `cust_zm` zoom screen:

   ```
   start file "cust_zm.4gl"
       replace block Acust_zm form_path
              with form "../lib_zm.4gs/cust_zm";
   ```

5. Create a `base.set` file to call your extension file and run `fg.make` to merge and compile the code.

   Depending on your development system, you may want to run `fg.make` twice. The first time you run it, pass the `-F` flag to build a compiled *.a library file. The second time you run it, pass the `-R` flag to build the *.RDS library directory.

Once you have completed the above steps, the library is built and ready to be used. To hook the zoom screen into one of your programs, however, you must remember to link in the library itself (with a libraries trigger). You also need to remember to attach the zoom screen to a calling field using either the Form Painter or by adding the zoom line to the program's main form specification file. For example, the following zoom line links in the library `cust_zm` zoom screen:

```
zoom      = key=customer_num, screen=cust_zm, table=customer
```

Index

A

Acceptable Values 2-7
Access Log File A-2
Aliasing, Column 6-2

C

Column Aliasing
- column formats 6-5
- overview 6-2
- setting up 6-3

Column Formats 6-5
Concurrency
- code examples 7-16
- handling errors 7-10
- implementing 7-5
- lowlevel.4gl 7-20
- midlevel.4gl 7-16
- overview 7-2
- setting up 7-5

D

Demonstration Programs
- benefits 1-2
- command bar 1-4
- main menu 1-5
- menus demo 1-6
- overview 1-2
- report demo 1-16
- screen demo 1-9
- starting 1-4

E

errlog File A-3

F

fg.screen -M A-10
Float Format
- acceptable values 2-7
- applying to report programs 2-12
- applying to screen programs 2-9
- cgxffmtr table description 2-2
- definitions 2-3
- functionality 2-5
- generated code 2-11
- overview 2-2
- rounding 2-5
- sample monetary values 2-2
- setting up 2-4

Formatting Columns 6-5

G

Generator Access Variables A-5

H

Hooking in Screens
- review 4-3
- socket_items trigger 4-3

L

Langauge Translation
- applying to report programs 3-6

Language Translation
- applying to screen programs 3-4
- overview 3-2
- stxlangr table 3-2
- y/n field logic 3-4

Library Containing Zoom Screens A-10
Library Makefile A-10
Log Files
- access log A-2
- errlog A-3

M

Menu, Hiding Options A-6

Menus Demo 1-6
ml_get_filter Function 5-9
ml_put_filter Function 5-8
ml_schedule Function 5-7

N

No Makefile Flag A-10

R

Report Column Aliasing 6-2
Report Concurrency
  code examples 7-16
  handling errors 7-10
  implementing 7-5
  lowlevel.4gl 7-20
  midlevel.4gl 7-16
  overview 7-2
  setting up 7-5
Report Demo
  building a program 1-18
  compiling the code 1-20
  running 1-16
  running the report code generator 1-20
  starting a program 1-17
Report Scheduling
  example 5-4
  implementing 5-3
  incorporating selection criteria 5-4
  library code 5-6
  midlevel.4gl 5-7
  new functions 5-3
  overview 5-2
Ring Menu, Hiding Options A-6
Rounding Floating Point Values 2-5
rptgenaccess Variable A-5
rptgenerrlog Variable A-5

S

Sample Monetary Values 2-2
Scheduling, Report 5-2
Screen Demo
  building a program 1-11
  compiling code 1-14
  opening a form specification file 1-12
  running 1-9
  running the screen code generator 1-13
  starting a program 1-10
Screen Hooking Logic
  overview 4-2
Screens, Zoom Library A-10
scrgenaccess Variable A-5
scrgenerrlog Variable A-5
Set Expain Statement A-6
socket_items Trigger
  adding 4-4
  description 4-3
  syntax 4-4
socketManager() Function 4-2
sockter_items Trigger
  example 4-4

Y

Y/N Field Logic 3-4

Z

Zoom Screens, Library of A-10