Fitrix™ *Report* Code Generator

Technical Reference

Version 4.11

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

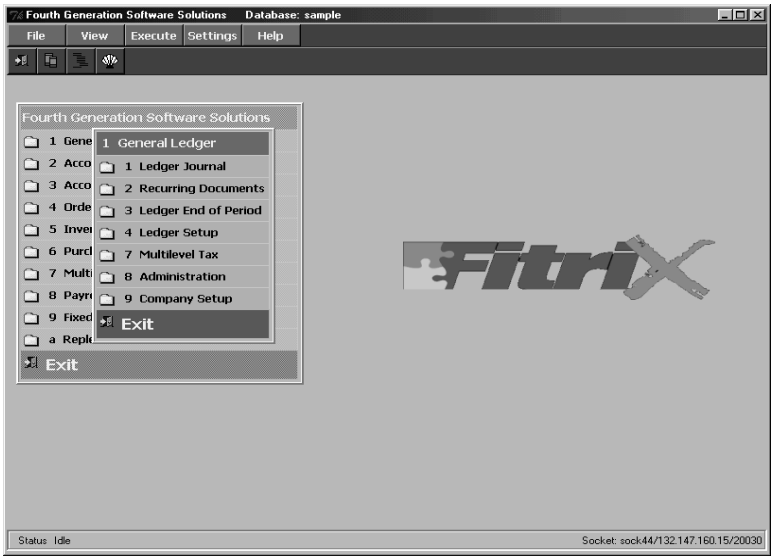
Corporate: (770) 432-7623
Fax: (770) 432-3448
E-mail: info@fitrix.com

Copyright

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

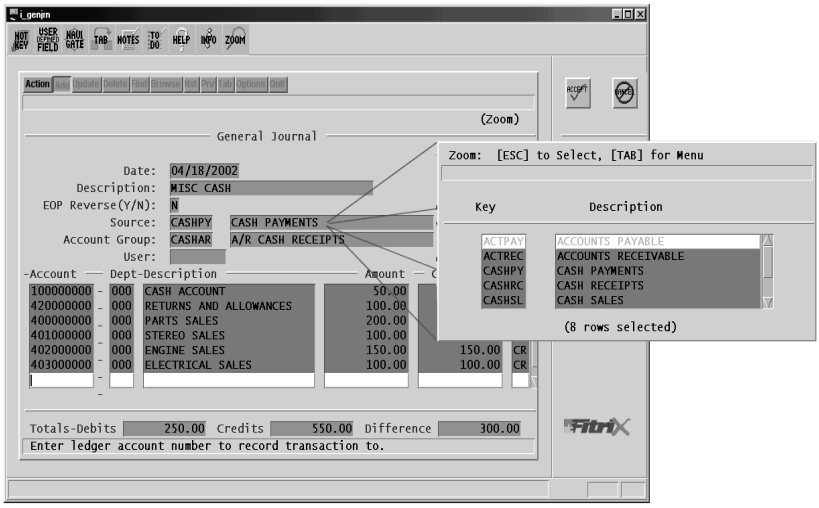
No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated.

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



Example 1: Menu Graphical Windows Mode

Here is another example:



Example 2: Data Entry Graphical Windows Mode

Displaying our products in graphic mode, as shown in Example 1 and Example 2, is customary for many Fitrix product users. However, your viewing mode is a user preference. Changing from character based to graphical based is a product specific procedure, so if you wish to view some applications in character mode, and some in graphical mode, that can be done as well.

If you have any questions about how to view your products in graphical mode, please consult your Installation Instructions or contact the Fitrix helpdesk at 1(800)374-6157. You can also contact us by email: support@fitrix.com. Please be prepared to offer your name, your company, telephone number, the product you are using, and your exact question.

We hope you enjoy using our products and look forward to serving you in the future.

Thank You,
Fourth Generation

Table of Contents

Chapter 3: Generating Source Code

Chapter 4: Compiling and Running

Chapter 5: Customizing Reports

Chapter 6: Creating Advanced Report Features

Chapter 7: New Features and Functionality

Chapter 8: Report Examples

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.

1

Introduction

Fitrix *Report* Code Generator uses the latest in Computer-Assisted Software Engineering (CASE) to produce complete 4GL code for structured, diagrammed report generation. Generated code is completely commented for you and Maintainable-By-Design (MBD).

This section covers the following topics:

- Code Generator Features
- Technical Reference Overview
- Installation and Preparation
- Directory Structure

Code Generator Features

Because of CASE technology, the Fitrix *Report* Code Generator can produce complete 4GL code for robust report programs. The Fitrix *Report* Code Generator:

- Creates hundreds of lines of INFORMIX-4GL code, which saves days of development time.
- Uses UNIX's `make` utility to manage code changes and minimize recompile time.
- Combines the power of the INFORMIX-4GL language with the ease-of-use of a sophisticated application generator.
- Adds first page headers, regular page headers, page trailers, and page breaks.
- Positions text flush left, flush right, or centered, and it truncates fields.
- Creates *dynamic* report elements including headers, lines, and footers.
- Allows *ad hoc* elements to be added at time of printing, including *where* selection and *order by* clauses.
- Supports runtime redirection of report output to screen, printer, file, or another program.
- Generates complete INFORMIX-4GL code, allowing you the absolute flexibility of changing or modifying anything you desire.
- Produces commented code to speed the addition of custom modifications for each report application.
- Reads report image files created with the Fitrix *Report* Writer.
- Structures and organizes code so that 90 percent of the basic routines never need to be touched or seen by the application developer.

In addition, you control all rights to Fitrix generated code. No special runtimes (other than INFORMIX-4GL runtime) are needed to move a compiled application to a different machine.

Technical Reference Overview

This reference manual contains eight sections. The following list shows the title and description of each section:

1. **Introduction:** Introduces the Fitrix *Report* Code Generator and describes product features and installation.
2. **Creating an Image (report.ifg) File:** Covers the first step in developing a complete report program. This section describes the `report.ifg` image file, illustrates an example `report.ifg` file, and explains `report.ifg` file components.
3. **Generating Source Code:** Covers the second step in developing a complete report program. This section shows how to create source code with the *Report* Code Generator.
4. **Compiling and Running:** Describes the third step in developing a complete report program. This section explains how to compile and run your report programs.
5. **Customizing Reports:** Describes the fourth step in developing a complete report program. This section shows how to create and merge modifications and customizations into report programs. It also illustrates how the Featurizer merges customizations into report source code.
6. **Creating Advanced Report Features:** Shows how to build and implement report prompts, scheduling programs, and other reporting events.
7. **New Features:** Describes new *Report* Code Generator features including larger selection variable sizes, new `Makefile` and image file block tags, and enhanced post processor flexibility.
8. **Examples:** Illustrates `report.ifg` and source code files. Also shows sample report output.

Installation and Preparation

In order to run Fitrix *Report* Code Generator, make sure your system contains the following items:

- UNIX/XENIX operating system
- INFORMIX-4GL version 4.10 or later
- C language compiler
- The standard UNIX `make` utility
- Fitrix *Report* Code Generator program

INFORMIX-4GL installations

Follow installation instructions included with the program diskettes. These instructions include steps for installing the C compiler/Development System and the `make` utility.

Report Code Generator Installations

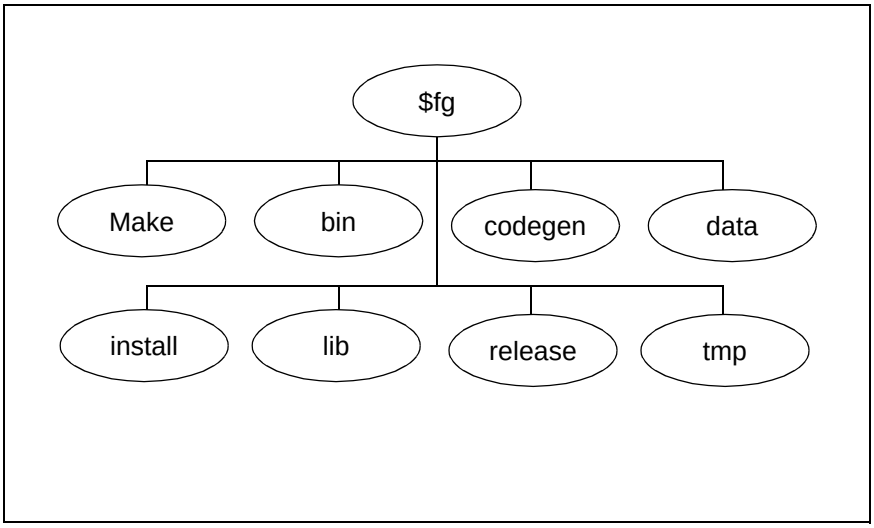
Follow installation instructions included with the Fitrix *Report* Code Generator diskettes.

Directory Structure

When you install the Fitrix *Report* Code Generator, a basic directory structure is created. This section introduces you to that structure. In addition, this section covers the preferred directory structure for developing report programs. In the following diagrams, an ellipse indicates a directory or system variable and a rectangle indicates a file or group of files.

Overview of the Directory Structure

The following diagram represents basic directory structure of the Fitrix *Report* Code Generator:



These directories form the basis of Fitrix *Report*:

$fg: This required variable points to the base directory for all Fitrix *CASE* Tools and applications. The `$fg` variable is typically set to `/usr/fourgen`.

Make: This directory contains all the files necessary to compile and link generated, 4GL code.

bin: This directory contains executable program files, such as `fg.report`.

codegen: This directory contains several code directories including `report.4gm` and `screen.4gm`. These directories contain 4GL code for the *Report* products and the Featurizer (`fglpp`).

data: This directory contains database directories. Database directories are required if you are using the INFORMIX-SE engine.

install: This directory contains installation files, such as `def` and `files` files.

lib: This directory contains library directories and files along with unload files and the library `dbmerge`.

release: This directory contains Fitrix *Report* Code Generator and CASE Tools release notes.

tmp: This directory contains the installation log (.log) files.

Optional Directory Variables

In addition to the required `$fg` variable, you can set a few other optional variables. These variables let you maintain your applications and the Tools themselves in separate base directories (other than `$fg`). These variables also give you the ability to install and use the new Tools on a system and set of applications without overwriting the old Tools.

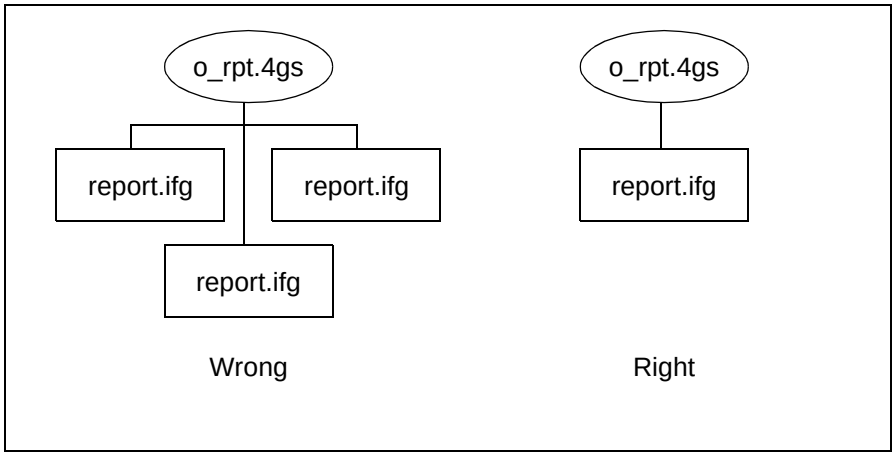
`$fgmakedir` If set, the `fg.make` script looks for make files in this directory rather than `$fg` (even though the local `Makefile` contains `$fg`).

`$fglibdir` If set, the `fg.make` script looks for upper-level libraries in this directory rather than `$fg`.

`$fgtooldir` If set, Tools executables, such as 4GL programs executed by calls to the *Report* Code Generator, are searched for in this directory rather than `$fg`.

Report Directory Structure

Fitrix *Report* Code Generator works best when each report program is kept in its own directory. Although it is possible to work with multiple reports in a directory, through clever file manipulation, placing one report in a single directory is preferred.

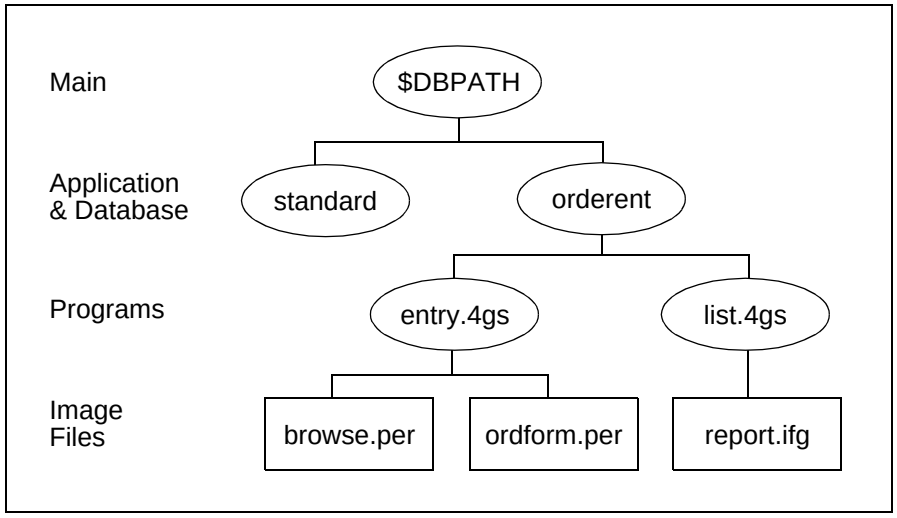


A good structure for organizing applications has a main directory that contains distinct application directories. Each application directory holds individual program directories, such as a report program directory. You should give the program directories names that reflect their content. By convention, program directories are given a .4gs extension (4GL Source). For example, a program directory that contains a report on monthly sales might be named `m_sales.4gs`.

In addition to application directories, the main directory can also hold a database directory. Within the database directory, table data and indices can reside. You can use the INFORMIX-4GL `$DBPATH` variable to set the path of your main directory.

As an example, consider an order-entry application built from the `standard` database. (The `standard` database comes with INFORMIX-4GL.) This simple application might contain only two programs: an order-entry program that logs customer orders, and a report program that lists orders made by each customer. The application directory could be named `orderent` and reside in the same directory as the `standard` database. The program directories might be named `entry.4gs` and `list.4gs`, respectively. Both program directories

would reside in the `orderent` application directory. Within the `list.4gs` directory could be the `report.ifg` file that is used to generate source code for the report program.



The `ordform.per` and `browse.per` files in the `order.4gs` directory are used by Fitrix *Screen* to generate the order-entry program.

2

Creating an Image File

The first step in building a report program involves creating an image file. An image file, or `report.ifg` file, contains shorthand commands and picture layouts of a report. The Fitrix *Report* Code Generator interprets these commands and layouts and produces thousands of lines of commented source code.

This section covers the following topics:

- Image File Description
- Additional Commands
- Image File Limitations
- Example Files

The First Step to Developing a Complete Report Program

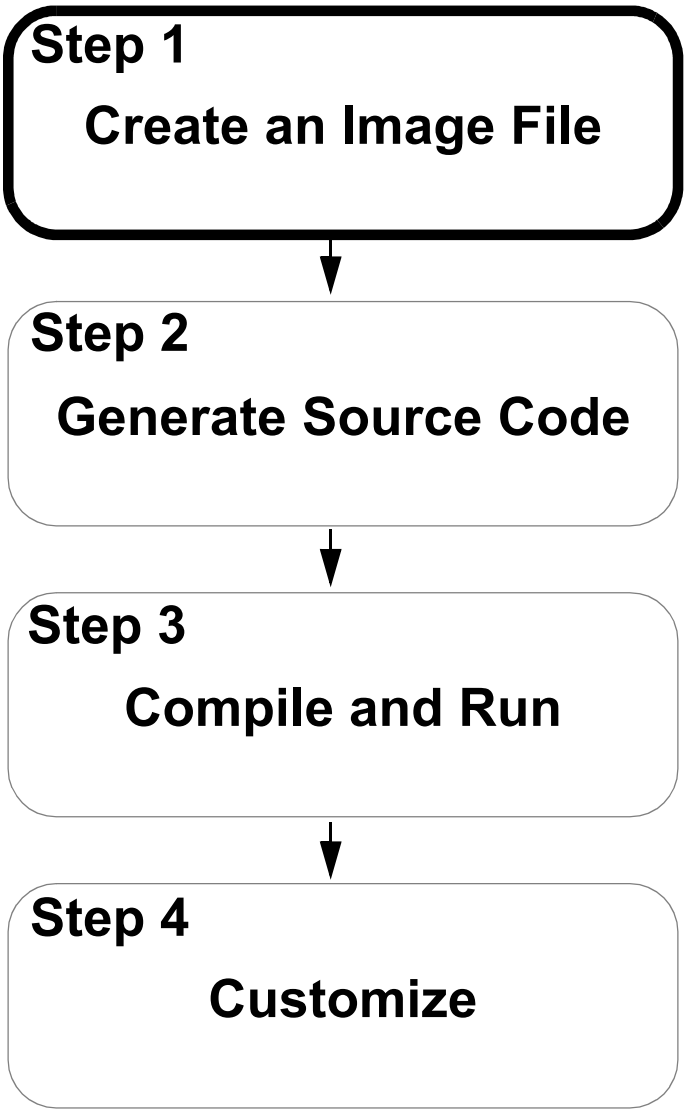


Image File Description

Every image file contains several sections. These sections specify what database the report uses, where the report prints to, the columns selected by the report, and many other report characteristics. Each section must follow a general syntax so the *Report* Code Generator can interpret the information in the section and produce source code.

Sections begin with a keyword followed by a statement or control block. In general, all image files, which are always given the name `report.ifg`, use the following syntax:

```
database section
[output section]

# format section
[page header control block]
[on first row control block]
[before group control block]
[on every row control block]
[after group control block]
[page trailer control block]
[last row control block]

attributes section
select section
[default section]
```

The format section differs from the other sections. In the format section, you visually arrange your report elements. You design page headers and footers, assign column labels, and set field widths. The format section is made up of control blocks. These blocks give your report structure.

The following pages outline all the image file sections in more detail.

Database

The `database` section specifies the database to use for the report. This section is placed at the top of the image file.

database *database-name*

database	a required keyword.
database-name	the name of the database you want to use in the report.

- You can only specify one database per report.
- If you specify a database that does not exist, the *Report* Code Generator produces source code, but an error occurs during compilation of the executable code.
- The following example specifies the `standard` database.

  database standard

- You can specify a different database when you run the report. For more information on specifying different databases at runtime, refer to "Starting the Code Generator" on page 3-3.

Output

The `output` section specifies the page length and margins of the report. If you don't specify page length and margins in the `output` section, the *Report* Code Generator uses default settings.

```
output
   [top margin         integer]
   [bottom margin      integer]
   [left margin        integer]
   [right margin       integer]
   [page length        integer]
```

output	a required keyword.
top margin	keywords that specify the number of lines in the top margin of the report. (Default setting is three lines.)
integer	an integer value that specifies the number of blank lines or columns in a margin or page length setting.
bottom margin	keywords that specify the number of lines in the bottom margin of the report. (Default setting is three lines.)
left margin	keywords that specify the number of columns in the left margin of the report. (Default setting is five columns.)
right margin	keywords that specify the number of columns between the left edge of the page and the start of the right margin. Report widths cannot exceed 255 columns. (Default setting is 132 columns.)
page length	keywords that specify the number of lines in one page of the report. (Default setting is 66 lines.)

Format Section

In the format section of an image file, you layout the graphical elements of your report. The format section consists of control blocks. Each control block handles a different portion of the report output. For example, the `page header` control block handles the information you want to display along the top of each page of your report. In all, there are eight control blocks. All eight control blocks are optional.

```
control-block
{
layout
}
```

Within the layout portion of the control blocks, you use special symbols to represent database columns and column formats.

[	Starts a column.
+	Centers column.
>	Right justifies column to right margin.
<	Left justifies column to one space from the preceding column.
{	Left justifies column to end of the preceding column.
]	Ends a column (for character type columns).
[!	Starts dynamic header/footer (doesn't print if no rows in the group are empty).
[*	Starts dynamic data line (doesn't print if all columns on the line are null).

Page Header

The `page header` control block specifies the report page header. This control block typically contains report values such as date of printing, page number, and report title. You can also add custom constant values and banners. The page header prints immediately after the top margin.

```
page header
{
layout
}
```

`page header`	required keywords.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the `attributes` section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

On First Row

The `on first row` control block works exactly like the `page header` control block, but it only prints on the first page of your report. For instance, an `on first row` control block might contain the report title and date.

```
on first row
{
layout
}
```

`on first row`	required keywords.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the `attributes` section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

Before Group

The `before group` control block contains information you want to print prior to the detail portion of your report. In addition, before group control blocks contain the columns you want to group your data by.

```
before group of table.column
{
layout
}
```

`before group of`	required keywords.
`table.column`	the database column name the data are grouped by.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the `attributes` section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

- You can define up to eight `before group` control blocks.

On Every Row

The `on every row` control block contains the detail portion of your report. In this control block, most of your data are printed. This control block also contains column labels and other graphical aides that help you align and decipher your data.

```
on every row
{
layout
}
```

`on every row`	required keywords.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the attributes section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

After Group

The `after group` control block contains information you want to print after the detail portion of your report. In many cases, the `after group` control block contains subtotal values and calculations.

```
after group of table.column
{
layout
}
```

`after group of`	required keywords.
`table.column`	the database column name the data are grouped by.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the `attributes` section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

- You can define up to eight `after group` control blocks.
- You can calculate subtotals on up to ten columns.

Page Trailer

The `page trailer` control block contains information you want to print on the bottom of the page. This block is similar to the `page header` and `on first row` control blocks. A typical `page trailer` contains report values such as time of printing and page numbers.

```
page trailer
{
layout
}
```

`page trailer`	required keywords.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the `attributes` section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

On Last Row

The `on last row` control block contains information you want to print at the end of your report. This control block is a good location for summary information and grand total values.

```
on last row
{
layout
}
```

`on last row`	required keywords.
`{`	a symbol that specifies start of control block.
`layout`	one of the following items.
`special-symbol`	a symbol that represents a database element or field format.
`field-tag`	a variable that is defined in the attributes section.
`graphical-element`	any keyboard character you want to appear on your report.
`}`	a symbol that specifies end of control block.

- You can calculate totals on up to ten columns.

Attributes

The `attributes` section defines the field tags in an image file. A field tag identifies which columns, subtotal values, grand total values, math formulas, constant values, runtime values, and hidden columns the report uses. In addition, the attributes section controls how values and data are represented.

```
attributes
    field-tag = assignment [,modifier]
   [field-tag = ...]
```

attributes	a required keyword.
field-tag	one or more identifiers of a report value, column, or formula.
assignment	one of the following values.
table.column	a column in the database.
formonly.column **type** ***data-type***	a form-only column using any valid data type.
[sum|avg|min|max] **(*table.column*)**	an aggregate function that takes a database column as its argument.
[sum|avg|min|max] **(*formonly.column*)** **type** ***data-type***	an aggregate function that takes a form-only column using any valid data type as its argument.
constant "*string*"	a non-varying element or value.
date	a runtime value that specifies date.
time	a runtime value that specifies time.
count	a runtime value that specifies item count.
pageno	a runtime value that specifies page number.
lineno	a runtime value that specifies line number.

modifier	one of the following modifiers.
using "*string*"	a modifier that specifies a format for a `money` or `date` expression.
upshift	a modifier that converts character data to uppercase.
downshift	a modifier that converts character data to lowercase.
updown	a modifier that formats first character in uppercase and following characters in lowercase.

- Only one modifier may be used for each field tag definition.
- Constants cannot use any modifiers. The runtime values (`date`, `time`, `count`, `pageno`, and `lineno`) can take the `using` modifier, but not the `upshift`, `downshift`, or `updown` modifiers.
- Column names must be unique. A `formonly` column should not have the same name as a database column used in the report. (If the *Report* Code Generator finds duplicate column names, it uses only the first one. Thus if your report contains `customer.name` and `order.name` (and they are not join columns), only `customer.name` is used.
- The `updown` modifier is not a standard Informix data type. Columns that use this modifier print the first letter of a word in uppercase and the remaining letters in lowercase, with the exception of letters following Mc. The following shows some examples of the `updown` modifier:

```
Guido Molinari
Pete Obrien
Pete O'Brien
Odibbe McDowell
Ronald Macdonald
```

Select

The `select` section creates a cursor that selects and arranges the data in the report. In the `select` section, you define which data get selected by specifying the tables, joins, columns, and filter the report uses. You also decide how the selected data are sorted and grouped. Every image file must contain a `select` section.

```
select
   [more   = table.column]
    tables = table [, outer table...]
   [join   = table.column = table.column]
   [filter = criteria]
    order  = table.column [, table.column...]
```

select	a required keyword.
more	an optional keyword that defines columns not included on the report but needed in the `select` statement.
table.column	a column in the database.
tables	a keyword that defines the tables used by the report.
table	a database table.
outer table	a database table that is linked by an outer join.
join	a keyword that defines criteria for selecting the rows from the named table.
filter	a keyword that defines the selection criteria for the report.
criteria	an Informix selection criteria expression.
order	a keyword that specifies the columns to sort by.

- If the report uses multiple `more` statements, each `more` must be on a separate line. For example:

```
more = stxckrgd.doc_no
more = stxckrgd.reconciled
more = stxchrtr.incr_with_crdt
```

- To name tables that have an outer join to your report's main table, use the SQL modifier `outer`. For example:

```
tables = stxckrgd, outer stxhrtr
```

Defaults

The `defaults` section specifies miscellaneous report information, such as messages that appear while the report is running and the destination of the report output.

```
defaults
   [progname = program-name]
   [prcname = string]
   [rtmargin = string]
   [destin = destination]
   [quiet = integer]
   [prc_only = ]
   [allow_int = ]
```

defaults	a required keyword.
progname	a keyword that specifies the executable program name for the report.
program-name	a name for the executable report program.
prcname	a keyword that specifies the character string that is displayed in the upper left portion of the screen while the report is running.
string	a character string (i.e., word) that is displayed on the screen.
rtmargin	a keyword that specifies the character string that is displayed in the upper right portion of the screen while the report is running.
destin	a keyword that specifies the output destination of the report.
destination	one of the following output destinations.
screen	a keyword that displays output to the screen.

printer	the name of the printer that receives the report output.
file	the name of a file that receives the report output.
| ***program***	the name of a program that the report output is *piped* to.
quiet	a keyword that defines how many records the Code Generator processes before updating the count on the screen.
integer	an integer value that specifies the number of records between update intervals.
prc_only =	the processing only statement that tells the Code Generator not to print the report output. When set to Y, the *Report* Code Generator processes the report but does not print it. When set to N, the *Report* Code Generator processes and prints the report.
allow_int =	the allow interrupt statement prevents interrupt signals from halting the report process. When set to Y, interrupt statements can halt the report process. When set to N, report processes cannot be interrupted.

Additional Commands

You can use three other commands in an image file. These commands control pagination and report output.

page: The `page` command creates a page break in your report. You can place this command before a `before group` or after an `after group` control block in the format section. For example, to create a page break immediately following the `after group` control block, place the `page` command following the right brace:

```
after group of customer.customer_num
{
Subtotals for [A9       ]            ----------
                                     [B1       ]
}
page
```

separate: The `separate` command splits output over two pages. By default, three control blocks (`before group`, `after group`, and `on last row`) always print on a new page if the current page does not contain enough space. The `separate` command tells the *Report* Code Generator to split the output of these control blocks across two pages. The separate command goes above the left brace in the `before group`, `after group`, and `on last row` control blocks.

pause: The `pause` command creates a paging prompt for the report program. This command only works for reports that print to the screen. After printing a page of output, a paging prompt appears:

```
Press [ENTER] to continue:
```

When the you press [ENTER], the program prints a second page of output to the screen.

Image File Limitations

Because the Code Generator must interpret image files of varying size and content, a few limitations exist. When creating an image file, regard the following restrictions:

- Image files cannot define reports wider than 255 characters.
- Image files cannot exceed 200 lines.
- Tabs are not allowed in image files.
- Image files cannot contain mathematical expressions. Create math logic in extension (.ext) files and merge them into your source code. For more information on extension files, refer to "Extension (.ext) Files" on page 5-10.
- Image files must contain a `database`, `attributes`, and `select` section. The other sections are optional.
- Image files must always be named `report.ifg`.

Example Files

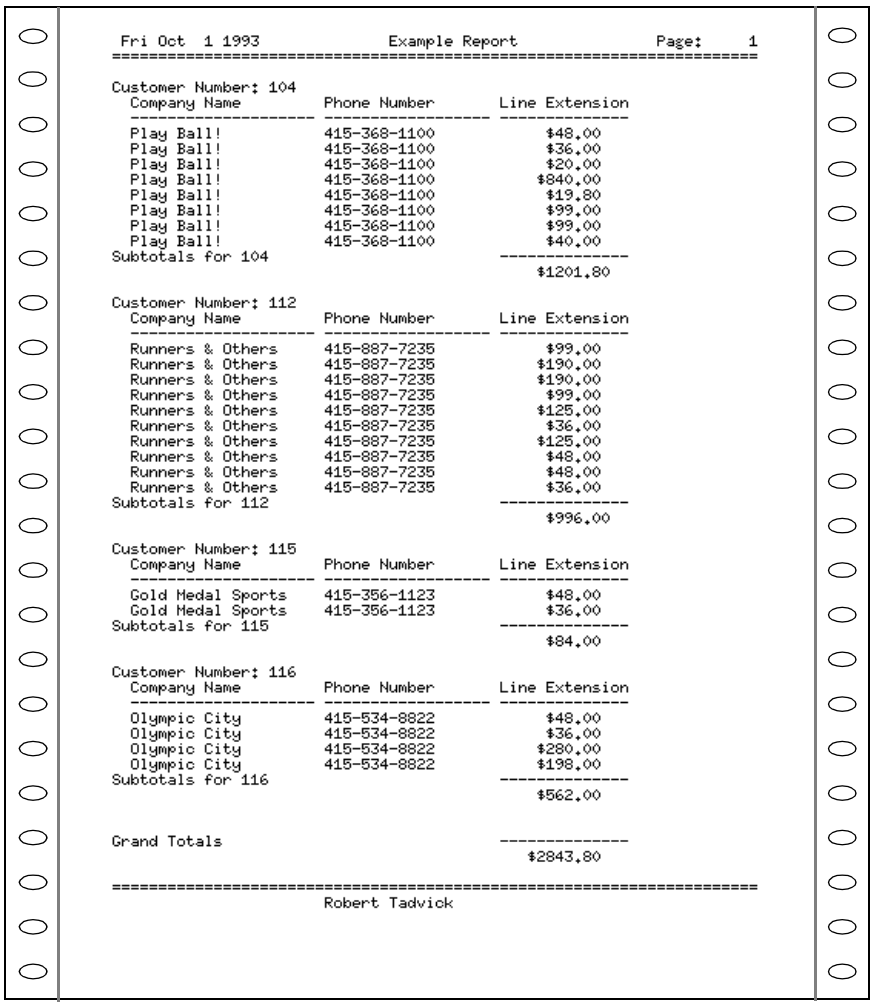
The following pages show an image (`report.ifg`) file and sample report output. Refer to these examples when you are creating your own image files. You can find additional examples in "Report Examples" on page 8-1.

Sample Image File

```
database standard

output
    top margin       3
    bottom margin    3
    left margin      5
    right margin    75
    page length     66

page header
{
<A1    ]           +A2                          ]          Page: >A3 ]
=========================================================================
}

on first row
{
                   [A4                          ]
=========================================================================
}

before group of customer.customer_num
page
{

Customer Number: [A5       ]
  Company Name         Phone Number       Line Extension
  -------------------- ------------------ --------------
}

on every row
{
  [A6                ] [A7              ] [A8          ] [*
}

after group of customer.customer_num
{
Subtotals for [A9       ]                 --------------
                                          [B1          ]
}

on last row
{

Grand Totals                              --------------
                                          [B2          ]
}

page trailer
{
=========================================================================
                     [B3                 ]
}

attributes
    A1 = today, name=HD_date
    A2 = constant "Example Report", name=HD_title
    A3 = pageno using "<<<<<", name=HD_page
    A4 = constant "Example Report", name=FR_title
    A5 = customer.customer_num using "<<<<<<<<<<<", name=BF1_customer.customer_num
    A6 = customer.company, name=customer.company
    A7 = customer.phone, name=customer.phone
    A8 = items.total_price, name=items.total_price, subt=Y
    A9 = customer.customer_num using "<<<<<<<<<<<", name=AF_1_customer.customer_num
```

```
    B1 = sum(items.total_price), name=SUBT_AF_1_items.total_price
    B2 = sum(items.total_price), name=TOT_LR_items.total_price
    B3 = constant "", name=TR_user

select
    tables = customer, orders, items
    join   = items.order_num = orders.order_num and orders.customer_num = customer.
customer_num
    order  = customer.customer_num

defaults
    progname  = brianh
    prcname   = Example
    destin    = report.out
```

Sample Report Output



```
 Fri Oct  1 1993              Example Report                Page:     1
=====================================================================

Customer Number: 104
  Company Name         Phone Number       Line Extension
  -------------------- ------------------ --------------
  Play Ball!           415-368-1100            $48.00
  Play Ball!           415-368-1100            $36.00
  Play Ball!           415-368-1100            $20.00
  Play Ball!           415-368-1100           $840.00
  Play Ball!           415-368-1100            $19.80
  Play Ball!           415-368-1100            $99.00
  Play Ball!           415-368-1100            $99.00
  Play Ball!           415-368-1100            $40.00
Subtotals for 104                           --------------
                                               $1201.80

Customer Number: 112
  Company Name         Phone Number       Line Extension
  -------------------- ------------------ --------------
  Runners & Others     415-887-7235            $99.00
  Runners & Others     415-887-7235           $190.00
  Runners & Others     415-887-7235           $190.00
  Runners & Others     415-887-7235            $99.00
  Runners & Others     415-887-7235           $125.00
  Runners & Others     415-887-7235            $36.00
  Runners & Others     415-887-7235           $125.00
  Runners & Others     415-887-7235            $48.00
  Runners & Others     415-887-7235            $48.00
  Runners & Others     415-887-7235            $36.00
Subtotals for 112                           --------------
                                                $996.00

Customer Number: 115
  Company Name         Phone Number       Line Extension
  -------------------- ------------------ --------------
  Gold Medal Sports    415-356-1123            $48.00
  Gold Medal Sports    415-356-1123            $36.00
Subtotals for 115                           --------------
                                                 $84.00

Customer Number: 116
  Company Name         Phone Number       Line Extension
  -------------------- ------------------ --------------
  Olympic City         415-534-8822            $48.00
  Olympic City         415-534-8822            $36.00
  Olympic City         415-534-8822           $280.00
  Olympic City         415-534-8822           $198.00
Subtotals for 116                           --------------
                                                $562.00


Grand Totals                                --------------
                                               $2843.80

=====================================================================
                       Robert Tadvick
```

3

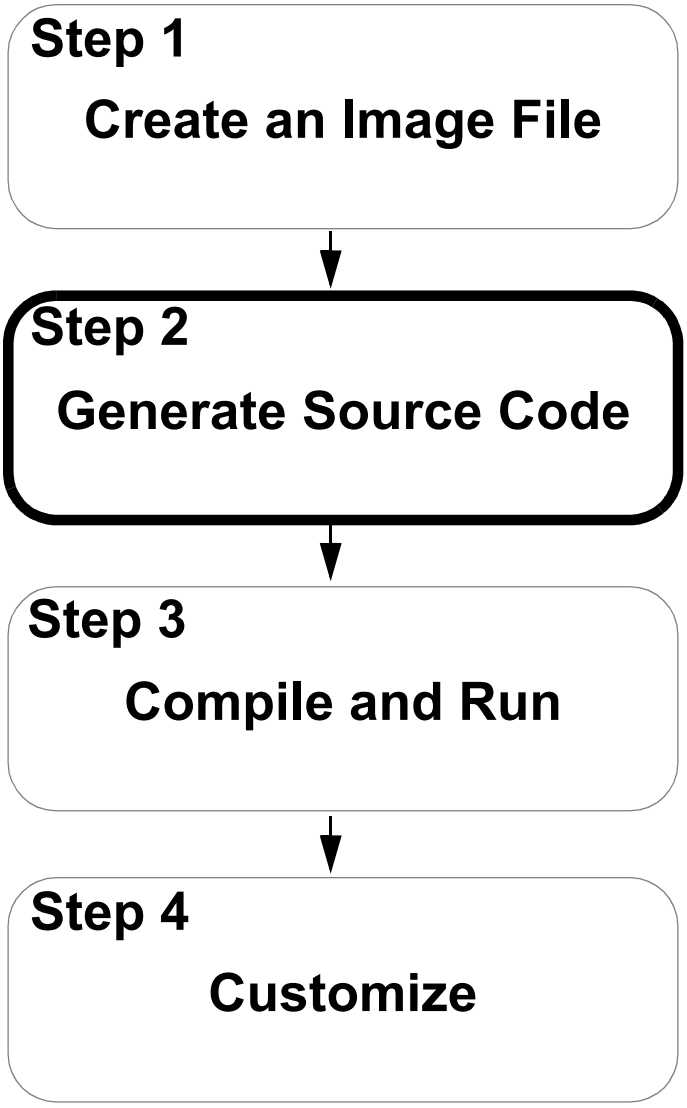
Generating Source Code

Once you create an image file, you are ready to run the *Report* Code Generator. The Code Generator creates the source code for your report program. To run the Code Generator, you use the `fg.report` command. The *Report* Code Generator then takes your image file and creates five source code files and a `Makefile`.

This chapter covers the following topics:

- Starting the Code Generator
- Handling Duplicate Files
- Reviewing the Source Code Files

The Second Step to Developing a Complete Report Program



Starting the Code Generator

You use the `fg.report` command to initiate the *Report* Code Generator. The Code Generator takes your `report.ifg` file and creates five source code files and a `Makefile`. From these files, you can compile an executable report program.

`fg.report [-fg] [-dbname database] [-r "report"] [-s "selection-set"] [-f] [-o n] [-yes|-y|yes|-no|-n|no]`

`fg.report`	Initiates the *Report* Code Generator.
`-fg`	Links in the libraries needed by reports that run with Fitrix Accounting modules. Use this optional flag when generating reports for Fitrix Accounting modules.
`-dbname database`	Specifies a different database then the database specified in the `report.ifg` file. If no database is set in the `report.ifg` file and the `-dbname` flag is not used, the Code Generator defaults to the `standard` database.
`-r "report"`	Starts the Image Maker and creates a `report.ifg` file from a Fitrix *Report* Writer report. From this `report.ifg` file, the *Report* Code Generator creates source code for a report program. You must place the report name inside quotation marks.
`-s "selection-set"`	Changes the selection set used by the *Report* Writer report. You must place the selection set name inside quotation marks. Use the `-s` flag in conjunction with the `-r` flag.
`-o n`	Changes level of screen output. Where `n` is a value between 1 and 5. Use `-o 1` to limit screen output and `-o 5` to show all screen output.

`-f`	Suppresses screen output while the Code Generator creates source code. This flag speeds up the code generation process. This flag is synonymous with `-o 1`.
`-yes|-y|yes|-no|-n|no`	Answers Duplicate Files message (see "Handling Duplicate Files" on page 3-5). The `yes` flag specifies option 1 and the `no` flag specifies option 3.

- All the flags associated with `fg.report` are optional flags. However, some of these flags give your reports tremendous flexibility. From a single `report.ifg` file, you can create source code that uses different databases and selection sets.

Handling Duplicate Files

Before the Code Generator places source code files or the `Makefile` into the current directory, it checks the directory for existing report files. You should develop report programs in their own directory; refer to "Report Directory Structure" on page 1-7 for more information. Existing report files are usually the result of a report you created previously. Sometimes these existing reports contain custom work that you do not want destroyed. If source code files already exist, the Code Generator provides a list of options. For example, if a `Makefile` already exists, the following message and options appear:

```
There currently exists a file called: Makefile

Would you like me to:

  1) Overwrite Makefile
  2) Append the new Makefile to the existing Makefile
  3) Move Makefile to Makefile.old
  4) Write to Makefile.new
  5) Don't write Makefile at all, or
  6) Exit program

  (If you wish to create file.diff, type
  a 'd' after the selection. example: 2d)

  Enter Selection:
```

A similar list of options appears for all existing source code files. Use the following table to decide which option you want to choose.

Option	Result
1	Specifies the overwrite option; the Code Generator replaces the old version of the file with the new version.
2	Specifies the append option; the Code Generator appends the new file to the end of the old file.
3	Specifies the move option; the Code Generator adds the `.old` extension to the existing file and writes the new file.
4	Specifies the write option; the Code Generator leaves the existing file alone and writes the new file with a `.new` extension.
5	Specifies the don't write option; the Code Generator skips the creation of this file and proceeds to the next file.

Option	Result
`6`	Specifies the exit option; the Code Generator exits the source code generation process without writing any more files.
`d`	Creates a file that shows the differences between the old file and the new one. You can use the `d` option in conjunction with the other options. For example, if you enter `4d` the old version does not change, the `.new` extension is added to the new file, and a `.diff` file is created. The `.diff` file shows the differences between the old and new files.

Reviewing the Source Code Files

Once the *Report* Code Generator completes creating source code, six newly generated files appear in your current directory. Five of these files are source code files, which are given a .4gl extension. The other is a file called the `Makefile`.

File	Contents
`globals.4gl`	contains global record definitions. These definitions include report record, cursor current record, cursor next record, and control record. In addition, this file contains a library communications area. This area holds global variables that communicate with library functions.
`lowlevel.4gl`	contains control block functions that handle data retrieval, such as `before group`, `on every row`, and `after group`.
`main.4gl`	contains error handling and program initialization and termination logic. As well, this file contains the logo function.
`midlevel.4gl`	contains data selection and filter logic in addition to cursor preparation and page break functions. For example, you can find the functions `ml_join()`, `ml_filter()`, and `ml_order` in this file.
`report.4gl`	contains page layout and format information. This file specifies placement of column data and labels in addition to header, footer, and margin locations.
`Makefile`	references UNIX `make` utility in `$fg/Make`. In addition, the `OBJFILES` line shows which object files are linked, and the `LIBFILES` line shows library search precedence. For more on the `Makefile`, refer to "The Makefile" on page 4-12

4

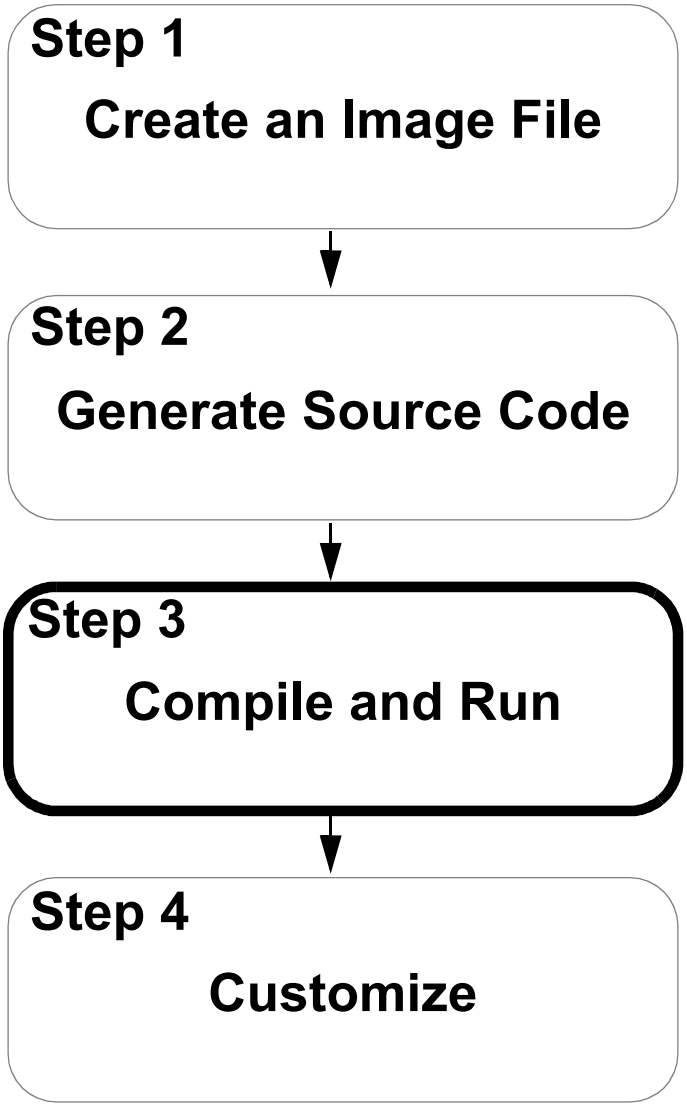
Compiling and Running

After code generation, the next step to creating a report program involves compiling the source code. Source code compilation creates a working report program. This section outlines the compilation process and the method for initiating report programs.

This section covers the following topics:

- Compiling generated code
- Using `fg.make` to compile
- Compiling and linking libraries
- Compiling your entire application
- Executing the final program

The Third Step to Developing a Complete Report Program



Compiling Generated Code

Compiling code means turning 4GL source code into a working program. Fitrix *Report* Code Generator provides the facilities to do this for a single program or for an entire set of programs.

The script for compiling your 4GL source code is `fg.make`. This script can compile individual programs, all the programs in a module, or even an entire application. If you are using the INFORMIX-4GL Rapid Development System, `fg.make` compiles programs into pseudo-code (called *p-code)* object files. If you are using the INFORMIX-4GL C Compiler Version, `fg.make` compiles programs into C source code. Refer to "Differences Between RDS and C Compiles" on page 4-4 for more on the two compile versions.

If you have both Informix products (the Rapid Development System and the C Compiler Version) on your system, `fg.make` assumes you want to use the Rapid Development System. You can, however, override this behavior. Add the `-F` flag to the `fg.make` script (e.g., `fg.make -F`). This flag forces `fg.make` to use the C compiler.

Depending on your current directory, `fg.make` completes the following tasks:

- At the application directory, `fg.make` compiles each module listed in the application `Makefile`.
- At the module directory, `fg.make` compiles each library and program listed in the module `Makefile`.
- At the library directory, `fg.make`:

  1. Converts form source (.per) files to form (.frm) files. Form source (.per) files and form (.frm) files are used by the Fitrix *Screen* Code Generator. If you have purchased Fitrix *Screen* products, you can learn more about these files in the Fitrix Screen *Technical Reference*.
  2. Converts source (.4gl) files to object (.4go or .o) files.
  3. Loads object files into the archive (.a file or .RDS directory).
  4. Removes the object files produced in step two.

- At the program directory, `fg.make`:

1. Merges extension (.ext) files with original (.org) files to produce source (.4gl) files. For more on extension files, refer to "Extension (.ext) Files" on page 5-10.
2. Converts form source (.per) files to form (.frm) files;
3. Converts source (.4gl) files to object (.4go or .o) files;
4. Links object files with objects in a library archive file. These archive files are listed in the program `Makefile`. This final step produces the program (.4gi or .4ge) file.

Note

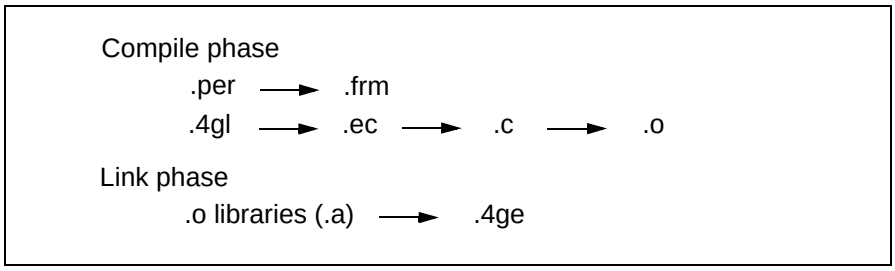
The `fg.make` script requires the standard UNIX `make` utility. This utility determines which files are compiled. If your machine lacks this utility, you must copy it from a machine that has it. The `make` utility is usually located in `/bin/make`.

Differences Between RDS and C Compiles

You can compile INFORMIX-4GL source code into two different forms: a binary executable (machine specific) form (.4ge), or a pseudo-code form (.4gi) that is interpreted by a *runner* program (`fglgo`). The first form uses the C Compiler Version and the second form uses the Rapid Development System. These forms are known as *C compile* and *RDS compile*, respectively.

During C compile, 4GL source code (.4gl) files go through several transformations. The first transformation uses `fglpc`, an Informix ESQL/C program, which converts source code files into ESQL/C (.ec) files. These ESQL/C files are then transformed into pure C code (.c) files. At this point, compilation is turned over to `cc`, the UNIX C compiler on your system. It produces object (.o) files. Finally, the UNIX C compiler runs `ld`, the UNIX linker, which links object (.o) files to each other and to objects stored in a library archive file. This process produces a binary (.4ge) file that you can run directly.

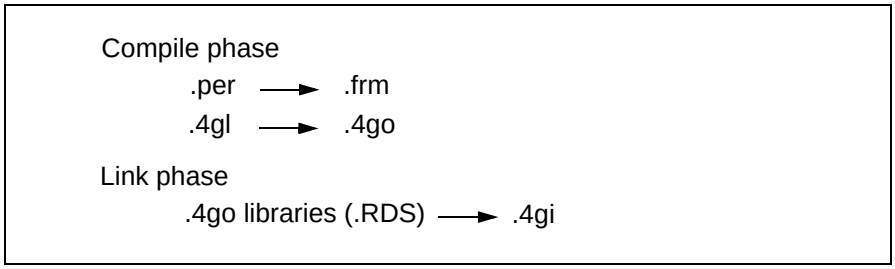
The following figure illustrates the C compile process.



An RDS compile differs from a C compile. Initially, the `fglpc` program transforms source code (.4gl) files into p-code object (.4go) files. These p-code object files are then concatenated using the UNIX `cat` command. Next the `link.rds` shell script is used. This script emulates `ld`, the UNIX linker. It searches library archives specified in the program `Makefile` and locates p-code object (.4go) files needed to complete the compile.

When `link.rds` is done, a p-code (.4gi) file exists. You can execute this p-code file with `fglgo`, the Informix runner program.

The following figure illustrates the RDS compile process.



Note

RDS is a tremendous developer's tool. It has a first class debugger (`fgldb`), which can interpret p-code (.4gi) files. In addition RDS compiles are quick and completely portable between machines. RDS also works well for your end users. It is an excellent idea to have RDS and the debugger on your users' and customers' systems.

Using fg.make to Compile Your Program

When you use the *Report* Code Generator to generate source code, a `Makefile` is created. The `fg.make` script uses that `Makefile`, which must be in your current directory. See "The Makefile" on page 4-12 for more information on the `Makefile`.

The `fg.make` script is not complicated. It has two purposes: to set up environment variables and to run the appropriate compilation program (C compile or RDS compile). The programs that do the compiling use environment variables to determine some of their actions. That means you can change the default behavior of `fg.make` by setting those variables in your own environment.

For example, if you have both RDS and C compiles on your system, `fg.make` assumes you want to use RDS. You can force a C compile by passing the `-F` flag to the `fg.make` script. This flag overrides the underlying environment variable, which is called `make_method`. By default, this variable is set to RDS, which corresponds to an RDS compile. When `-F` is used, you override this variable. If you always want to use a C compile, you can set `make_method` to 4GL, which corresponds to a C compile and the `-F` command flag.

```
fg.make [-h] [-F|-R] [-L library] [-M makefile]
[-T tags][-m {n|o|f|of}] [-o execname] [-l] [-f] [-D]
[-r] [-u] [-a] [-c] [args]
```

-h	Displays `fg.make` command flags and flag descriptions.
-F	Overrides `make_method` variable and performs a C compile. Its environment variable equivalent is `make_method=4GL`.
-R	Overrides `make_method` variable and performs an RDS compile. Its environment variable equivalent is `make_method=RDS`.
-L *library*	Lets you specify additional libraries you want `fg.make` to link. These libraries appear in the `Makefile` above the upper level libraries. Its environment variable equivalent is `xtra_lib=`*`library`*.

-M *makefile*	Allows you to specify a file other than `Makefile`. This flag is useful when you are testing.
-T *type*	Lets you specify which type of `Makefile` to create. You can create the following types: application, module, library, program, shell, and make.
-mn	Prevents `fg.make` from performing a merge. The Featurizer is not called. Its environment variable equivalent is `no_merge=y`.
-mo	Runs the Featurizer without a subsequent compilation. Its environment variable equivalent is `merge_only=y`.
-mf	Overrides time stamp comparison logic and forces the Featurizer to perform a merge. Its environment variable equivalent is `force_merge=y`.
-mfo	Forces the Featurizer to merge and override the time stamp comparison logic without compiling or linking. Its environment variable equivalent is `merge_only=y` and `force_merge=y`.
-o *execname*	Specifies name of the target library archives in library compiles (*`outname`*`.a` or *`outname`*`.RDS`). In program compiles, it specifies the name of the program file (*`outfile`*`.4ge` or *`outfile`*`.4gi`). This flag strips any extensions you might add to it. It is useful for testing.

-l

Instructs `fg.make` to link object files together into a program (.4gi) file, in an RDS compile. This flag works without checking for modifications between source (.4gl) and corresponding object (.4go) files. Its environment variable equivalent is `link_only=y`.

You can use `-l` when a local source (.4gl) file has been modified and compiled (with `fglpc`) into an object (.o) file, with the remainder of the application source code held constant.

The `-l` flag causes `fg.make` to skip `fglpc` and `form4gl` (i.e., skip the compilation of source (.4gl) files and form (.frm) files) and run only the link part of the `fg.make` suite of shell scripts. If you run `fg.make` in `link_only` mode, always rebuild the `filelist.RDS` in the local program directory. Also, if a library has been compiled, `fg.make` run in `link_only` mode in the local program directory rebuilds the `depend.RDS`, `func_map.RDS`, and `unresolved.RDS` files in the library. If the `link_only` option is used, the Featurizer is not run.

-f

Limits the work done by `fg.make` each time it is run for a program. The `link.rds` part of the compile creates a list of files that must be concatenated with the local object (.4go) files to create the program (.4gi) file (under RDS). That list is saved in the local directory under the name `filelist.RDS`. As long as no new calls to library functions have been added to the program being compiled, this list need not be recreated each time `fg.make` is run. The Featurizer is still run when `-f` is used. Its environment variable equivalent is `fast_link=y`.

-D

Creates a dependency list (`filelist.RDS`). The `-D` flag lets you rebuild your `filelist.RDS` without having to rebuild the program (.4gi) file. This flag works only with RDS compiles.

-r	Causes `link.rds` to make multiple passes through the library list when making a program so that functions are more likely to be resolved cleanly. This flag can cause the standard UNIX linker (`cc` and `ld`) to fail, depending on the capabilities of the platform linker. You should only use this option if a standard `make` correctly resolves your functions, and `fg.make` does not. Also note that this could allow you to write non-portable code. This flag has no meaning if `fast_link` has been specified. Its environment variable equivalent is `recursive_link=y`.
-u	Causes `link.rds` to warn the user of any function calls it was unable to resolve. This flag has no meaning if `fast_link` has been specified. Its environment variable equivalent is `list_unresolved=y`.
-a	Causes all files to be recompiled regardless of dependencies. Its environment variable equivalent is `no_use_make=y`.
-c	In the program directory, this flag causes `fg.make` to stop after it compiles the source code. It does not continue on to produce the program. In other words, this flags causes `fg.make` to skip the link phase. In the library directory, this flag causes `fg.make` to stop after it compiles the source code. It does not continue on to load the archive (either the .a file or the .RDS directory).
args	Objects (files) to compile. The default is the list of objects in the `Makefile`.

Many of these flags work together. Some are mutually exclusive. For example, consider `fg.make -mfo`. This command skips all compilation except the Featurizer block merge. Likewise, `fg.make -fl` skips the compile phase, goes right to the link (`-l`) phase, and uses the current list of library files (produced by the last link) rather than producing a new list (`-f`).

On the other hand, some flags cannot be used together. Specifying `-l` implies `-mn` and overrides the `-mo` and/or the `-mf` flags. Likewise, if `-R` and `-F` are specified, whichever one occurs last on the command line takes effect.

Just as an aside, you can list single-character flags together and `-m` flags together if they occur last on the line. For example, `fg.make -iurflmof` is acceptable, where `mof` is equivalent to `-mo` and `-mf`. The three-character flags must stand alone.

Speeding Application Compiling

When you make slight changes to a source (.4gl) file or an extension (.ext) file, you need not remerge, relink, and recompile your entire program. You can limit `fg.make` so that only your changes are recompiled. Thus saving you time and speeding up the recompilation process. These shortcuts, however, only apply if you are using RDS. C compiles cannot be expediated.

Changing a Source (.4gl) File

Altering a source (.4gl) file is the most frequent type of change. When you run `fg.make`, it checks all your source (.4gl) files to see which ones you have changed. This check takes up time. In addition, each time you run `fg.make`, a new list of library files is built. You can avoid these steps with the `fglpc` command. This command lets you specify the source (.4gl) file you want to update, assuming you did not add new library function calls. Since you know what source files you have changed, you can use two commands to create a working application. First, use the `fglpc` command and the source (.4gl) file you have changed.

`fglpc` *`filename`*`.4gl`

The `fglpc` command recompiles your source (.4gl) file into an object (.4go) file. You can then use `fg.make` with the `-fl` flag to merge your new object (.4go) file.

`fg.make -fl`

The `-fl` is the fast link flag. This flag puts together the compiled local programs and all the library programs. It assumes you have run a complete `fg.make` at some time in the past on your program to create a list of library files. It also assumes you haven't changed the `Makefile` to require a different set of libraries.

A Change to an Extension (.ext) File

After you make an extension (.ext) file, it must be merged with the generated source code. Several `fg.make` command flags affect extension file merging and do various other steps in the compilation process.

When you just want to merge an extension file, use the -mo flag.

`fg.make -mo`

This flag, which means merge only, simply merges extension files into source code. It only performs this single step.

Normally, `fg.make` does a timestamp comparison before merging an extension (.ext) file with a source (.4gl) file. This comparison governs when a merge is necessary. If the extension (.ext) file is newer than the source (.4gl) file, `fg.make` performs the merge. Consider the following two files:

```
-rw-rw-rw-   1 dona  informix     3777 Aug  6 11:05 main.4gl
-rw-rw-rw-   1 dona  informix      596 Aug  6 11:18 logo.ext
```

The extension file `logo.ext` is newer than the source file `main.4gl` (`11:18` opposed to `11:05`). In this case, `fg.make` merges the two files. Now consider a second example:

```
-rw-rw-rw-   1 dona  informix     3777 Aug  6 11:05 main.4gl
-rw-rw-rw-   1 dona  informix      596 Aug  6 11:00 logo.ext
```

This time `logo.ext` is older than `main.4gl` (`11:00` opposed to `11:05`). No merge is performed.

When you want to force a merge without respect to timestamps, use `-mfo`.

`fg.make -mfo`

All extension (.ext) files get merged, even those older than the source (.4gl) files.

When you want to force a merge and do all of the other compilation steps as well, use `-mf`, such as:

`fg.make -mf`

The `-mf` flag causes a forced merge and creates object files, links, and perfoms additional compilation tasks.

The Makefile

The `fg.make` script reads a description file that contains information to produce a program. By default this description file is called `Makefile`. The `Makefile` is created during the generation of source code.

Here is an example of a generated program `Makefile`:

```
#############################################################################
# Copyright (C) 1992 Your Company Name Here
# All rights reserved.
# Use, modification, duplication, and/or distribution of this
# software is limited by the software license agreement.
# Sccsid: %Z% %M% %I% Delta: %G%
#############################################################################
# Makefile for an Informix report

#_type - Makefile type
TYPE = program

#_name - program name
NAME = tmp.4ge

#_objfiles - program files
OBJFILES = globals.o lowlevel.o main.o midlevel.o report.o

#_forms - perform files
FORMS =

#_libfiles - library list
LIBFILES = ../lib.a \
           $(fg)/lib/report.a \
           $(fg)/lib/user_ctl.a \
           $(fg)/lib/standard.a

#_globals - globals file
GLOBAL = globals.4gl

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

#_all_rule - program compile rule
all:
        @echo "make: Cannot use make. Use fg.make -F for 4GL compile."
```

This example `Makefile` contains six macros or variables and a rule. Each these elements may be different for your program, depending on the reports you are building, and whether you have additional source files. The follwoing list describes the elements in the example Makefile on the previous page:

TYPE: This macro contains the type of `Makefile`. A `Makefile` can be one of six types: program, library, application, module, shell, or make.

NAME: This macro contains the name of the compiled program. For an RDS compile, `fg.make` converts the program extension to .4gi.

OBJFILES: This macro contains a list of local object files. These files become linked together to create the compiled program.

FORMS: This macro contains a list of .frm files used by the program. These files are created from form specification (.per) files, which are associated with the Fitrix *Screen* Code Generator, for more on .per and .frm file, refer to the Fitrix Screen *Technical Reference Manual*.

LIBFILES: This macro lists the names of the library archives to search to resolve function calls. For an RDS compile, `fg.make` converts the extensions to .RDS.

Note

When doing an RDS compile, `fg.make` produces a list of the object files that it has resolved from the libraries. This list, `filelist.RDS`, can be reused in later compiles by specifying the `-f` flag with `fg.make`. You can only use this flag, which results in a faster compile, when no new function calls have been added to the code.

GLOBAL: This macro contains an entry for `globals.4gl`. All local object files depend on the `globals.4gl` file.

all: This `make` rule informs you not to use the UNIX `make` utility. If a user attempts to use the `make` utility, the following message appears:

```
make: Cannot use make. Use fg.make -F for 4GL compile.
```

Changing the LIBFILES Macro with Block Commands

You can use block commands to alter the `LIBFILES` macro. In an extension (.ext) file, these lines use the brute force method. For example:

```
########################################################################
start file "Makefile"
########################################################################
  replace_block TOF NUL from "LIBFILES" thru "$(fg)/lib/standard"
   LIBFILES = ../libadv.a \
              ../lib.a \
              ../../all.4gm/libadv.a \
              $(fg)/lib/user_rpt.a \
              $(fg)/lib/scr.a \
              $(fg)/lib/standard.a \
              $(fg)/lib/user_ctl.a
              ;
```

Using the -L Flag to Link Custom Libraries

The block command method results in a physical change to the `Makefile`. The `fg.make` script provides a method for specifying additional libraries without actually changing the `Makefile`. This facility can be very useful if you wish to try out new features in a library but do not wish to make the change permanent.

For example, suppose you write some useful functions and put them in a custom library directory such as `$fg/lib/standard.cus`. You can physically change your `Makefile` using an extension file (discussed above), or you can include the custom library without physically changing your `Makefile`. To do so, use the `-L` flag with the `fg.make` script. The following line shows how to include `standard.cus` with the `-L` flag:

`fg.make -L standard.cus`

This command effectively acts as if you had changed the `LIBFILES` macro to look as follows:

```
LIBFILES = ../lib.a \
           $(fg)/lib/scr.a \
           $(fg)/lib/standardcus.a \
           $(fg)/lib/standard.a \
           $(fg)/lib/user_ctl.a
```

You can also specify more than one library at a time, for example:

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

The above line produces the same effect as changing `LIBFILES` to look like the example on the next page:

```
LIBFILES = ../lib.a \
           $(fg)/lib/scradv.a \
           $(fg)/lib/scr.a \
           $(fg)/lib/standardcus.a \
           $(fg)/lib/standard.a \
           $(fg)/lib/user_ctl.a
```

It is also possible to modify the pathname of a custom library, for instance:

`fg.make -L /usr/our_work/lib/standard.cus`

The above line produces the same effect as changing `LIBFILES` to look as follows:

```
LIBFILES = ../lib.a \
           $(fg)/lib/scr.a \
           /usr/our_work/lib/standardcus.a \
           $(fg)/lib/standard.a \
           $(fg)/lib/user_ctl.a
```

You can add new libraries to the end, but do not use a period, such as:

```
fg.make -L newguy
```

This command affects `LIBFILES` as follows:

```
LIBFILES = ../lib.a \
           $(fg)/lib/scr.a \
           $(fg)/lib/standard.a \
           $(fg)/lib/user_ctl.a \
           newguy.a
```

If your `LIBFILES` macro is already customized to look like this:

```
LIBFILES = ../lib.a \
           ../../all.4gm/lib.a \
           $(fg)/lib/scr.a \
           $(fg)/lib/standard.a \
           $(fg)/lib/user_ctl.a
```

and you need to insert a library in front of the second occurrence of `lib.a`, you can include more than the word `lib` in your prefix. Insert a question mark instead of the slash so `fg.make` does not interpret the slash to mean pathname. For example:

```
fg.make -L all.4gm?lib.adv
```

The above line produces the same effect as changing `LIBFILES` to look as follows:

```
LIBFILES = ../lib.a \
           ../../all.4gm/libadv.a \
           ../../all.4gm/lib.a \
           $(fg)/lib/scr.a \
           $(fg)/lib/standard.a \
           $(fg)/lib/user_ctl.a
```

The following list covers rules that govern the use of the `-L` flag:

- The argument prefix (portion before period) specifies where to insert the library.
- The argument suffix (portion following period) is part of the library name.
- A slash in the argument specifies a pathname. A slash does not affect where the name is inserted.

- A question mark can replace a slash, if the slash is needed as part of the insertion criteria.
- When there is no match, the library is put at the end with no change.

Compiling Libraries

Much of the RDS compile parallels the C compile. The source .4gl files are transformed into object files (.4go or .o), and non-local function calls are resolved by searching the library archives listed in the `LIBFILES` macro. But it's this last process that is, in fact, the most different between the RDS and C compiles.

There are two classes of libraries. One class consists of the `standard`, `user_ctl`, and `report` libraries, which provide the flow of control of generated programs and a number of specialized functions that provide features of these programs.

The other class of libraries are those that you maintain yourself for common functions that are used by more than one of your programs or that modify the behavior of functions.

The `fg.make` script is used to maintain both classes the same way, but it is not advisable to make changes to the supplied functions. Your changes are lost when you install the next release. It is possible to add or change these functions by creating your own libraries.

To create your own library, there are two things you must consider: where it is physically located, and what sequence it is linked into your program.

Consider a library of functions that are common to a family of programs. You have a program source directory for each program. If you review the example `Makefile`, note that the first entry in `LIBFILES` is `../lib.a`. The `lib.4gs` file contains the source for this archive.

Creating the Library Archive

A library archive contains the compiled objects and catalogs used for linking your programs. A 4GL archive is a file with an extension of .a. An RDS archive is a directory with an extension of .RDS. A 4GL archive is created with the UNIX `ar` utility and its catalogs are stored internally. The RDS archive is created by `fg.make` directly and its catalogs are stored as files in the archive.

To create a library archive, you must have a `Makefile` in your library. The following shows an example of a library `Makefile`:

```
########################################################################
# Copyright (C) 1993 Your Company Name
# All rights reserved.
# Use, modification, duplication, and/or distribution of this
# software is limited by the software license agreement.
# Sccsid: %Z% %M% %I% Delta: %G%
########################################################################

# Makefile for an Informix function library

TYPE = library

LIBFILES = \
    $(LIB)(funtion-name1.o) \
    $(LIB)(function-name2.o)\
    $(LIB)(function-name3.o)\
    $(LIB)(function-name4.o)

FORMS=

LIB=../report.a

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

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

As you can see, a library `Makefile` contains four sections and a rule. The following list describes each element in a library `Makefile`:

TYPE: This macro indicates the type of the `Makefile`, which in this case is library. There are six types of which library is one.

LIBFILES: This macro contains the object files that are put in the library archive.

FORMS: This macro lists the .frm files used by the library functions, in the above case there are no .frm files used.

LIB: This macro contains the name of the library archive. It does not have to match the name of the library source directory. For example, if you create a library to hold custom functions from the `$fg/lib/report.4gs` directory, there is a convention for doing so: You create your library source directory as `$fg/lib/report.cus`, and you make the `LIB` macro in your `Makefile` look like the following line:

```
LIB = ../reportcus.a
```

This strategy allows you to use the `-L` flag when compiling programs with `fg.make`. The command `fg.make -L report.cus` automatically links your custom library just before the `report` library.

It is also possible to use the same name in the `LIB` macro for different libraries. For example, your `Makefile` can contain the following line:

```
LIB = ../report.4gs
```

This line causes your objects to be loaded into the same archive as our software company's objects. Just remember you must recompile your library after a newFitrix installation.

For an RDS compile, `fg.make` converts the .a extension to .RDS.

To create the library archive, run `fg.make` in the library directory.

When `fg.make` does a 4GL compile, it creates .o files for the files listed in the `LIBFILES` macro from the corresponding .4gl files and loads them into the archive file in the directory. It creates the archive if it doesn't exist. The `$(LIB)` symbol is special to the `make` utility. It tells `make` that the modification date of the source file is checked against the object in the archive instead of against an actual .o object file, so the .o file is deleted once it is loaded into the archive.

When `fg.make` does an RDS compile, it creates .4go files rather than .o files. These files are then moved over to an archive directory. This directory is created if it does not exist. In addition, the .4gl files are copied to the archive directory.

There are two reasons for keeping the .4gl files in both the .4gs and the RDS. First, the .4gl source file is needed in the archive for the linking process. Second, it must be possible to continue to link functions from the archive while modifications are being made to the .4gl files in the .4gs directory.

In addition to the .4gl and .4go files in `lib.RDS`, there are four catalog files. These are `func_map.RDS`, `depend.RDS`, `unresolved.RDS`, and `resolved.RDS`.

- The **`func_map.RDS`** file is a list of all the functions in this directory. Next to the function name is the name of the file it is in. During the linking phase of a program RDS compile, `link.RDS` refers to this list to find the names of the files containing the "unresolved" functions it is searching for.
- The **`depend.RDS`** file is a list of all the files any file depends on. Once `link.RDS` has found the names of the files that will resolve functions for it, it must then find the names of any other files that the found ones also depend on.

- The **unresolved.RDS** file is a list of all the functions that were called by functions in lib.RDS but were not resolved there. Link.rds refers to this to find out what new function names it has to add to its list of unresolved functions before it goes on to the next library.
- The resolved.RDS file is a list of all the files and function calls that were resolved in this library.

These files must be rebuilt every time fg.make does an RDS compile in the library.

If you have modified a .4gl file in lib.4gs, normally you need to run fg.make in lib.4gs to compile it. But if your modification does not include changes to function names, or added, deleted, or changed function calls, it is not necessary to rebuild those .RDS files in the .RDS directory.

Compiling Your Entire Application

Consider organizing your programs in a hierarchy. The top level would be the application, the second level a module of that application, and the third would be the programs themselves. The following explains how to set up your hierarchy.

Create a directory for your entire application. It's recommended that you do this in the $fg directory, though that is by no means a requirement. The name for this directory isn't set by convention, so make the name something meaningful.

In your application directory, create directories for each of the modules in your application. The names for your module directories should have .4gm as an extension, but the prefix can be anything that you consider meaningful. Examples might be sales.4gm, rcvbls.4gm, inventory.4gm. Also, put the application Makefile in this directory.

Use this as a model for the example $fg/myapplication/Makefile:

```
########################################################################
# Makefile for an Informix Application
########################################################################
TYPE     =  application
APPL     =  myapplication
MODULES  =  sales rcvbls inventory
#-----------------------------------------------------------------------
all:
        @echo "make: Cannot use make. Use fg.make to compile."
```

To compile your entire application, type `fg.make` in the application directory. To compile only specific modules, give the module names as arguments (for example `fg.make sales rcvbles`).

Compiling a Module

Put your program directories in the module directory. The names of these program directories normally have a .4gs extension. Examples in `sales.4gm` might be `entry.4gs`, `invoice.4gs`, and `post.4gs`. Also, put the module `Makefile` in the module directory.

To compile your entire module, type `fg.make` in the module directory. The following page contains an example module `Makfile` that you can use as a model:

```
########################################################################
# Makefile for an Informix module
########################################################################
TYPE   = module

MODULE = sales.4gm

LIBS  = lib

PROGS = entry invoice post prog4 \
        prog5 prog6 prog7 and_so_on
#---------------------------------------------------------------------- #
all:
       @echo "make: Cannot use make. Use fg.make to compile."
```

Application and Module Compilation with $cust_path

When compiling at the module level, all program directories with an extension found in the `$cust_path` variable are compiled.

For example, if `invoice` is listed in the module `Makefile` and `$cust_path = bth:4gc:4gs`, then `invoice.4gs`, and `invoice.bth` are compiled if they exist. These directories are compiled in reverse order of their `$cust_path` listing.

Running Report Programs

As soon as source code has been compiled, it can be executed. There are a number of command line arguments that can be specified upon invocation. This section addresses these arguments, and explains the invocation of programs compiled with C and RDS. Later, the usage of the `run` UNIX shell script is explained.

Invoking Compiled Programs

The method of executing a program depends on the compilation method you used to compile the source code.

The INFORMIX-4GL C compile version compiles source (.4gl) files down to object (.o) files, which are then linked together into an executable (.4ge) file. This executable file can be invoked by simply typing its filename at the UNIX prompt.

The INFORMIX-4GL RDS compile version converts source into pseudo-code, which is stored in object code files (.o). The object files are linked together into a non-executable program file (.4gi).

A number of command line arguments can be used when invoking a program generated by the *Report* Code Generator.

---

fglgo *program-name*.4gi [-dbname *database*] [order "*order-by-clause*"] [filter "*filter-clause*"]

-dbname	Specifies the database to run against.
order	Specifies the order of initial selection.
filter	Limits the initial selection.

The database can be selected on the command line. For example, the following `fglgo` command specifies the `stores` database:

```
fglgo report.4gi -dbname stores
```

The name of the database must follow the `-dbname` argument.

Other command line arguments allow you to pass a filter clause and order by clause to the program. This controls the selection and order of report data.

You can define the initial filter for the selection of data by specifying the filter on the command line. For example, the following command only selects customer numbers higher than 100.

```
fglgo report.4gi filter "customer_num >100"
```

Note

The example above only works for an `integer` type field. If you want to select a string, you must quote the string.

You can also specify a command line argument to order the initial selection of documents. You can order by any column, though the columns must be in the main table. For example, the following line orders by the `po_num` column:

```
fglgo report.4gi order "po_num"
```

The column is sorted according to ASCII conventions.

5

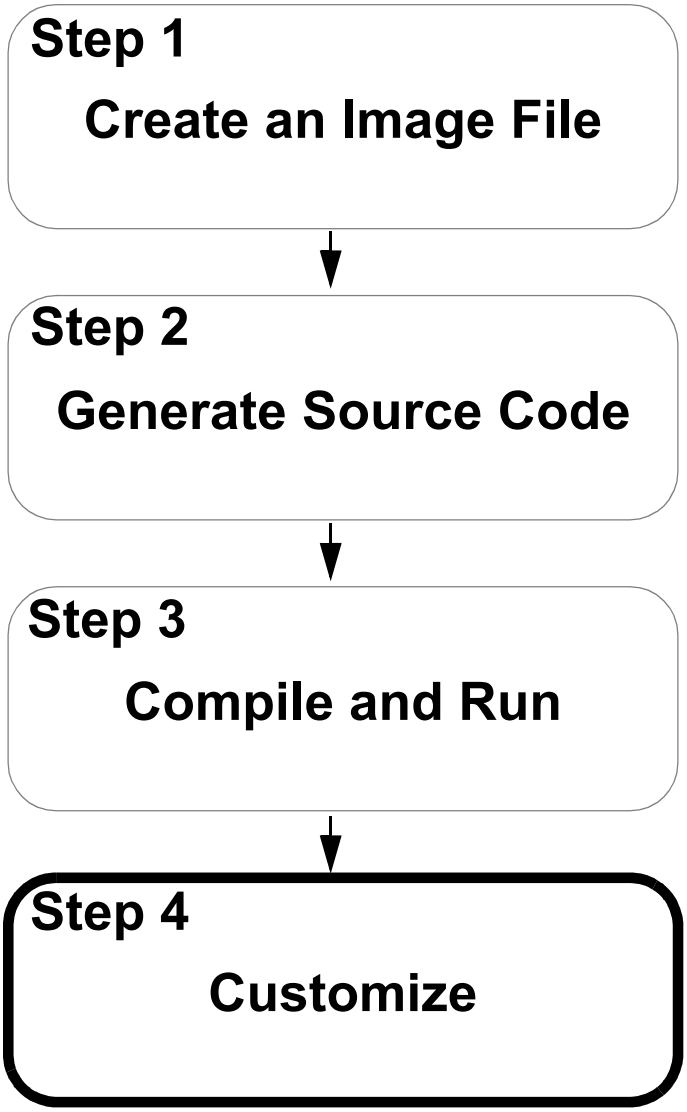
Customizing Reports

This section explains how to customize your applications while maintaining regenerability. Special files store all of your modifications in separate pieces known as *blocks*. How these blocks get merged into the source code with the Featurizer is also discussed.

This section covers the following topics:

- Featurizer Overview
- Block Commands Overview
- Pluggable Features and Feature Sets
- Pre-merged Generated Files
- Flow of the Featurizer

The Fourth Step to Developing a Complete Report Program



Featurizer Overview

The Featurizer merges custom modifications into INFORMIX-4GL source code produced by the Code Generator. The Featurizer pre-processes the source code (.4gl files created by the Code Generator) just before it is compiled (converted into object code). The Featurizer merges blocks and feature sets.

The following list outlines some key terms, concepts, and functionality of the Featurizer.

Regenerability: The ability of a code generation tool to re-create the base source code while maintaining custom modifications. For the application to be regenerable, any modifications done to the source code after initial generation must be applied to the new source code that has been regenerated. The Featurizer gives you true regenerability.

Source Code Blocks: Following programming conventions, source code can be divided into small chunks or blocks. A block is the definition of specific lines within a source code file. Blocks are denoted by block tags also called block IDs.

Our software provides a set of block commands that allow the insertion of new blocks, deletion of blocks, replacement of blocks, and alterations of lines within a block.

Note

By convention, blocks are defined as the physical lines of code that perform a logical function. Logical functions include initializing variables, checking validations, updating the disk, or any logical group of source code lines. Blocks should be separated by white space (blank lines), and they should be relatively small.

Custom Directories: Base .4gl programs should be stored in separate directories with the filename extension of .4gs. In order to maintain different versions of the same application on a system, a custom directory is created, and the differences in source code are stored in the custom directory. A generic custom directory extension is .4gc.

You may choose any three-character extension for custom directories. At runtime, setting your `$cust_key` environment variable to a custom extension runs the programs stored in that directory.

At pre-processing time, a custom directory search path is specified that merges source code and extensions from other directories. This allows you to store only the differences in a custom directory (vs. a copy of the original). When the original is changed, a re-compile in the custom directory brings forward changes from the other directories in the search path.

Pluggable Features: Logically, features are things that can be plugged in or unplugged based on the need for that feature. Physically, features are groups of source code extension (.ext) files throughout the application.

If a feature is installed (plugged in), that source code is applied to the application. If it is not installed (un-plugged), the source code for that feature is not merged into the final source code.

Organizing source code into features has several advantages. It allows for plug in/out functionality, it allows the application to have multiple versions, and it allows for the organization of source code for a particular unit of work into one area. This makes it very easy to identify the effect of a feature on the application.

Pluggable features can be used in different ways. In addition to the plug in/out functionality, they can be used to maintain different upgrade versions of the application, different customer requirements, product testing, etc.

Feature Sets: Simple groups of plug-in features. Since some features may be incompatible with other features, you may wish to group features into different sets that are known to work together. When compiling an application, you can specify which feature set to apply.

Feature set files, which are always given the name `base.set`, include a list of features in the order that they are applied to the source code.

Running the Featurizer

You can initiate the Featurizer in any one of three ways:

1. From `fg.make`.
2. During code generation.

3. Directly from the command line.

Invoking From the fg.make Utility

The most common way of running the Featurizer is through the `fg.make` compilation script. Each time you run `fg.make` to compile your programs, the Featurizer is automatically invoked and merges any necessary files into your program. Flags are available with `fg.make` to control whether you want to merge or not to merge blocks when calling `fg.make`. Refer to "Using fg.make to Compile Your Program" on page 4-6 for more information on the `fg.make` utility.

Invoking From the Code Generator

The Code Generator automatically creates the block tags in the generated code. After the code is generated, the Code Generator automatically invokes the Featurizer, which searches for and merges .ext files into the generated code.

Executing the Featurizer Directly

You can also run the Featurizer directly at the UNIX command line. The following lists the syntax for the `fglpp` command.

```
fglpp [-dbname database][-C][-force][-set filename.set]
[-yes][-trace][filename...]
```

-dbname *database*	Specifies the database name to use.
-C	Inserts comments into merged code noting origin of blocks.
-force	Ignores file date/time and forces pre-processing of files.
-set *filename*.set	Specifies the feature set file to use.
-yes	Overwrites files without write permission.
-trace	Displays `fglpp` activity to screen.

filename...	The file(s) to pre-process. If omitted, a list is built of the files that need pre-processing.

Block Commands Overview

To modify source code within a block, there are a set of block commands to indicate what you wish to do to that block. Block commands go into files with a .ext extension. The Featurizer reads the block commands in the .ext file and act on the specified block in the source code.

Here are some examples of simple block commands:

```
before block llh_add insert

after block llh_add serial

replace block llh_lookup not_found

delete block llh_lookup must_find
```

A block command takes two arguments:

1. The function name that contains the block.
2. The name of the block (called the *block name* or *block ID*).

Using Block Commands to Manipulate Code

The following are some block command examples to help give you an idea of what block commands are and how they work. Using the `llh_add` example from the previous section, say you want to place one extra line before the `insert` command:

```
let m_orders.entry_date = today
```

Again, here is the unmodified `llh_add` function:

```
###############################################################################
function llh_add()
###############################################################################
# This function inserts data into the header table.
#
    #_define_var - define local variables
    define
        #_local_var - local variables
        new_rowid integer  # Rowid after insert

    # Set the serial field
    let m_orders.order_num = 0
```

```
    #_insert - Insert the data
    insert into orders values(m_orders.*)
    let new_rowid = sqlca.sqlerrd[6]

    #_serial - Bring back the serial field & display it
    let m_orders.order_num = sqlca.sqlerrd[2]
    let p_orders.order_num = sqlca.sqlerrd[2]
    call llh_display()

    #_on_disk_add
    #_end

    #_rowid - Reset rowid
    let sqlca.sqlerrd[6] = new_rowid

end function
# llh_add()
```

Use the `before block` command to add this one extra line before the `insert` block. Thus in the .ext file, you can place the following block command and source code:

```
start file "header.4gl"
    before block llh_add insert
        let m_orders.entry_date = today ;
```

This block command goes into the .ext file under the line `start file header.4gl` because you are modifying the source code in `header.4gl`.

The .ext file is read by the Featurizer, and the Featurizer pre-processes the appropriate .4gl file to include the extra line of code. After pre-processing, this is the result in the `llh_add` function (in `header.4gl`):

```
    # Set the serial field
    let m_orders.order_num = 0

    let m_orders.entry_date = today

    #_insert - Insert the data
    insert into orders values(m_orders.*)
    let new_rowid = sqlca.sqlerrd[6]
```

Custom Code Merged Before Block

If you want to insert the custom logic after the `insert` block, then use the `after block` command in the .ext file as follows:

```
    after block llh_add insert
         let m_orders.entry_date = today ;
```

This changes the `llh_add` function in the following manner:

```
#_insert - Insert the data
insert into orders values(m_orders.*)
let new_rowid = sqlca.sqlerrd[6]

let m_orders.entry_date = today

#_serial - Bring back the serial field & display it
let m_orders.order_num = sqlca.sqlerrd[2]
let p_orders.order_num = sqlca.sqlerrd[2]
call llh_display()
```

Custom Code Merged After Block

Note that all block commands in .ext files are delimited by semicolons.

You can even replace blocks. Let's say you wanted to add your custom logic *between* the `insert into` and the `let new_rowid` lines of code. You could replace the entire block with the replace block command:

```
replace block llh_add insert
    insert into orders values(m_orders.*)
    let m_orders.entry_date = today
    let new_rowid = sqlca.sqlerrd[6] ;
```

This results as follows in `llh_add`:

```
# Set the serial field
let m_orders.order_num = 0

#_insert - Insert the data
insert into orders values(m_orders.*)
let m_orders.entry_date = today
let new_rowid = sqlca.sqlerrd[6]

#_serial - Bring back the serial field & display it
let m_orders.order_num = sqlca.sqlerrd[2]
let p_orders.order_num = sqlca.sqlerrd[2]
call llh_display()
```

Custom Code Replaces Block

You can even search for strings in blocks and place code before or after a string of code within a block.

You can delete blocks with the following command:

```
delete block llh_add insert ;
```

The effect on `llh_add` is as follows:

```
# Set the serial field
let m_orders.order_num = 0

#_serial - Bring back the serial field & display it
let m_orders.order_num = sqlca.sqlerrd[2]
let p_orders.order_num = sqlca.sqlerrd[2]
call llh_display()
```

Code Block Deleted

In addition to manipulating code within blocks, you can add code to the top or bottom of a .4gl file. You use block commands with various reserved words as arguments to the commands. In lieu of the function name argument in a block command, you could specify TOF for Top of File or EOF for End of File. If you used these reserved words as the function name argument to the block command, the block name argument would be NUL for null, since there is no block at the top or bottom of a .4gl file.

Here is a block command that places extra code at the bottom of a the `header.4gl` file:

```
start file "header.4gl"

    after block EOF NUL
        display "this code is at the end of header.4gl"
        sleep 3 ;
```

Notice how `after block EOF NUL` acts—it puts text at the end of files. `TOF`, `EOF`, and `NUL` must all be uppercase.

Extension (.ext) Files

As mentioned earlier, block commands are placed in .ext files. Basically .ext files serve two purposes: the first is to provide a means of plugging and unplugging features; the second is to simply hold block commands which always need to be merged into the basic program.

Note

An .ext file can be named with any combination of letters, numbers and underscores. You cannot, however, use hyphens or any other symbol in an .ext's name.

For more information on the concept of pluggable features, refer to the separate section "Pluggable Features and Feature Sets" on page 5-21.

Specifying Which .ext Files to Merge

You must specify all .ext files you want to be merged by listing them in a file named `base.set`.

A more detailed description of `base.set` files is available in"Pluggable Features and Feature Sets" on page 5-21.

Specifying Source Code Files

The `start file` command allows you to specify which .4gl files you want your block commands to work on. The `start file` command, along with the blocks that correspond to it, are placed in .ext files. The syntax of the `start file` command is as follows:

`start file "`*`filename`*`"`

Example:

```
start file "header.4gl"

after block mlh_clear init
    initialize my_record.* to null ;
```

The following is an example of how you can use an .ext file, a `start file` command, and a block command to make a customization to a section of .4gl code.

Suppose that you wish to modify the function `mlh_clear` in `header.4gl`. You can do it with blocks. Here is an example of the `mlh_clear` function in `header.4gl`:

```
###########################################################################
function mlh_clear()
###########################################################################
#
    #_define_var - define local variables

    #_init - Initialize
    initialize p_orders.* to null
    initialize q_orders.* to null
    initialize m_orders.* to null

end function
# mlh_clear
```

You see the `define_var` and `init` blocks in `mlh_clear`. You wish to apply the following block command and code to the `init` block:

```
after block mlh_clear init
    initialize my_record.* to null ;
```

First you need to create a .ext file to put your block command in. Since this modification does not relate to a specific pluggable feature, you would create a `base.ext` file to put it in.

Next, you would add the `start file` line to specify which file you want to apply the block to.

Here is how you would apply the above `after block` command to `header.4gl`:

```
    start file "header.4gl"

    after block mlh_clear init
        initialize my_record.* to null ;
```

The result of the above block command would be in `header.4gl` as follows:

```
###############################################################################
function mlh_clear()
###############################################################################
#
    #_define_var - define local variables

    #_init - Initialize
    initialize p_orders.* to null
    initialize q_orders.* to null
    initialize m_orders.* to null
    initialize my_record.* to null

end function
# mlh_clear
```

Block Command Logic

The function name and block ID can also be viewed as *scopes*, or *starting points*. The Featurizer first searches for the function name. Once it locates the function name it searches for the block ID within that function name. Once this is found, code manipulation takes place. Function name and block ID really stand for major known section of the file and minor known section of the file, respectively. The block ID is the block tag without the #_.

The use of `from`, `after`, `to`, `thru`, or `through` can further define the block ID starting location. The keywords `thru` and `through` are synonymous.

The following function names and block IDs have special meaning when used in Block Command Statements:

- The `TOF` function name specifies the top of the file.
- The `EOF` function name specifies the end of the file.
- The `NUL` block ID means that there is no associated block tag for this command.

- a_*field-name* targets #_after_field *field-name*
- b_*field-name* targets #_before_field *field-name*
- c_*field-name* targets #_after_change_in *field-name*
- e_*event-name* targets #_on_event *event-name*

Block Command Statements

This section lists the syntax of each Block Command Statement and its definition.

start file "*filename*"

This command specifies that the commands below this line are working on the specified filename. The filename must be in quotes. It is required as the first block command in the .ext file, and may appear throughout the file to change the file associated with the block commands that follow this. An example of filename could be fg_funcs.4gl. For more information refer to "Specifying Source Code Files" on page 5-11.

before block *function-name block-ID*

This inserts the text directly above the first line of the block. The special function name of TOF inserts the text at the top of the file.

in block *function-name block-ID* {before | after} "*string*"

This inserts the text either before or after the line that begins with the specified string. before or after is required. The line identification string can be 50 characters max. The special function name of EOF is not allowed in this command.

after block *function-name block-ID*

This inserts the text after the last line of the block. The special function name of EOF inserts the text at the end of the file.

replace block *function-name block-ID* [{from | after} "*string*"] [{to | thru} "*string*"]

This replaces the specified block (or portion of a block) with the given text. You may specify `through` instead of `thru`. The line identification strings can be up to 50 characters long. If the entire block is specified (with no from/after or to/thru strings) only the text portion of the block is replaced. The #_ block tag line and the #_`end` line (if present) are preserved. The special function name of EOF is not allowed in this command.

`delete block` *`function-name block-ID`* `[{from | after} "`*`string`*`"] [{to | thru} "`*`string`*`"]`

This deletes the specified block (or portion of a block). The line identification strings can be up to 50 characters long. The special function name of EOF is not allowed in this command.

One special delete block command can be used to delete the entire contents of a file. It is `delete block TOF NUL thru "`*`string`*`"`, where string is the last line in the file.

Note

Caution: The `delete block` command deletes all existing block tags within the specified block, thus making it difficult to maintain regenerability.

`function define` *`function-name`*

This command only allows you to define new or additional local variables used in a specific function. If you need to add some local variables to a specific function, use this command. If the function specified by function name does *not* have the `define` keyword in it (there are no local variables previously defined in this function), the Featurizer puts the `define` keyword in, before adding the variables.

Note

Semicolons: All block commands except `delete block` require additional text following the command. This additional text *must* be terminated with a semicolon. In the case of the `delete block` command, you do not need a semicolon.

Using Strings in Block Commands

Using strings in block commands should be avoided if possible. The reason being the generated code may change in future releases causing the Featurizer to be unable to locate your strings.

Since block tags will not change in future releases you can be sure your code will remain compatible if you rely on these points in the code. However, if you use a string to locate a block, the generated code may change over time with enhancements which may break your string searches.

A string can consist of up to 50 characters.

Note

Very important: When using strings, you must include the text from the beginning of the line through the string that you are trying to target. In other words, you cannot specify a string that begins in the middle of a line of text. If you try this, it results in a Featurizer error. See the following example.

Example:

```
let abc = xyz.
```

If you use string equal to `abc`, the Featurizer errors out. If you use string equal to `let abc` (again, including text up to the beginning of the 4GL line you are trying to target), the Featurizer finds the line.

Illustrated above in the `replace` and `delete` block commands, is the use of strings such as `after`, `from`, `to`, `thru`, and `through`. When deciding which one to use you must decide whether or not you want to include the line of code that matches the string pattern in the effect of the change. In other words, using `from abc` in a delete block causes the line of code containing the string `abc` to be deleted as well. Consider the following to help your decision:

- after string - line matching string is un-affected
- from string - line matching string is affected
- to string - line matching string is un-affected
- thru string - line matching string is affected
- through string - line matching string is affected

Note

A back slash must proceed double quotes in a string block command.

The following example DOES NOT work:

```
"when scr_fld = "stock_num""
```

Adding a back slash before the double quotes works.

```
"when scr_fld = \"stock_num\""
```

Block Identification & Grouping

The start of a block is always a line that begins with a #_ as the first non-blank character of the line.

The end of a block is determined by the following rules:

A - The next block at the same indentation level, or

B - Any text to the left of the block identification line, or

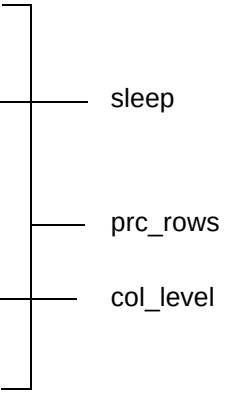
C - An `end function` statement as the first words of the line, or

D - An explicit #_`end` block marker

Given these rules for ending blocks, any block indented to the right of another block is considered contained in the first block.

This works well for programming constructs that have control processing (like if/ end if, case/end case, foreach/end foreach, etc.) Consider the following program segment:

```
1    #_prc_rows - Process the rows in the cursor
2    foreach abc_cursor into my_rec.*
3
4        #_sleep - Had much sleep lately?
5        if my_rec.recent_sleep = "Y"
6        then
7            display "Need more sleep..."
8            let my_rec.need_sleep = "Y"
9        end if
10
11       #_col_level - Need a cholesterol level checkup?
12       if my_rec.eats_fats = "Y"
13       then
14       if my_rec.num_hamburgers > 20
15           then
16               display "Checkup is due..."
17               let my_rec.need_checkup = "Y"
18           end if
19        end if
20
21   end foreach
22
23   #_nxt_blk - Next block..
```



Block	Start Line	End Line	Rule
prc_rows	1	21	A
sleep	4	9	A
col_level	11	19	B

If you wish to group a number of blocks that have no control loop structure, you may indent the blocks within the group.

If a block is indented due to logical grouping, by convention there should be an `#_end block` *`block-name`* marker. This is not required by the Featurizer, but it is a convention that should be practiced. For exmaple notice line #19 in the following code:

```
1    #_bldcmd - Build the shell command to run that gets a list of
2    # all .ext files in the current directory and in the
3    # custom directory paths.
4
5        #_stfind - Start the find command & add current directory
6        let scratch = "cd ..; find ",
7          dir_name clipped, ".", dir_ext clipped
8
9        #_addcus - Add custom directories
10       for cur_path = 1 to num_paths
11           let scratch = scratch clipped, " ", dir_name clipped,
```

```
12            ".", cust_path[cur_path]
13        end for
14
15        #_finfind - Complete the find command
16        let scratch = scratch clipped, "'(' -name '?*.ext' -o ",
17          "-name '?*.ext' ')' -print 2>/dev/null"
18
19    #_end block bldcmd
20
21    #_prcfiles - Process
22    while true
23        call c_command(scratch)
24          returning stat_flag, stat_exit, sql_filter
25
26        #_noelem - No more elements to read
27        if stat_flag < 1 then exit while end if
28
29    end while
```

Note that the `prcfiles` block would have ended the `bldcmd` and `finfind` blocks implicitly, but the explicit #_`end` block line should be used.

Note on Block Replace and Block Delete

If a replace or delete block command is passed a string that causes the deletion to *span* a block start or end line, the block ID for the spanned block is deleted (for example, it cannot be used in a later block ID). If the following command is specified on the following file:

```
delete block TOF stfind from "dir_name" thru "for cur_path"

1    #_bldcmd - Build the shell command to run that gets a list of
2    # all .ext files in the current directory and in the
3    # custom directory paths.
4
5        #_stfind - Start the find command & add current directory
6        let scratch = "cd ..; find ",
7          dir_name clipped, ".", dir_ext clipped
8
9        #_addcus - Add custom directories
10        for cur_path = 1 to num_paths
11            let scratch = scratch clipped, " ", dir_name clipped,
12            ".", cust_path[cur_path]
13        end for
14
15        #_finfind - Complete the find command
16        let scratch = scratch clipped, "'(' -name '?*.ext' -o ",
17          "-name '?*.ext' ')' -print 2>/dev/null"
18
19    #_end block bldcmd
20
21    #_prcfiles - Process
22    while true
23        call c_command(scratch)
24          returning stat_flag, stat_exit, sql_filter
25
26        #_noelem - No more elements to read
27        if stat_flag < 1 then exit while end if
28
29    end while
```

deleted

unusable blocks

Given the previous file, lines seven through 10 would be deleted. Since the command spanned over the top of the `addcus` block, the `addcus` block ID cannot be used any longer. The deletion also spanned *past* the end of the `stfind` block, and the `stfind` block ID cannot be used any longer. The larger `bldcmd` block ID is left intact because the deletion was completely within it.

Spanning blocks for deletion is not suggested because it disturbs the logical grouping of blocks. In the above example, it would have been better to delete both the `stfind` and `addcus` blocks, then insert any new logic above the `finfind` block.

Note

If the text of a command inserts or replaces block labels, the text of the insertion is scanned for any new block IDs. The block scan is limited to the end of the insertion. When inserting blocks, there is no way to have any new block label span past the end of the insertion.

Custom Block ID (Tags) Conventions

- Block IDs contain no white space and are unique to 20 characters. They can only contain alphanumeric and underscore characters.
- Block IDs should uniquely identify the block within the function and be somewhat readable.
- Block IDs are case sensitive.
- Block IDs are lowercase letters followed by a space-dash-space and a verbal block description starting in an uppercase letter.
- Block IDs never change. Their description changes, the code in them changes, but their IDs always stay the same.

The following shows a few sample block IDs.:

```
#_init - Initialize
#_verify_credit - Verify the credit limit
#_ln_calc - Calculate the order line amount
```

Pluggable Features and Feature Sets

Pluggable Features: are individual features that are stored in source code extension (.ext) files. The filename specifies the feature that it contains. For example, a file containing source code for the balance forward feature might be called `balfwd.ext`.

Feature Sets: contain a list of features to apply to the application. Feature set files are named `base.set`. Each feature contained in a `base.set` file is stored in an .ext file. All .ext files are specified one to a line, and are listed in their order of merging.

Once you have a feature self-contained in a .ext file, you have the ability to *plug* the feature into the program. To *plug in* a feature means that you instruct the Featurizer to merge the code just for that feature into the .4gl source code files. The Featurizer takes the feature-driving code from the .ext file and merge it into the rest of the source code.

Pluggable Features (.ext Files)

An .ext file contains all of the source code necessary to drive one feature. You determine which .4gl file to perform work on by using the `start file` block command. A `start file` command must precede any block commands. You can specify multiple commands in an .ext file to perform modifications to multiple files.

Note

In order for the source code in your .ext files to be merged, you must list the name of each .ext file in a feature set (`base.set`) file.

The prefix of an .ext file describes the feature for which it contains code. For example, in `approval.ext`, you might find code in block commands that drives an approval entry feature. For `secur.ext`, you could find code in block commands that institutes security on a program.

For example, with .ext files you can create the specific logic that drives an approval feature. This way you can easily plug in or unplug this feature from your different applications.

An .ext file has no sections, therefore, a `start file` command is always issued in an .ext file to indicate which file to insert the code.

Feature Set (base.set) Files

You instruct the Featurizer which features to plug in through a `base.set` file. A `base.set` file holds the user-specified settings for that program. The `base.set` file is the user's feature list.

You specify features in the `base.set` file as the names of the .ext files without the .ext extensions. In a `base.set` file, anything placed one space after the feature is not read by the Featurizer. You can use the rest of the line for comments. The following example of `base.set` merges the code for the approval and installs features into the .4gl files:

```
approval - prompts for approval for orders of $500
instvals - pulls up list of valid values for shipping instructions
```

When you invoke the Featurizer, the features in the `base.set` it are merged in *the order listed*. Each feature listed in the `base.set` file must have an associated .ext file of the same name.

Note

Since the Featurizer looks for only one `base.set` file, you must be sure that the `base.set` file in your current directory contains all of the features you want to incorporate into your program. In other words, if you have a common function specified in the `base.set` directory at your application level and you want to include those functions in a specific program, you must either specify that application level `base.set` file, or specify each individual .ext file listed in that application `base.set` in a new `base.set` file located in the program directory. If you want to add new features to your program with .ext files, you must be sure to add those features to the `base.set` file.

Pre-merged Generated Files (.org Files)

The Code Generator and the Featurizer both create .org files. Whenever a block is merged into a 4gl file, an .org file is created which is a copy of the .4gl file before anything gets merged into it. The .org file contains source code in its generated but pre-merged form.

The Code Generator and .org Files

When the Code Generator is run, it searches to see if any .org files are present in the current directory, or in the custom directory path. If it does find an .org file, the Code Generator creates a new .org file with the same filename prefix. If an .org file is not found, a .4gl file is created instead.

The Featurizer and .org Files

Whenever the Featurizer merges a block into a .4gl file that does not have an associated .org file, an .org file is created by copying the .4gl to an .org. If an .org file does not exist for a specific .4gl, such as `header.4gl`, the Featurizer assumes that this particular .4gl does not have any blocks in it. The Featurizer then copies that `header.4gl` file to a `header.org` file. Once an .org file exists, the Featurizer loads the .org, merges the blocks into it, then creates a new .4gl file that contains the merged code. Every time a merge takes place, the merge is performed on the .org file to create a new .4gl.

The Featurizer creates an .org file in the current directory for every file specified with a `start file` command.

Removing Blocks from Existing .4gl Files

The following logic only applies to the situation where you used to have blocks merged into a file and decide that you no longer want anything merged into that file.

Say you once had a `schedule.ext` file with a `before block` that has already been merged into `header.4gl`, and you decide you no longer want it. All you have to do is remove that extension file and then run the Featurizer.

Special logic has been added to the Featurizer to automatically handle this situation. The Featurizer copies the `header.org`, which must exist if the `header.4gl` has been merged before, over to `header.4gl`, thus restoring `header.4gl` to its original generated state.

Flow of the Featurizer

The following describes the operational flow of the Featurizer.

1. **Load feature sets into the database.**

   All .ext files for the specified feature set are located in the current directory and the custom directory search path. If any of these files have been modified since the last compile, they are marked as modified, and loaded into the database.

2. **Build a list of files to process.**

   If a file or list of files is passed onto the command line, the Featurizer merges only those files. The `-force` option is assumed if files are specified on the command line.

   If no files are specified on the command line, the Featurizer must build the list. It does this in two phases.

   First, it builds the initial list as all files that have been referenced in all the .ext files in the current directory and the custom directory search path.

   If the `-force` option is specified on the command line, this initial list is used, and this step is complete.

   Second, the Featurizer checks each .ext file in the list to see if they have been modified since the last merge. If a file has not been modified (the modification date of the file is the same as the .4gl file), the file is ignored. If the file has been modified since the last merge, then the Featurizer remerges that file.

3. **From the list of files to process, each file is Pre-processed as follows:**

   1. Determine the original (.org) source file to work from, and load it into memory.

      The .org file is usually in the current directory, but if it doesn't exist here, the custom directory search path is searched to find the .org file to work from.

      The name of the .org file is built by appending .org to the destination file-name, or by replacing any three-character file extension with org. It then loads this .org file into memory for processing.

If no .org file is found (meeting this naming criteria) in the search, a UNIX `cp` command is run on the .4gl file to create an .org in the current directory. The name of this .org file is the same as the destination filename with any three-character extension replaced by .org. If the destination filename does not have a three-character extension, then .org is appended to the filename to determine the .org filename (up to 14 characters).

2. Build a list of commands (CMDs) to apply to this file.

   Commands (CMDs) are block commands stored in the .ext files for this feature set.

   The sequence that CMDs are merged into the code is significant. The order is determined by the file they are located in, and their relative position within that file. The ordering rules follow:

- CMDs stored in lower-level directory search paths are applied before CMDs in the current directory. The default order is .4gs, then .4gc, then the current directory. This order may be overridden with the CUSTPATH setting.
- All CMDs in one directory are processed before any CMDs in another directory in the search path.
- The order of .ext files is determined by the order that the features are specified in the `base.set` file for this feature set.
- CMDs are then merged in their order within the .ext files.

3. Execute that list of commands in their proper sequence.

   After the list of CMDs has been built, each CMD is individually processed. If the block within the .org file isn't found, an error is displayed unless the CMD originated from a higher directory in the search path.

4. Create .tmp files and/or .4gl files.

   The Featurizer outputs to a .tmp file. It then compares the .tmp file with the existing .4gl file, if there is one. If there is no difference, the original .4gl file is untouched, thus preserving the time stamp of that .4gl file. If no .4gl files are present, the Featurizer copies the .tmp files into .4gl files.

Note

Do not use .tmp extensions for your files. The .tmp extension is used by the Featurizer as well as the Code Generator. If you use a .tmp extension the file will be removed.

Filename Extensions

Extension	File Explanation
.4gm	Indicates an application module directory.
.4gs	Represents a 4GL source code directory.
.4gc	Stands for a custom 4GL source code directory.
.4gl	Represents a 4GL source code file.
.4go	Indicates an object code file compiled with RDS.
.o	Indicates an object code file compiled with the C Compiler.
.4ge	Represents an executable program file, which is run directly from the command line.
.4gi	Represents a program file, which is executed with the runner `fglgo` or the `fgldb` debugger.
.ifg	Indicates an image file or `report.ifg`. This file contains a picture or image of the report format.
.ext	Stands for source code extension file. These files contain custom code that the Featurizer merges into the original source code.
.set	Indicates a feature set file or `base.set`. This file holds a list of .ext files that are merged into the report source code.
.tmp	Represents a temporary file. This extension is reserved for use by the Featurizer and Code Generator.
.org	Represents an original source code file. When extension files are merged by the Featurizer, original source code gets preserved in these files.
.opt	Stands for option file or report.opt. This file lets you set custom report program variables on the source code directory level. For more on option (.opt) files, refer to "Backward Compatibility" on page 7-2.

Featurizer Environment Variables

$fg: Path to the fourgen directory (used to find executables so you do not have to be within `$fg` while running the Featurizer).

$cust_path: If this variable is set before code generation and no `CUSTPATH` variable exists in an existing `Makefile`, then the value of `$cust_path` is written into the new `Makefile`. If `CUSTPATH` is already set in a `Makefile`, the `$cust_path` variable is ignored. This variable provides a path that the Featurizer searches for .ext files to merge.

Featurizer Limitations

Limitations	Number	Notes
Files it can pre-process in one directory	50	A
Custom directories to search in CUSTPATH	10	A
Features in a feature set	100	A
Characters in custom directory extensions	3	B
Number of #_ block definitions in one file	1000	A
Lines in the (.org + .ext files)	7500	A, C
Block CMDs for an .org file	unlimited	D
Characters in one line	unlimited	E
Block nesting levels	10	A

Note	Description
A	The internal program array limit.
B	By convention.
C	The number of lines (excluding blank lines) in the .org file and the number of lines in all .ext files that refer to this .org.
D	An .org file can contain any number of block commands as long as the total number of lines does not exceed the line limit specified in [C].
E	The number of characters right of the indentation level. If it exceeds 70 characters, the lines are (internally) split into as many 70-character lines as necessary. Each split internally consumes a new line (of which there are a limited number, see [C] above).

Troubleshooting Tips

Question: Where is the Featurizer located?

Answer: The utility, `fglpp.4ge`, is located in the `$fg/codegen/screen.4gm/fglpp.4gs` directory.

Question: What changes to my program require regeneration of my program versus simply merging my files with `fg.make`?

Answer:

1. Addition of new fields to a screen.
2. Deletion of fields from a screen.
3. Addition or deletion of lookups and zooms.
4. Addition of a global event.
5. Addition of a local event.
6. Changes to your table schemes.

Question: Are comments acceptable in my extension files?

Answer: Comments are acceptable in most cases.

Question: How do I cause the Featurizer to never be run from `fg.make` or the generator until I decide I want to turn it back on?

Answer: Set the environmental variable `no_merge=Y`.

Question: Where do I look for error messages explaining why the Featurizer is aborting?

Answer: These can be found in the file `fglpp.err`. This file resides in the program directory in which you are currently working.

6

Creating Advanced Report Features

Quality report programs contain many features and customizations. The *Report* Code Generator lets you create customizations to fit your needs. This section discusses some practical report customizations that you might want to add to your report.

This section covers the following topics:

- Designing Report Prompts
- Modifying Report Functions for Job Scheduling
- Using Database Transactions for Posting
- Creating Transaction Logging Functions
- Issuing a Commit Work Without Closing the Cursor
- Migrating Applications to Other Systems

Designing Report Prompts

Programs that produce reports have many parts. One part allows the user to enter selection criteria, another part retrieves the data for the report, and a third formats the data and creates the actual output.

The code for the last two parts of this process, the retrieval of data and the formatting of data, are created by the Fitrix *Report* Code Generator and are completely consistent in general design.

The first part of this process is not generated by the Code Generator. This is because of the variety of different methods available for allowing the user to enter selection criteria.

Obtaining Selection Criteria

There are four different methods of obtaining selection criteria from the user. The four methods are:

1. **Creating Single Value Prompts**

   A single value prompt accepts one value for a single variable at a time. This type of prompt works best when users make quick single criteria inputs. For instance, when the user must enter an archive date, a dialog box appears and prompts the user for the date. Once the date is entered, a validity check is performed on the date. If valid, the date gets passed as a selection statement. If not valid, the date is converted to a valid date, or the prompt re-appears and requests a valid entry.

2. **Creating Input Forms**

   Input forms accept one value for one or more variables at a time. For example, when users want to specify a range of dates, a dialog box appears that contains an input form. On the form, the user can enter a value into a start date and end date field. The entered dates are checked to verify they are valid, and the selection statement is created. Input forms allow for sophisticated types of logic including when-leaving and when-entering field logic to preassign, check, and format values.

3. **Creating Query-By-Example Forms**

The Query-By-Example method lets you enter one or more values for one or more variables at a time. For example, a report built from several different columns might require large selection criteria. This method lets you build large selection statements for several report values at once. In addition, this method works well in situations involving a number of different fields for which you want to specify a number of different relationships, such as equal-to, greater-than, and less-than.

4. Creating Command Line Selection Criteria

The command line selection criteria method accepts one value per variable and the entry of one or more values at a time. This method allows you to pass selection criteria taken from outside the program into the program. The Fitrix *Menus* software used to write the menus allows for the construction of a data-entry form at the menu level. This form takes user input and then passes it to an INFORMIX-4GL program. One advantage to using this method is that the same selection criteria can be passed to several different programs without having the user re-enter it. Each 4GL program gets the criteria information from its arguments and converts it into a criteria string.

Sample Programs

The following are examples of each method of getting user-defined criteria. The main function `create_selection` calls on the `sel_cust` function. There are four different `sel_cust` functions. Each one is an example of the different methods of getting user-defined criteria.To create regenerable input prompts you must own the Fitrix *Screen* Code Generator. For more on integrating prompts into your report program, see "Adding in Report Prompts" on page 7-13

The `sel_cust` form is used for input and construct methods.

```
###########################################################
# sel_cust form
###########################################################
screen
{
---------- Select Customers -------------------
Customer Code: .a
Customer Name: .b
-------------------------------------------------}
tables
strcustr
attributes
a = strcustr.cust_code,upshift
b = strcustr.bus_name,upshift
end
```

```
instructions
screen record s_data (strcustr.cust_code, strcustr.bus_name)

function
create_selection()
# This function prepares the select statement
# from table strcustr (customer reference) for
# execution. The criteria (customer business
# name and customer code) is based on the value
# returned from the function sel_cust()
#
  define
    sel_criteria char(256),
    sel_stmt char(320)
  call sel_cust() returning sel_criteria
  let sel_stmt = "select * from strcustr where(",
    sel_criteria clipped, ")"
  prepare get_curs from sel_stmt
end function
```

Single Value Prompts

```
################################################################
function sel_cust()
# returns cust_sel
################################################################
# This function prompts the user to enter
# customer code from which it creates a
# construct statement and returns it. If an
# invalid customer code is entered, the program
# will prompt the user for a valid one
#
  define
    cust_code code(6),
    cust_sel char(256),
    invalid_cust smallint
  open window selwin at 6, 6 with 14 rows,
    70 columns attribute (border, blue)
display
"======================================================================="
at 3,1
  let invalid_cust = true
  while invalid_cust
    let invalid_cust = false
    whenever error continue
      prompt "Enter Customer Code: "
        for cust_code
    whenever error call error_handler
    # validate the customer code
    if status != 0 or validate_cust(cust_code)
    != 0
    then
      let status = 0
      let invalid_cust = true
      call fg_err(1) # invalid cust code
    end if
  end while
  close window selwin
  let cust_sel = "strcustr.cust_code = ",
    cust_code
  return cust_sel
end function
```

Input Forms

```
##############################################################
function sel_cust()
# returns cust_sel
##############################################################
# This function allows the user to enter
# customer selection criteria (customer business
# name and customer code) using input from a
# selection screen from which it creates a
# construct statement and returns it. if an
# invalid customer code is entered, the user
# will be placed back into the customer code
# field
#
  define
    p_cust_code like strcustr.cust_code,
    p_bus_name like strcustr.bus_name,
    cust_sel char(256)
  open window selwin at 6, 6 with 14 rows,
    70 columns attribute (border, blue)
  display
"=============================================================================" at
3,1
  input p_cust_code, p_bus_name from
  s_data.cust_code, s_data.bus_name
    after field p_cust_code
      if validate_cust(p_cust_code) != 0
      then
        call fg_err(1) # invalid cust code
        next field p_cust_code
      end if
  end input
  if int_flag = 1
  then
    let int_flag = 0
    exit program(1)
  end if
  close form sel_screen
  close window selwin
  let cust_sel = "strcustr.cust_code = ",
    p_cust_code, " and strcustr.
      bus_name = ", p_bus_name
    return cust_sel
end function
```

Query-By-Example Forms

```
##############################################################
function sel_cust()
# returns cust_sel
##############################################################
# This function allows the user to enter
# customer selection criteria (customer
# business name and customer code) using query
# by example from which it creates a construct
# statement and returns it.
#
  define
    cust_sel char(256)
  open window selwin at 6, 6 with 14 rows,
    70 columns attribute (border, blue)
  display
"=============================================================================="
```

```
at 3,1
  display " ENTER SELECTION CRITERIA" at 2,1
    attribute(white)
  display
    "Press [DEL] to Cancel
    or [ESC] to Select"
    at 2,30 attribute(white)
  open form sel_screen
    from "../../lib.4gm/ar.4gs/sel_cust"
  display form sel_screen
  construct cust_sel
    on
      strcustr.cust_code,
      strcustr.bus_name
    from
      s_data.cust_code,
      s_data.bus_name
  if int_flag = 1
  then
    let int_flag = 0
    exit program(1)
  end if
  close form sel_screen
  close window selwin
  return cust_sel
end function
```

Command Line Selection Criteria

```
############################################################
function sel_cust()
# returns cust_sel
############################################################
# This function gets the customer codes from the
# command line and converts them into a
# selection statement
#
  define
    cust_sel char(256),
    cust_code char(6),
    n smallint # working number
  if num_args() > 10
  then
    call fg_err(2) # too many argument on the
                   # command line
    exit program(1)
  end if
  for n = 1 to num_args()
    let cust_code = arg_val(n)
    if validate_cust(cust_code) != 0
    then
      call fg_err(1) # invalid cust code
      exit program(1)
    end if
    if n = 1
    then
      let cust_sel = "strcustr.cust_code = ",
        cust_code
    else
      let cust_sel = cust_sel clipped,
        " or strcustr.cust_code = ", cust_code
    end if
  end for
  return cust_sel
end function
```

Report Production and Formatting

The *Report* Code Generator *report production* routine has a certain set of rules that governs its use. In this process, a series of data records is passed to this routine. It then manipulates and formats the data for output. The code for report production and formatting is found in the `report.4gl` file.

For reasons of modularity and to make it easier to reuse, the report-generation routine has its own unique environment and is separated from all other routines and functions. Global variables cannot be used by the report-generation routine. The only interaction this routine has with other 4GL programs is in terms of the data that is passed to it by those programs.

The report-generation routine is itself divided into several parts. The `FORMAT` section creates the report image as it appears on paper.

Here are several subroutines we have created for the formatting section of the report-generation routine:

1. Text Centering Subroutine

This subroutine centers a field (or fields) in the middle of the page, no matter what amount of data is included. It does the following:

- Assigns a temporary character variable to the variable to be printed.
- Assigns a temporary integer variable to be the center column of the report.
- Calculates and assigns an integer variable to the starting print column.
- Prints the temporary variable at the calculated column.

```
let scratch = rpt.field(s)
let mid_column = 40
let x = mid_column - (length(scratch)/ 2)
print column x, scratch clipped
```

2. Text Right Justification Subroutine

This routine right-justifies a field or fields. In other words, it aligns the data with the right-hand margin, no matter how wide the data is. In addition, it does the following:

- Assigns a temporary character variable to the variable to be printed

- Assigns a temporary integer variable to be the last column of report
- Calculates and assigns an integer variable to the starting print column
- Prints the temporary variable at the calculated column

```
let scratch = rpt.field(s)
let last_column = 81
let x = last_column - length(scratch)
print column x, scratch clipped
```

3. Dynamic Heading and Footer Subroutines

This is used to print heading and footer lines only if there are rows that are printed for that report section. This is the case when you have a cursor of information from a header-detail join relationship where you are printing your header information in the `before group` section and the detail information in the `on every row` section. In your report, you want to have a heading for your detail only if there is detail associated with the header. This routine does the following:

- In the `before row` section, it tests to see if there is detail data and sets a flag to print, or not to print, a heading. If the flag is set, it then prints the heading; otherwise it does not print anything.
- To prevent the printing of blank lines when there is no detail, it tests to print or not print in the `on every row` routine before group of `rpt.header`.

```
if rpt.detail1 is not null or
  rpt.detail2 is not null
then
  let dynamic_flag = true
else
  let dynamic_flag = false
end if
if dynamic
then
  print "DETAIL HEADING"
end if
on every row
if rpt.detail1 is not null or
  rpt.detail2 is not null
then
  print rpt.detail1,rpt.detail2
end if
```

4. Page Number On Group Subroutine

When printing multiple page forms, this subroutine keeps track of the page number per group.

```
page header
  if p_pageno is null
  then
```

```
    let p_pageno = 0
  end
  if let scratch = pageno - p_pageno
    using "<<<<<<"
  print column x, scratch clipped
before group rpt.field
  skip to top of page
after group rpt.field
  let p_pageno = pageno
```

Incorrect Trailer Information Subroutine

While printing forms, you want the program to skip to the top of the page in the before group section. If there is trailer information to print, the current data record is from the next record, so your trailer output is incorrect. This subroutine corrects the problem. It does the following:

- Defines a trailer record the same as your trailer.
- `rpt.variables` and sets a flag to indicate `after group`.
- If an `after group` has occurred, it sets the printing of the trailer information to the previous record.

```
               last_trlr record
                 1st_trlr_info like ....,
 2nd_trlr_info like ....,
 3rd_trlr_info like ....
end record
before group rpt.field
  skip to top of page
after group rpt.field
  let after_group_flag = true
  let last_trlr.1st_trlr_info = rpt.1st_trlr_info
  let last_trlr.2nd_trlr_info = rpt.2nd_trlr_info
  let last_trlr.3rd_trlr_info = rpt.3rd_trlr_info
page trailer
  if after_group_flag is null
  then
    let after_group = false
  end if
  if after_group_flag
  then
    let after_group = false
  else
let last_trlr.1st_trlr_info = rpt.1st_trlr_info
let last_trlr.2nd_trlr_info = rpt.2nd_trlr_info
let last_trlr.3rd_trlr_info = rpt.3rd_trlr_info

end if
print last_trlr.1st_trlr_info
print last_trlr.2nd_trlr_info
print last_trlr.3rd_trlr_info
```

Modifying Report Functions for Job Scheduling

The Fitrix *Menus* program allows users to schedule reports to run at a specified future time. The scheduling logic provides scheduling for any job using the `:print:` instruction as well as for reports generated by the `:ifxreport:` instruction. In the case of the Informix generated report, it is the responsibility of the report program itself to allow itself to be scheduled.

The following changes to `globals.4gl, midlevel.4gl`, and the file containing input logic allow a report program to take advantage of the scheduling logic.

1. Add these variable definitions to `globals.4gl`:

```
sel_flag smallint,              # flag for selection only processing
bg_flag smallint,               # flag for background processing
job_id like stxfiltr.unique_id, # unique job id
```

2. Add this argument processing logic to `ml_defaults()` in `midlevel.4gl`:

```
define
  n_args smallint,
  n smallint

# initialize globals
let n_args = num_args()
let sel_flag = false
let bg_flag = false
let job_id = ""

# check for flag
for n = 1 to n_args
    if arg_val(n) = "-s"
    then
      let sel_flag = true
      let job_id = arg_val(n + 1)
    end if
    if arg_val(n) = "-b"
    then
      let bg_flag = true
      let job_id = arg_val(n + 1)
    end if
end for
```

3. Add this logic to your input processing function:

```
# check for background processing
if not bg_flag
then
  ...
```

```
  ...
  (existing input function logic)
  ...
  ...
else
    # get criteria from stxfiltr
    whenever error continue
    select sel_filter
    into selection_string
    from stxfiltr
    where stxfiltr.unique_id = job_id

    # make sure string was found
    if status
    then
      # default selection string in case of error, may be "1=2"
      let selection_string = "1=1"
    end if

    # delete the stxfiltr values
    delete from stxfiltr where stxfiltr.unique_id = job_id
    whenever error call error_handler
end if

# selection only processing
if sel_flag
then
    # insert the selection string into stxfiltr
    whenever error continue
    insert into stxfiltr values (job_id, 1, selection_string)
    whenever error call error_handler

    # done - exit program
    exit program(status)
end if
```

You can even add several pieces of data by using the `seq_no` field in `stxfiltr`. For example, in the Fixed Assets module, the option `Print Depreciation Report` (`p_assetd`) there is a prompt for a date in the *check* phase of the report only. This date is stored in `stxfiltr` with `seq_no = 2` for scheduled jobs. When retrieved the global normally filled by the prompted value is assigned the retrieved value (use a character string to fetch the value then convert it to date using `let`).

For further information on the establishment of job scheduling with Fitrix *Menus*, see the section titled "Report Scheduling Through Fitrix *Menus*."

Using Database Transactions for Posting

In Fitrix Accounting, many programs *post* data in one form or another. This means that when the process is run, the data is changed or updated in some way. Typically, it is taken from an entry file and moved to an activity file, but this is just one type of such posting.

In Fitrix Accounting, posting programs begin with either the `o-` (if they produce printed output) or the `p_` (if they just change data) prefixes. In writing these posting programs, we used the SQL concept of *database transactions*. Database transactions offer the programmer some unique features:

- They can be used to guarantee that everything between your `begin work` and `commit work` statements is either done 100%, or is not done at all (great for power failures, and those programs that blow-up in the middle of posting because that null value wasn't taken into consideration).
- They give you the ability to *rollback work* (automatically undo the changes made to the database since the last `begin work` command).
- They give you the ability to *rollforward database*. That is the ability to take a backup of your database, and apply all changes made to it since that backup to bring it up to date.

If transactions offer so much, why aren't they used very often? Most programmers will see the value in the first two benefits outlined, but it's the `rollforward database` statement that makes transactions, for the most part, unusable. The reason for this is simple: if you are able to `rollforward database,` then you must always use transactions. The problem with this restriction is twofold.

First, the transaction `logfile` can grow out of control in a very short time. In some testing, the `logfile` grew larger than the database itself within an hour.

Second, `begin/commit/rollback work` consumes system resources when they are required for only a small portion of database i/o that occurs. In Fitrix Accounting, only posting routines need to use `begin/commit/rollback` statements.

The fact that Fitrix Accounting had to guarantee data integrity during posting required us to use database transactions, but we wanted to avoid maintaining a `logfile` that could grow to 20MB in just one day of activity. So a way was developed to turn transactions on when they were needed, and turn them off when they were not. This sacrifices the ability to `rollforward database` from a transaction `logfile`, but the overhead required by this feature was not worth the benefit.

Creating Transaction Logging Functions

There are two functions that turn database transactions on or off. The philosophy behind these functions is simple: INFORMIX-4GL knows if the database uses transactions by the presence or absence of a row in `systables` with the tabname `syslog.` If you want database transactions, add the row: if not, delete it. The following pages contain the layout of the functions `add_log` and `chk_log.`

1. The `add_log(dbname,logpath)` function:

   If the `syslog` row isn't in `systables,` then insert it, close the database, reopen the database (it then opens using transactions), and delete the `syslog` row (the default is no `syslog` row).

```
################################################
function add_log(dbname,logpath)
################################################
# Add a record "syslog" into systables.
# Close and reopen the database for logging.
# Deletes the record "syslog"
#
  define
    log_rowid integer,
    dbname char(14),
    logpath char(64)
  let status = 0
  whenever error continue
  select rowid into log_rowid from systables
    where tabname = "syslog"
  if status != 0
  then
    let status = 0
    # not there - insert it.
    insert into systables
      (tabname,owner,dirpath,tabid,
      rowsize,ncols,nindexes,nrows,
      created,version,tabtype)
    values
      ("syslog","informix",logpath,
      0,0,1,0,0,today,0,"L")
    if status != 0
    then
      # Cannot add the syslog row
      call fg_error("lib_all","log_on",1)
      exit program(1)
    end if
    close database
    database dbname
    if status != 0
    then
      # Cannot open the database logfile.
      call fg_error("lib_all","log_on",2)
      exit program(1)
    end if
    # delete it for the next user.
```

```
    delete from systables
      where tabname = "syslog"
  end if
  whenever error call error_handler
end function
# add_log
```

2. The `chk_log(dbname,logpath)` function:

   If the `syslog` row is in `systables,` then a user is opening the database with transactions right now. Wait a while and check again. If the wait is too long, then just delete the `syslog` row from `systables.` Then close, and re-open the database (this time without transactions).

```
################################################
function chk_log(dbname,logpath)
################################################
# Check table systables to see if transactions are set on. If so,
# sleep until the record is removed.
#
  define
    sys_rowid integer,
    dbname char(14),
    logpath char(64),
    cnt smallint
    let status = 0
    whenever error continue
    select rowid into sys_rowid from systables
      where tabname = "syslog"
    if status = notfound
    then
      # database opened without transactions
      let status = 0
    else
      # wait for database to open without trx
      open window wait_sys at 2,4
        with 1 rows, 60 columns
        attribute(white,border)
      display
      " Waiting for table to become available..."
      at 1,1
      let cnt = 0
      while cnt < 5
        sleep 5
        select rowid into sys_rowid from systables
          where tabname = "syslog"
        if status = 0
        then
          let cnt = cnt + 1
          continue while
        else
          exit while
        end if
      end while
      if cnt = 5
      then
# problem with syslog record. it should never be there for this
# long. the only time the syslog record should be there is when
# you run a process with transactions, and in that case, the
# record is inserted only for the time it takes to close the
# database, re-open it (with transactions), then delete that
# syslog record. We're going to have to 86 that syslog record...
        delete from systables
```

```
            where rowid = sys_rowid
          end if
          close window wait_sys
          close database
          database dbname
        end if
        whenever error call error_handler
    end function
```

Before using these functions yourself, take heed of the following caveats:

1. The Informix `syslog` convention is not documented, and although there is no reason for them to change it, Informix remains open for changing database logging philosophies that may render these functions inoperable (the `syslog` convention is currently used in all Informix platforms);

2. You lose the `rollforward database` capabilities.

If you can live with these two caveats, then you can start enjoying the benefits of database transactions using these two routines. All 4GL programs that Fitrix produces utilize these routines because we demand the guarantee of data integrity without the overhead of the `rollforward database` function.

Issuing a Commit Work Without Closing the Cursor

Fitrix Accounting uses the `begin/commit/rollback work` statements to control application transactions. Application transactions are defined as the transactions of a business. Checks and invoices, for example, are complete transactions for applications. When these types of transactions are posted, they are done a batch at a time and a cursor is used to gather the data for posting.

A check (transaction) may take data from one to many of the rows retrieved from the cursor. Use the *work* concept for just those rows of the transaction (the check), rather than all rows in the cursor. A problem arises when you commit work for the first check. Doing so closes the main posting cursor (just what the documentation says it does). When you try reading in the next check from the cursor, a 4GL error occurs due to a closed cursor. To tackle the problem, you have to understand the nature of a cursor. A cursor is merely a method of gaining orderly access to several rows of data returned from an SQL statement. Routines like the following are used to work on cursors:

```
################################
function c1_define(sql_str)
################################
#
  define sql_str char(200)
  prepare get_curs from sql_str
  declare c1_curs scroll cursor
    for get_curs
  open c1_curs
end function
# c1_define()
################################
function c1_fetch(curs_num)
################################
#
  define curs_num integer
  fetch absolute curs_num c1_curs
    into c1_data.*
end function
# c1_fetch()
```

In the main body of the program, Fitrix posting programs rely on these types of routines to get data into the `c1_data.*` record (actually, these are shortened versions of the real thing, with a lot of the error-handling code removed). As discussed earlier, the first time a `commit work` statement is issued, you can no longer call on the `c1_fetch()` routine because the cursor is closed. The way to get around this is to create an entity called a *fake cursor*. A fake cursor is a temp table created by this process for the purpose of obtaining an orderly method of accessing data

returned from the SQL statement. A cursor must be defined in order to create the temp table, then the data is moved from the cursor to the temp table. The cursor is closed, and the `fetch` routine takes data from the temp table instead of the cursor. When the process is complete, the temp table is removed (or you can issue a `drop table` on the temp table when you are done with it). The temp table is then treated as the cursor. By using this strategy, you can keep your cursor open outside of `begin/commit work` statements.

Here is an example of the same routines with the fake cursor logic inserted. This routine lets `begin/commit/rollback transactions` to occur without closing the cursor:

```
##################################
function c1_define(sql_str)
##################################
#
  define sql_str char(200)
  prepare get_curs from sql_str
  declare c1_curs cursor
    for get_curs
  create temp table fake_curs (
    check_number char(10),
    doc_no integer,
    ...)
# the fake_curs table is defined to look like the data elements
# of the c1_curs cursor.
  foreach c1_curs into c1_data.*
    insert into fake_curs
    values(c1_data.*)
  end foreach
  close c1_curs
end function
# c1_define()

##################################
function c1_fetch(curs_num)
##################################
#
  define curs_num integer
  select * into c1_data.*
    from fake_cursor
    where rowid = curs_num
end function
# c1_fetch()
```

Note the following about the preceding functions:

- The define section opens the cursor just for the amount of time it takes to transfer the data in the cursor into the temp table.
- The temp table is defined with the exact same data elements as are defined in the `c1_curs` cursor.

- When the insert statement occurs on the temp table, the cursor rows are inserted sequentially and can be accessed by `rowid` (see `c1_fetch)` in the order that they were written to the temp table.
- The fetch function has been re-written to fill the `c1_data.*` record from the temp table rather than the cursor.

The above method for accessing data versus direct cursor access has the following disadvantages:

- It uses twice as much disk space for the same routine, since it must copy the cursor data to the temp table.
- The cursor definition routine is much slower since it must move all data from the real cursor into the fake cursor.
- When you change the layout of the cursor, you must remember to change the layout of the temp table to match the new cursor layout.

Although this is not the most graceful way to gain access to data returned from a select statement, if you really have to perform `begin/commit/rollback work` within large cursors, at least you have a method. Once understood, it is an easy concept to remember and it is proven in the field. This technique has been used in the Fitrix Accounting system, and although it is slow to startup, it gets the job done as advertised.

Moving Applications to Other Systems

To successfully run programs generated with the *Report* Code Generator on systems other than the development system, a few steps must be taken. These steps ensure that the tables, data, and reports your program needs to operate exist on the system to which you are transferring the program, and that your program knows where to find them.

The following steps are required to add the necessary tables to the application database:

1. Create the following directories on the target system:
   - `$fg/Make`
   - `$fg/bin`
   - `$fg/lib/data/library.dat`
   - `$fg/lib/data/library`
2. Copy the files in the following three directories from the development system, to the directories you created on the target system:
   - `$fg/bin`
   - `$fg/lib/data/library.dat`
   - `$fg/lib/data/library`
3. Change your `$PATH` variable on the target system to include the `$fg/bin` directory.
4. Be sure that each database to be converted is in the `$DBPATH`.
5. Run `mklib -dbname` *`database`* on each database that needs converted.

   The `mklb` script adds a number of `cg*` and `stx*` tables to your database.

If your application includes prompts or input forms, these steps are required to make forms available to the application:

1. Create a `$fg/lib/forms` directory on the target system.
2. Copy the files in the `$fg/lib/forms` directory on the development system into the `$fg/lib/forms` directory on the target system.
3. Add `$fg/lib/forms` to your `$DBPATH` on the target system.

The following is a list of the minimum files required to move your application from one system to another.

- .4gi and .frm files
- `$fg/lib/data/library.dat/*`
- `$fg/lib/data/library/dbmerge.4gi`
- `$fg/lib/forms/*.frm`
- `$fg/bin/mklib`
- `$fg/Make/*`
- Your startup scripts or custom runners

  `$fg/bin` needs to be in the `$PATH`

  `$fg/lib/forms` needs to be in the `$DBPATH`

Note

You can also use the optional variables `$fgmakedir`, `$fglibdir`, and `$fgtooldir` to point to your `Make` directory, upper-level libraries (`lib`) directory, and your Tools executable directory. Refer to "Optional Directory Variables" on page 1-6 for more information about these variables.

7

New Features and Functionality

This release of the *Report* Code Generator includes several new features and improved functionality.

This section covers these features, which include the following topics:

- Larger Selection Statement Variables
- Post Processor Flexibility
- Print Statement Block Tag Logic
- Custom Image File Block Tags
- Block Tags in `Makefile`
- Adding in Report Prompts

Larger Selection Statement Variables

The *Report* Code Generator now uses larger selection statement variables to prepare and declare selection statements. This increased size more than doubles previous capabilities.

Previous Variable Size	New Variable Size
`ct.sel_join char(200)`	`sel_join char(512)`
`ct.sel_filter char(200)`	`sel_filter char(2048)`
`ct.sel_order char(200)`	`sel_order char(512)`
`ct.sel_stmt char(1024)`	`sel_stmt char(4096)`

In addition these variables are now located in `midlevel.4gl` instead of `globals.4gl`, and the `ct.` has been dropped from their name. This move and subsequent name change gives you easier access and more control over the selection statement variables; you can increase their size whenever you near a limitation without having to change any libraries.

Backward Compatibility

An early work around to the limited size of selection statements used block commands and string replacements to increase selection limits. If you have programs that depend on this work around and you want to regenerate them, you can use an environment variable and an options file to maintain backward compatibility.

The `rpt_select` environment variable takes two options: `local` and `global`. The `global` option sets all variable values to their former size, name, and location. This is the option you should use to maintain backward compatibility. The `local` option sets all selection variables to their new size, name, and location.

You can set the `rpt_select` variable to `global` on a system, module (4gm) or program level (4gs). By default this variable is pre-set to `local`, so all selection variables start with the larger sizes.

To set on a system level, type `rpt_select="global"` in the `report.org` file. This file is located in the `$fg/codegen/options` directory.

To set on a module or program level, create a `report.opt` file in the directory that contains the programs you want to affect. In the `report.opt` file, add the `rpt_select="global"` line.

The ml_ct_sel_compat( ) Function

In addition to the new variables, a new function has also been added to `midlevel.4gl`. This function, named `ml_ct_sel_compat()`, sets local selection variables to any value passed from the command line or any other initializing method.

Post Processor Flexibility

The *Report* Code Generator lets you customize generated source code with a post processor. A post processor is useful for many tasks, such as making global changes to code, replacing or altering code blocks, and implementing bug fixes.

The `fg.report` program runs a post processor on the local application if the environment variable `$local_rpt` is set. Use this variable to point to the name of the program you wish to run on the generated 4GL code.

The same arguments that you pass to `fg.report` get passed to your post-processor program.

For example, assume you write an initialization function (say, `my_init()`) that is more relevant than the generic `init()` function that the Code Generator creates. You may want `main.4gl` to call `my_init()` rather than the `init()` function. You can set up a post processor to change the initialization call in `main.4gl` to `my_init()`.

To change `init()` in `main.4gl` to `my_init()`:

1. **Write a shell script (`chg_init`, for example) that uses the `sed` utility to remove `init()` and add a call to `my_init()`, such as:**

```
#chg_init
sed "s, call init, call my_init, " main.4gl > main.tmp &&
mv main.tmp main.4gl
```

2. **Set your `$local_rpt` environment variable to the name of the post-processor script (you might want to set this variable in your `.profile` file), for example:**

```
local_rpt=chg_init; export local_rpt
```

Once the Code Generator completes creating source (.4gl) files, your local `main.4gl` file contains the function call `my_init()` rather than `init()`.

Print Statement Block Tag Logic

In previous versions of the Fitrix *Report* Code Generator, the `report.4gl` file contained a block tag for each report row. These block tags, which were numbered sequentially, allowed you to create custom code for each line in the report. In other words, you could use an extension file and block command to act on any `report.4gl` print statement. The following shows an example of the old logic:

Print Statement Block Tags in report.4gl

```
format
   page header
        whenever error call error_handler
        #_print_1 - tag for the following print statement
        print
            column 1, today using "mm/dd/yy";
            let scratch = "CUSTOMER ORDER HISTORY LIST"
            let x = 40 - (length(scratch) / 2)
            print
                column x, scratch clipped;
            let scratch = pageno using "Page: <<<<"
            let x = 81 - length(scratch)
            print
                column x, scratch clipped
        #_print_2 - tag for the following print statement
        print
        #_print_3 - tag for the following print statement
        print
            column 1, "Customer"
        #_print_4 - tag for the following print statement
        print
            column 1, "Number",
            column 11, "Name",
            column 30, "Company",
            column 54, "Phone"
        #_print_5 - tag for the following print statement
        print
            column 1, "----------------------------------------",
            column 41, "-----------------------------------"

page trailer
        #_print_6 - tag for the following print statement
        print
        #_print_7 - tag for the following print statement
        print
            column 1, "----------------------------------------",
            column 41, "-----------------------------------"
        #_print_8 - tag for the following print statement
        print
            column 1, pageno using "Page: <<<<";
            let scratch = pageno using "Page: <<<<"
            let x = 81 - length(scratch)
            print
                column x, scratch clipped
```

This version of the *Report* Code Generator uses new print statement block tag logic. Instead of the entire `report.4gl` file having sequentially numbered print statement block tags, this version numbers print statement block tags according to the control block that contains them. In other words, the numbering scheme restarts at one for each control block. The new code looks as follows:

```
format
  #_format - Format section
      #_page_header - Report block for page header.
          #_err - Trap fatal errors.
          whenever error call error_handler
          #_page_header_1 - Print statement
          call put_vararg(today using usg.today)
          call put_vararg("CUSTOMER ORDER HISTORY LIST")
          call put_vararg(pageno using usg.pageno)
          let header_image = imageManager_getLine(1)
          print header_image clipped
          #_page_header_2 - Print statement
          print
          #_page_header_3 - Print statement
          let header_image = imageManager_getLine(2)
          print header_image clipped
          #_page_header_4 - Print statement
          let header_image = imageManager_getLine(3)
          print header_image clipped
          #_page_header_5 - Print statement
          let header_image = imageManager_getLine(4)
          print header_image clipped
      #_end - End of report block.

      #_page_trailer - Report block for page trailer.
          #_page_trailer_1 - Print statement
          print
          #_page_trailer_2 - Print statement
          let header_image = imageManager_getLine(5)
          print header_image clipped
          #_page_trailer_3 - Print statement
          call put_vararg(pageno using usg.pageno)
          call put_vararg(pageno using usg.pageno)
          let header_image = imageManager_getLine(6)
          print header_image clipped
      #_end - End of report block
```

Print Statement Block Tags for page header Control Block

Print Statement Block Tags for page trailer Control Block

As you can see, this new scheme limits the number of lines affected when you insert a custom print statement.

For instance, suppose you create a report that contains 30 print statements. When the *Report* Code Generator constructs your source code (.4gl) files, it adds 30 print statement block tags to your `report.4gl` file, one tag for each print statement. The old method numbers these block tags from one to 30. As mentioned above, the new numbering scheme restarts at one for each control block.

Now suppose this same report program uses several extension files containing block commands that act on these print statement block tags. Using the old method, if a situation arises where you need to remove a print statement, all your extension files must be reworked. However, with the new method, you only need to alter the extension files that reference the modified control block.

Backward Compatibility

Although this new numbering scheme makes modifying reports easier, you can regenerate your existing report programs with the old method. The *Report* Code Generator uses a new variable that controls which method gets used, namely `rpt-tagtype`. This variable, which you set on a local basis in the `report.opt` file and globally the `$fg/codegen/options/report.org` file, accepts one of two values: `block` or `line`. By default, the *Report* Code Generator uses the new print statement block tag logic, in other words `rpttagtype` is set to `block`. If you want to use the old method, set `rpttagtype` to `line`.

As a rule, set `rpttagtype` to `line` when you are working with previously generated programs. In the `report.opt` file type:

```
rpttagtype=line
```

When developing new report programs, use the new numbering scheme to take advantage of the simplified code and regenerability.

Custom Image File Block Tags

Another new feature of the *Report* Code Generator is custom image file block tags. You can now add your custom block tags in the format section of the image file. These block tags, upon code generation, get placed into the `report.4gl` file. Once in the `report.4gl` file, you can use extension files and block statements to customize the source code. For example, the following image (`report.ifg`) file contains a custom block tag in the `before group` control block:

```
before group of customer.customer_num
{
#_custom_block - Example custom block tag
[f000     [f001             [f002                  [f008
   Activity Report:                                      [!
      Order #  PO #       Ordered        Shipped        Paid         [!
      ------------------------------------------------------------------  [!
}
```

Custom Block Tag

When this example image file is generated into source code, the custom block tag gets added to the appropriate line in the `report.4gl` file:

```
        before group of rpt.customer_num
        #_b_customer_num - Before group customer_num

            if
                rpt.order_num is not null
             or rpt.po_num is not null
             or rpt.order_date is not null
             or rpt.ship_date is not null
             or rpt.paid_date is not null
            then
                let dynamic = true
                need 4 lines
            else
                let dynamic = false
            end if

            #_custom_block - Example custom block tag
            call put_vararg(rpt.customer_num using usg.customer_num)
            call put_vararg(rpt.fname)
            call put_vararg(rpt.company)
            call put_vararg(rpt.phone)
            let line_image = imageManager_getLine(7)
            print line_image clipped
```

Custom Block Tag Added to report.4g

Custom tags are useful, because they never change. Whereas print statement block tags change when new lines are added, custom tags are completely regenerable. When you create extension files that act on custom block tags, your modifications are preserved each time you regenerate the code.

Numbering Scheme Variable

When you create custom block tags in your `report.ifg` file, you must decide how you want the print statement block tags in the `report.4gl` file to increment. There are two increment methods. The first method counts your custom block tags; in other words, it is an *absolute* numbering scheme. The second method does not count your custom block tags, thus the print statement block tags remain consistent no matter how many custom block tags you add to your `report.ifg` file. This second method is known as *relative*.

In almost every case, the *Report* Code Generator defaults to the method appropriate to your situation. However, at times you might want to specify the non-default method.

To specify the non-default method, use the new `rpttagnum` variable. You can set this variable to either `absolute` or `relative`. On the local directory level, use the `report.opt` file to set this variable. For example, to set the `rpttagnum` to `absolute`, add the following line to the `report.opt` file:

```
rpttagnum=absolute
```

On a global basis, set this variable in the `$fg/code-gen/options/report.org` file.

Note

The `rpttagnum` variable defaults to a different value depending on the value in the `rpttagtype` variable. When `rpttagtype` is set to `block`, `rpttagnum` defaults to `relative`. When `rpttagtype` is set to `line`, `rpttagnum` defaults to `absolute`. For more on the `rpttagtype` variable, refer to "Print Statement Block Tag Logic" on page 7-5.

The following lines of code show an example of both `absolute` and `relative` numbering schemes, notice how the number of the third print statement block tag differs between the two methods:

Absolute Block Tag Numbering Scheme

```
#_page_header - Report block for page header.

    #_err - Trap fatal errors.
```

```
    whenever error call error_handler

    #_page_header_1 - Print statement
    call put_vararg(today using usg.today)
    call put_vararg("Syscolumns ")
    call put_vararg(pageno using usg.pageno)
    let header_image = imageManager_getLine(1)
    print header_image clipped

    #_custom_tag - Custom block tag
    let header_image = imageManager_getLine(2)
    print header_image clipped

    #_page_header_3 - Print statement
    let header_image = imageManager_getLine(3)
    print header_image clipped

    #_page_header_4 - Print statement
    let header_image = imageManager_getLine(4)
    print header_image clipped

    #_page_header_5 - Print statement
    print

    #_page_header_6 - Print statement
    print

#_end - End of report block.
```

Relative Block Tag Numbering Scheme

```
#_page_header - Report block for page header.

    #_err - Trap fatal errors.
    whenever error call error_handler

    #_page_header_1 - Print statement
    call put_vararg(today using usg.today)
    call put_vararg("Syscolumns ")
    call put_vararg(pageno using usg.pageno)
    let header_image = imageManager_getLine(1)
    print header_image clipped

    #_custom_tag - Custom block tag
    let header_image = imageManager_getLine(2)
    print header_image clipped

    #_page_header_2 - Print statement
    let header_image = imageManager_getLine(3)
    print header_image clipped

    #_page_header_3 - Print statement
    let header_image = imageManager_getLine(4)
    print header_image clipped

    #_page_header_4 - Print statement
    print

    #_page_header_5 - Print statement
    print

#_end - End of report block.
```

A good time to use the non-default method is when you add a custom image file block tag to a report that uses the old print statement logic (i.e., `rpttag-type=line`), and you regenerate that report. In this case, you do not want `rpt-tagnum` set to `absolute`, because `absolute` numbers each block tag including your custom tag, which in turn throws off every extension file that acts on the changed tag numbers. Instead, set `rpttagnum` to `relative`. This setting preserves all the existing print statement block tags. In essence this setting numbers around your custom tag.

Block Tags in Makefile

As you might have noticed already, the `Makefile` now comes with generated block tags before each `Makefile` macro. The new `Makefile` style makes including custom libraries and source files into your generated programs easier. In addition, you inherit all the flexibility and functionality of block statements and extension files, such as pluggable features and feature sets, without having to use difficult-to-maintain string replacement logic in block statements. The following example `Makefile` illustrates the new block tags:

```
########################################################################
# Copyright (C) 1993 <Your Company Name>
# All rights reserved.
# Use, modification, duplication, and/or distribution of this
# software is limited by the software license agreement.
# Sccsid: %Z% %M% %I% Delta: %G%
########################################################################
# Makefile for an Informix report

#_type - Makefile type
TYPE = program

#_name - program name
NAME = tmp.4ge

#_objfiles - program files
OBJFILES = globals.o lowlevel.o main.o midlevel.o report.o

#_forms - perform files
FORMS =

#_libfiles - library list
LIBFILES = ../lib.a \
           $(fg)/lib/report.a \
           $(fg)/lib/user_ctl.a \
           $(fg)/lib/standard.a

#_globals - globals file
GLOBAL = globals.4gl

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

#_all_rule - program compile rule
all:
        @echo "make: Cannot use make. Use fg.make -F for 4GL compile."
```

Adding in Report Prompts

As discussed in "Designing Report Prompts" on page 6-2, many reports require the end-user to enter selection criteria in a prompt prior to running a report. If you own the Fitrix *Screen* Code Generator and Form Painter, you can use the following extension file to *automatically* hook in these prompts. Follow the general steps outlined below to link in your query screens:

1. Create a query screen in the Fitrix *Screen* Painter.
2. Run the Fitrix *Screen* Code Generator.
3. Create an image (`report.ifg`) file for your report.
4. Run the *Report* Code Generator.
5. Create the extension file (shown below) and call it in the `base.set` file.
6. Run `fg.make` to build the report program.

Before following any of these steps, you should reference "Customizing Reports" on page 5-1. In that section you will find information that describes merging custom code into generated code, using the Featurizer, and making your programs regenerable.

Report Prompt Extension File

Add the following lines of code to an extension file. Notice that you must supply the name of your prompt screen in three locations within this extension file. The italicized word *`screen_name`* denotes where you should enter the name of your prompt screen less the .per extension.

```
#---------------------------------------------------------------------
start file "Makefile"
#---------------------------------------------------------------------

  libraries
    $(fg)/lib/scr.a;

#---------------------------------------------------------------------
start file "midlevel.4gl"
#---------------------------------------------------------------------

function define ml_filter
   m smallint,
   n smallint;
```

```
replace block ml_filter sel_filter

   call socketManager("screen_name", "query", "default")
   let sel_filter = null
   let n = fgStack_pop()
   if n = 0
   then
       let int_flag = true
       # set default filter if user continues
       let sel_filter = "1=1"

   else
       for m = 1 to n
           let sel_filter = sel_filter clipped, fgStack_pop()
       end for
   end if;

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

  after block main after_init
    let scr_id = "default"
    ;

  at_eof

    ###########################################################################
    function switchbox(funct)
    ###########################################################################
    # This is the switchbox function for version 4.11.UC1 screens.
    # It is used to pass flow control to the appropriate screen functions.
    #
        #_define_var - define local variables
        define
            #_local_var - local variables
            funct char(20)   # Function to pass on to the screen

        #_post_scr_funct -  Post the current function
        let scr_funct = funct

        #_switchbox -  Pass flow control to appropriate screen
        case
          # put your screen in here
          when scr_id = "screen_name" call S_screen_name()
          when scr_id = "default" call lib_screen()
          #_otherwise - otherwise clause
          otherwise let scratch = "no screen"
        end case

        #_scr_funct - Reset scr_funct upon return
        let scr_funct = ""

    end function
    # switchbox()
    ;
```

8

Report Examples

Sometimes the best way to learn is by example. This chapter illustrates an image file and other 4GL source code files that can help you develop your own report programs. Study these examples and try them yourself.

This section contains examples of the following files:

- Report Image File
- Source Code .4gl Files
- Report Output

Report.ifg

```
database standard

output
    top margin         0
    bottom margin      0
    left margin        0
    right margin      80
    page length       66

page header
{
<A1    ]                +A2                           ]               Page: >A3 ]
================================================================================
}
on every row
{

Cust No         [A4      ]
                                                      [*
Last Name       [A5           ]
                                                      [*
Company Name    [A6                                   ]
                                                      [*
Order No        [A7      ]
                                                      [*
Order Date      [A8    ]
                                                      [*
Pay Date        [A9    ]
                                                      [*
Address Line #1 [B1                ]
                                                      [*
}

attributes
    A1 = date using "mm/dd/yy", name=HD_date
    A2 = constant "Customer Activity             ", name=HD_title
    A3 = pageno using "Page: <<<<<", name=HD_page
    A4 = customer.customer_num, name=customer.customer_num
    A5 = customer.lname, name=customer.lname
    A6 = customer.company, name=customer.company
    A7 = orders.order_num, name=orders.order_num
    A8 = orders.order_date, name=orders.order_date
    A9 = orders.paid_date, name=orders.paid_date
    B1 = customer.address1, name=customer.address1

select
    tables = customer, orders
    join   =  orders.customer_num = customer.customer_num

defaults
    progname  = query
    prcname   = Customer Activity
    destin    = report.out

data group Demo Data Set
    tbls   = customer orders items stock manufact state
    lang   = ENG
```

Globals.4gl

```
database standard

globals
    define
    #_define
        #_define_0
        #_end
        #_using - Dynamic 'using' variables
        usg record
            today char(16),  # Default: "mm/dd/yy"
            pageno char(22)  # Default: "Page: <<<<<"
        end record,

        #_rpt_rec - report record
        rpt record                  # record for the report
            customer_num like customer.customer_num,
            lname like customer.lname,
            company like customer.company,
            order_num like orders.order_num,
            order_date like orders.order_date,
            paid_date like orders.paid_date,
            address1 like customer.address1
        end record,

        #_curs_rec - cursor current record
        curs record                 # record like the cursor
            customer_num like customer.customer_num,
            lname like customer.lname,
            company like customer.company,
            order_num like orders.order_num,
            order_date like orders.order_date,
            paid_date like orders.paid_date,
            address1 like customer.address1
        end record,

        #_curs_next_rec - cursor next record
        curs_next record            # next row of cursor
            customer_num like customer.customer_num,
            lname like customer.lname,
            company like customer.company,
            order_num like orders.order_num,
            order_date like orders.order_date,
            paid_date like orders.paid_date,
            address1 like customer.address1
        end record,

        #_ct_rec - control record
        ct record                   # control record (don't change)
            prcname      char(35),  # process name (message on upper left)
            rtmargn      char(35),  # message on upper right margin
            prc_only     char(1),   # process only? (no report) "y" or "n"
            allow_int    char(1),   # allow interrupts? "y" or "n"
            quiet        smallint,  # display every "quiet" rows
            destin       char(150), # "screen", "printer", "pipe", or "filename"
            sel_join     char(200), # join portion of selection stmt
            sel_filter   char(200), # filter portion of selection stmt
            sel_order    char(200), # order by clause for above
            sel_stmt     char(1024),# select statement for report cursor
            num_rows     integer,   # number of rows in the cursor to process
            cur_row      integer    # current row being processed
        end record,

        #_communication - communication area
        ######################################################################
        # Library communication area 4.10.UC1
```

```
        ###################################################################
        # Global variables in this section should not be changed.
        # They are used to communicate to the screen library functions,
        # and must be of the same type as defined in the library.
        # Don't remove these comments.  The codegenerator keys on them.
        #
        progid       char(17),   # Program identification
        scr_id       char(7),    # Current screen id
        menu_item    char(10),   # Current menu item running
        scr_funct    char(20),   # Current screen function being run
        sql_filter   char(512),  # Filter portion of SQL statement
        sql_order    char(100),  # Order portion of SQL statement
        input_num    smallint,   # Current input section within screen
        p_cur        smallint,   # Current input array element
        s_cur        smallint,   # Current screen array element
        scr_fld      char(40),   # Current screen field
        nxt_fld      char(40),   # Programmatic next screen field
        prev_data    char(80),   # Data before field entry
        this_data    char(80),   # Data after field entry
        data_changed smallint,   # Has the field data changed?
        hotkey       smallint,   # The hot key that has been pressed
        scratch      char(2047)  # Scratchpad for scribbling on and
                                 # communicating between functions
        # End library communication area
        ###################################################################

end globals
```

Lowlevel.4gl

```
globals "globals.4gl"

#_local_static - Local (static) variable definition


###########################################################################
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

    #_err - Trap fatal errors
    whenever error call error_handler

    #_b_first_row - Before first row processing

end function
# b_g_first_row()

###########################################################################
function on_every_row()
###########################################################################
# This function prepares the report record from the
# cursor record and other data.
#

    #_define_var - Define local variables

    #_before_every_row - Before on every row assignments

    #_customer_num - On every row processing for customer_num
    let rpt.customer_num = curs.customer_num

    #_lname - On every row processing for lname
    let rpt.lname = curs.lname

    #_company - On every row processing for company
    let rpt.company = curs.company

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

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

    #_paid_date - On every row processing for paid_date
    let rpt.paid_date = curs.paid_date

    #_address1 - On every row processing for address1
    let rpt.address1 = curs.address1

    #_after_every_row - After on every row assignments

end function
# on_every_row()

###########################################################################
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

    #_a_last_row - After last row processing

end function
# a_g_last_row()
```

Main.4gl

```
globals "globals.4gl"

#_local_static - Local (static) variable definition.


########################################################################
main
########################################################################
#

    #_define_var - Define local variables

    #_err - Trap fatal errors
    whenever error call error_handler

    #_set_up - Basic set up
    clear screen
    defer interrupt

    #_errlog - Start the error log
    call startlog("errlog")

    #_program_id - Set program id
    let progid = "brianh.query"

    #_open_window - Open window
    open window win1 at 2,3 with 22 rows, 76 columns
        attribute (border, red)

    #_put_scrlib - Calls to put_scrlib()
    call put_scrlib("dbname","")

    #_before_init - Before initialization processing
    call init()
    #_after_init - After initialization processing

    #_clear_window - Clear the window
    clear window win1

    #_flow_control - Call flow control
    call flow_control()

    #_exit_program - Exit program
    exit program (0)

end main


########################################################################
function logo()
########################################################################
#
    #_logo - insert your logo here.
    # This is the format of the company logo:
    #
    # display "PROGRAM NAME" at 4,3 attribute(blue)
    # display "Copyright (c) 1992" at 7,3 attribute(blue)
    # display "Your Company Name" at 9,3 attribute(blue)
    # display "Seattle, Washington  USA" at 11,3 attribute(blue)
    display "Loading Program..." at 16,3 attribute(blue)
    #_logo_sleep - insert the sleep for your logo here.
    sleep 2
end function
# logo()
```

Midlevel.4gl

```
globals "globals.4gl"

#_local_static - Local (static) variable definition
define
    #_misc_static - Misc static variables
    sel_join     char(512),  # join for selection stmt
    sel_filter   char(2048), # filter for selection stmt
    sel_order    char(512)   # order for selection stmt


############################################################################
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 =
        "orders.customer_num = customer.customer_num"

end function
# ml_join()

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

    #_define_var - Define local variables

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

end function
# ml_filter()

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

    #_define_var - Define local variables

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

end function
# ml_order()

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

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

    #_getcount - Build select statement for getcount
```

```
    let sel_stmt =
        "select count(*) ",
          "from ",
            "customer, orders ",
          "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

        #_declare_curs - Declare cursor from the string
        declare count_cursor cursor with hold for get_count

        #_read_data - Read the data
        open count_cursor

        #_fetch - Fetch statement
        fetch count_cursor into ct.num_rows

        #_close - Close the cursor
        close count_cursor

end function
# ml_getcount()

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

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

    #_define_cur - Build select statement for define_cur
    let sel_stmt =
        "select ",
            "customer.customer_num, ",
            "customer.lname, ",
            "customer.company, ",
            "orders.order_num, ",
            "orders.order_date, ",
            "orders.paid_date, ",
            "customer.address1 ",
          "from ",
            "customer, orders ",
          "where ",
            "(", sel_join clipped, ")",
            " and (", sel_filter clipped, ")"
      #_include_order - Include any valid order criteria
      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 - Fetch statement
    fetch rpt_cursor into curs_next.*

end function
# ml_fetch()

###############################################################################
function ml_curs_null()
###############################################################################
# This function sets all of the elements in the curs
# record to null.
#

    #_define_var - Define local variables

    #_curs_null - Initialize the curs record to null
    initialize curs.* to null

end function
# ml_curs_null()

###############################################################################
function ml_next_null()
###############################################################################
# This function sets all of the elements in the
# curs_next record to null.
#

    #_define_var - Define local variables

    #_curs_null - Initialize the curs record to null
    initialize curs_next.* to null

end function
# ml_next_null()

###############################################################################
function ml_curs_prep()
###############################################################################
# This function transfers data from the curs_next
# record to the curs record.
#

    #_define_var - Define local variables

    #_set_curs - Set curs record equal to curs_next
    let curs.* = curs_next.*
```

```
end function
# ml_curs_prep()

###############################################################################
function ml_before_break()
###############################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        bk_level    smallint    # break level setting

    #_return - Return if no break is required
    return

    #_action_label - Perform the correct break logic
    label bf_break_action:

end function
# ml_before_break()

###############################################################################
function ml_after_break()
###############################################################################
#

    #_define_var - Define local variables
    define
        #_local_var - Local variables
        bk_level    smallint    # break level setting

    #_return - Return if no break is required
    return

    #_action_label - Perform the correct break logic
    label af_break_action:

end function
# ml_after_break()

###############################################################################
function ml_output()
###############################################################################
#

    #_define_var - Define local variables

    #_output - Send output to report
    output to report report1 (rpt.*, usg.*)

end function
# ml_output()

###############################################################################
function ml_defaults()
###############################################################################
#

    #_define_var - Define local variables

    #_def_prcname - Default "prcname" value
    let ct.prcname = "Customer Activity"

    #_def_destin - Default "destin" value
    let ct.destin = "report.out"
```

```
end function
# ml_defaults()

###############################################################################
function ml_ct_sel_compat()
###############################################################################
#

    #_define_var - Define local variables

    #_set_sel_join - Pass any ct.sel_join value to local variable
    let sel_join = ct.sel_join clipped

    #_set_sel_filter - Pass any ct.sel_filter value to local variable
    let sel_filter = ct.sel_filter clipped

    #_set_sel_order - Pass any ct.sel_order value to local variable
    let sel_order = ct.sel_order clipped

end function
# ml_ct_sel_compat()
```

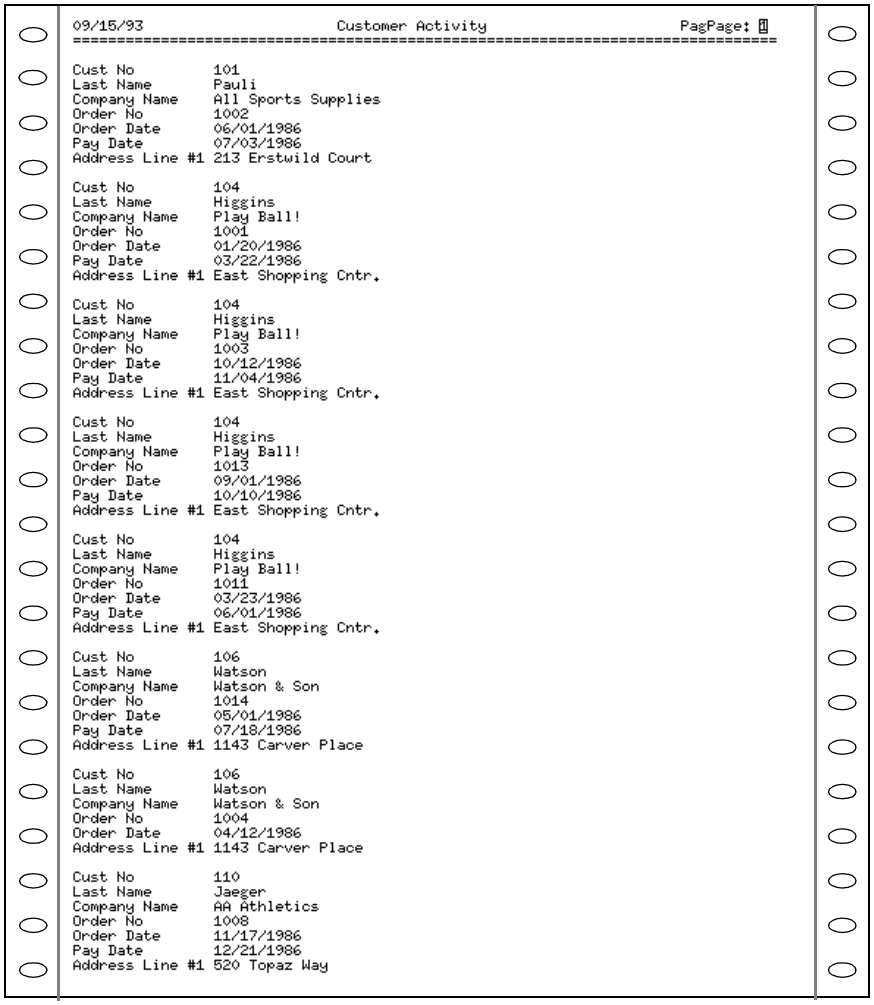
Report.4gl

```
globals "globals.4gl"

########################################################################
report report1(rpt, usg)
########################################################################
#
    #_define_var - Define local variables.
    define
        #_rpt_rec - Report record.
        rpt record                  # Record for the report
            customer_num like customer.customer_num,
            lname like customer.lname,
            company like customer.company,
            order_num like orders.order_num,
            order_date like orders.order_date,
            paid_date like orders.paid_date,
            address1 like customer.address1
        end record,
        #_end - End report record

        #_using - Dynamic 'using' variables
        usg record
            today char(16),  # Default: "mm/dd/yy"
            pageno char(22)  # Default: "Page: <<<<<"
        end record,
        #_end - End usg record

        header_image char(255),     # Header line image
        line_image char(255),       # Non-header line image
        dynamic     smallint,       # Boolean: Print this line?
        x           smallint        # Working number

    #_end - End of section.
    output
    #_output - Output section.

        #_def_top - insert "top" code here
        top margin         0

        #_def_bottom - insert "bottom" code here
        bottom margin      0

        #_def_left - insert "left" code here
        left margin        0

        #_def_right - insert "right" code here
        right margin      80

        #_def_page - insert "page" code here
        page length       66

    #_end - End of section
    format
    #_format - Format section

        page header
        #_page_header - Report block for page header.

            #_err - Trap fatal errors.
            whenever error call error_handler

            #_page_header_1 - Print statement
            call put_vararg(today using usg.today)
            call put_vararg("Customer Activity             ")
            call put_vararg(pageno using usg.pageno)
```

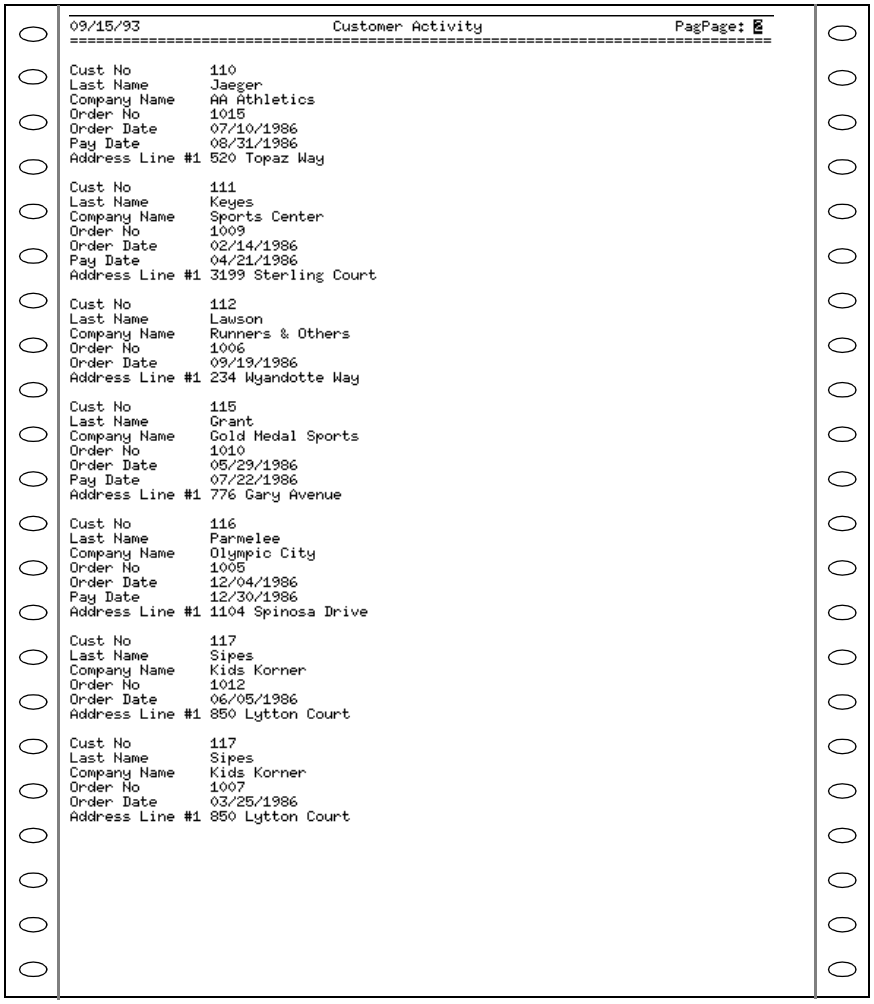
```
    let header_image = imageManager_getLine(1)
    print header_image clipped

    #_page_header_2 - Print statement
    let header_image = imageManager_getLine(2)
    print header_image clipped

#_end - End of report block.

on every row
#_on_every_row - Report block for on every row.

    need 8 lines

    #_on_every_row_1 - Print statement
    print
    if
        rpt.customer_num is not null
    then

        #_on_every_row_2 - Print statement
        call put_vararg(rpt.customer_num)
        let line_image = imageManager_getLine(3)
        print line_image clipped
    end if
    if
        rpt.lname is not null
    then

        #_on_every_row_3 - Print statement
        call put_vararg(rpt.lname)
        let line_image = imageManager_getLine(4)
        print line_image clipped
    end if
    if
        rpt.company is not null
    then

        #_on_every_row_4 - Print statement
        call put_vararg(rpt.company)
        let line_image = imageManager_getLine(5)
        print line_image clipped
    end if
    if
        rpt.order_num is not null
    then

        #_on_every_row_5 - Print statement
        call put_vararg(rpt.order_num)
        let line_image = imageManager_getLine(6)
        print line_image clipped
    end if
    if
        rpt.order_date is not null
    then

        #_on_every_row_6 - Print statement
        call put_vararg(rpt.order_date)
        let line_image = imageManager_getLine(7)
        print line_image clipped
    end if
    if
        rpt.paid_date is not null
    then

        #_on_every_row_7 - Print statement
        call put_vararg(rpt.paid_date)
        let line_image = imageManager_getLine(8)
```

```
            print line_image clipped
        end if
        if
            rpt.address1 is not null
        then

            #_on_every_row_8 - Print statement
            call put_vararg(rpt.address1)
            let line_image = imageManager_getLine(9)
            print line_image clipped
        end if

    #_end - End of report block.

end report
# report1()

#########################################################################
function load_image()
#########################################################################
# This function loads the report image into the imageManager package.
#
call imageManager_setRightMargin(80)
call image_name("brianh/query.4gs")
call image_line( 1, "<                      +
          Page: >")
call image_line( 2, "============================================================
====================")
call image_line( 3, "Cust No         [")
call image_line( 4, "Last Name       [")
call image_line( 5, "Company Name    [")
call image_line( 6, "Order No        [")
call image_line( 7, "Order Date      [")
call image_line( 8, "Pay Date        [")
call image_line( 9, "Address Line #1 [")

#_using - Default the dynamic using variables
let usg.today = "mm/dd/yy"
let usg.pageno = "Page: <<<<<"

end function
# load_image()
```

Report Output



```
09/15/93                        Customer Activity                        PagPage: 1
================================================================================

Cust No         101
Last Name       Pauli
Company Name    All Sports Supplies
Order No        1002
Order Date      06/01/1986
Pay Date        07/03/1986
Address Line #1 213 Erstwild Court

Cust No         104
Last Name       Higgins
Company Name    Play Ball!
Order No        1001
Order Date      01/20/1986
Pay Date        03/22/1986
Address Line #1 East Shopping Cntr.

Cust No         104
Last Name       Higgins
Company Name    Play Ball!
Order No        1003
Order Date      10/12/1986
Pay Date        11/04/1986
Address Line #1 East Shopping Cntr.

Cust No         104
Last Name       Higgins
Company Name    Play Ball!
Order No        1013
Order Date      09/01/1986
Pay Date        10/10/1986
Address Line #1 East Shopping Cntr.

Cust No         104
Last Name       Higgins
Company Name    Play Ball!
Order No        1011
Order Date      03/23/1986
Pay Date        06/01/1986
Address Line #1 East Shopping Cntr.

Cust No         106
Last Name       Watson
Company Name    Watson & Son
Order No        1014
Order Date      05/01/1986
Pay Date        07/18/1986
Address Line #1 1143 Carver Place

Cust No         106
Last Name       Watson
Company Name    Watson & Son
Order No        1004
Order Date      04/12/1986
Address Line #1 1143 Carver Place

Cust No         110
Last Name       Jaeger
Company Name    AA Athletics
Order No        1008
Order Date      11/17/1986
Pay Date        12/21/1986
Address Line #1 520 Topaz Way
```



```
09/15/93                        Customer Activity                       PagPage: 2
================================================================================

Cust No         110
Last Name       Jaeger
Company Name    AA Athletics
Order No        1015
Order Date      07/10/1986
Pay Date        08/31/1986
Address Line #1 520 Topaz Way

Cust No         111
Last Name       Keyes
Company Name    Sports Center
Order No        1009
Order Date      02/14/1986
Pay Date        04/21/1986
Address Line #1 3199 Sterling Court

Cust No         112
Last Name       Lawson
Company Name    Runners & Others
Order No        1006
Order Date      09/19/1986
Address Line #1 234 Wyandotte Way

Cust No         115
Last Name       Grant
Company Name    Gold Medal Sports
Order No        1010
Order Date      05/29/1986
Pay Date        07/22/1986
Address Line #1 776 Gary Avenue

Cust No         116
Last Name       Parmelee
Company Name    Olympic City
Order No        1005
Order Date      12/04/1986
Pay Date        12/30/1986
Address Line #1 1104 Spinosa Drive

Cust No         117
Last Name       Sipes
Company Name    Kids Korner
Order No        1012
Order Date      06/05/1986
Address Line #1 850 Lytton Court

Cust No         117
Last Name       Sipes
Company Name    Kids Korner
Order No        1007
Order Date      03/25/1986
Address Line #1 850 Lytton Court
```

Index

Symbols

A

B

C

D

E

F

G

I

S

T

V