Fitrix™

Screen

Technical Reference

Version 4.11

Restricted Rights Legend

Copyright

Software License Notice

Licenses and Trademarks

FITRIX MANUALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, FURTHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE FITRIX MANUALS IS WITH YOU. SHOULD THE FITRIX MANUALS PROVE DEFECTIVE, YOU (AND NOT FOURTH GENERATION SOFTWARE OR ANY AUTHORIZED REPRESENTATIVE OF FOURTH GENERATION SOFTWARE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION IN NO EVENT WILL FOURTH GENERATION BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH FITRIX MANUALS, EVEN IF FOURTH GENERATION OR AN AUTHORIZED REPRESENTATIVE OF FOURTH GENERATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. IN ADDITION, FOURTH GENERATION SHALL NOT BE LIABLE FOR ANY CLAIM ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH FOURTH GENERATION SOFTWARE OR MANUALS BASED UPON STRICT LIABILITY OR FOURTH GENERATION'S NEGLIGENCE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

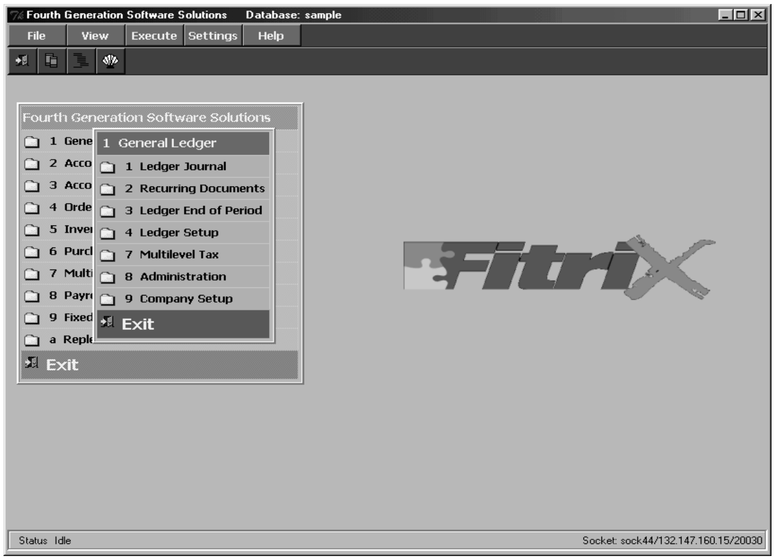
Fourth Generation Software Solutions
2814 Spring Road, Suite 300
Atlanta, GA 30339

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

Copyright

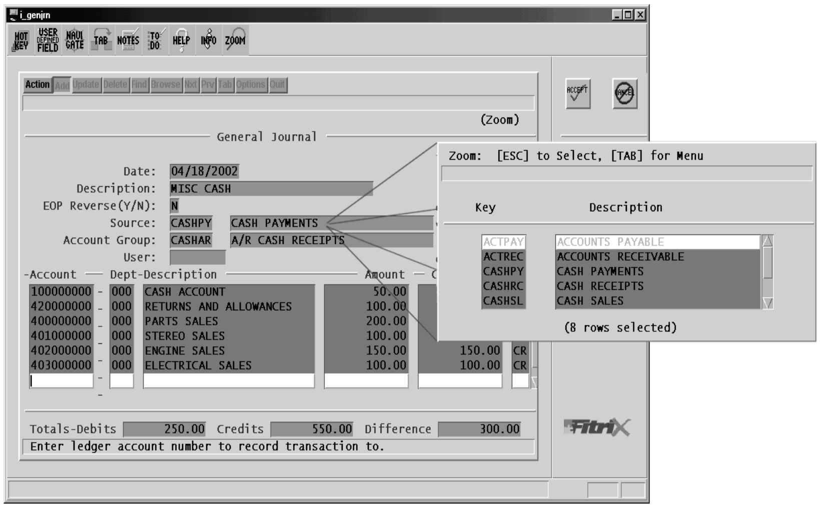
Welcome to the Fitrix Screen Technical Reference. This manual is designed to be a focused step-by-step guide. We hope that you find all of this information clear and useful.

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:



```
                                                      (Zoom)
 ──────────────────── General Journal ─────────────────────

              Date:  04/18/2002
       Description:  MISC CASH
   EOP Reverse(Y/N): N
            Source:  CASHPY   CASH PAYMENTS
     Account Group:  CASHAR   A/R CASH RECEIPTS
              User:
-Account ── Dept-Description ─────────────── Amount ─ C
 100000000 - 000 CASH ACCOUNT                 50.00
 420000000 - 000 RETURNS AND ALLOWANCES      100.00
 400000000 - 000 PARTS SALES                 200.00
 401000000 - 000 STEREO SALES                100.00
 402000000 - 000 ENGINE SALES                150.00    150.00 CR
 403000000 - 000 ELECTRICAL SALES            100.00    100.00 CR

 Totals-Debits     250.00  Credits     550.00  Difference     300.00
 Enter ledger account number to record transaction to.
```

```
Zoom:  [ESC] to Select, [TAB] for Menu

    Key            Description

    ACTPAY         ACCOUNTS PAYABLE
    ACTREC         ACCOUNTS RECEIVABLE
    CASHPY         CASH PAYMENTS
    CASHRC         CASH RECEIPTS
    CASHSL         CASH SALES

              (8 rows selected)
```

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

Part 1: Introduction to Fitrix Screen

Chapter 1: Introduction

Chapter 2: Getting Started

Chapter 3: The Data-Entry Interface

Part 2: The Form Painter

Chapter 4: Form Painter Basics

Chapter 5: Managing Forms

Chapter 6: Editing Forms

Chapter 7: Form Definition

Chapter 8: The Run Menu

Chapter 9: Database Administration

Part 3: The Code Generator

Chapter 10: Creating Screen Forms

Chapter 11: Source Code

Chapter 12: Customizing Your Base Program With Triggers

Chapter 13: The Featurizer and Blocks

Chapter 14: Compiling and Running Your Programs

Chapter 15: Advanced Features

Chapter 16: Version Control

Chapter 17: Language Translation

Chapter 18: Helpful Techniques

Part 4: Appendixes

Appendix A: Fitrix Screen Utilities

Appendix B: The .per Specification File

Appendix C: Program Migration

Appendix D: Fitrix Screen Tables

Appendix E: Control Key Defaults

Appendix F: Reserved Terms and Style Guide

Appendix G: Termcaps

Part One

Introduction to Fitrix Screen

1

Introduction

This chapter introduces you to Fitrix *Screen* and covers the following:

- Technical merits
- Overview of the Fitrix *Screen* product
- Introduction to the four major components of Fitrix *Screen*
- Features and capacities

Technical Merits

Fitrix *Screen* CASE Tools provide a complete application development system. Fitrix *Screen* offers significant technological advantages over any other application development product on the market. Fitrix *Screen* gives you the power, speed, and flexibility you need to create your applications.

- **Program maintainability** - One of the biggest costs associated with computer software is maintenance. Traditional software is so expensive to maintain because it is not designed with maintainability in mind. Only the original programmers can truly understand all of the complexities involved with "spaghetti code."

  Fitrix *Screen* takes the headache out of modifying your programs. Fitrix *Screen* creates highly commented and thoroughly documented code that is logically organized into functional objects. Fitrix *Screen* generated programs are also 100% regenerable. You need to add a field to your program? Simply run the Form Painter, add the field to your screen, define the field, regenerate the screen, then recompile the program. It is that easy.

- **Object-oriented design** - Much of the generated code is broken up into functional objects. These functional objects are stored in libraries and can be used interchangeably, thus preventing the duplication of code and simplifying maintenance. Since each object is designed to meet specific standards, they can be easily modified to be used as the foundation for new, more specialized objects.

- **4GL language** - 4GL languages are easier to learn and use than other languages and they are portable across platforms.

- **SQL database technology** - Structured Query Language (SQL) gives you the ability to store any piece of data in your database and the flexibility to access that data in any way necessary.

- **CASE technology** - Fitrix *Screen* employs Computer Aided Software Engineering (CASE) and actually *creates* most of the 4GL code needed to run a data-entry application, saving you months of development time.

- **WYSIWYG form development** - You can create your program simply by designing the form used for data-entry. A series of menus and prompts provide you with the tools and information you need to create your application.

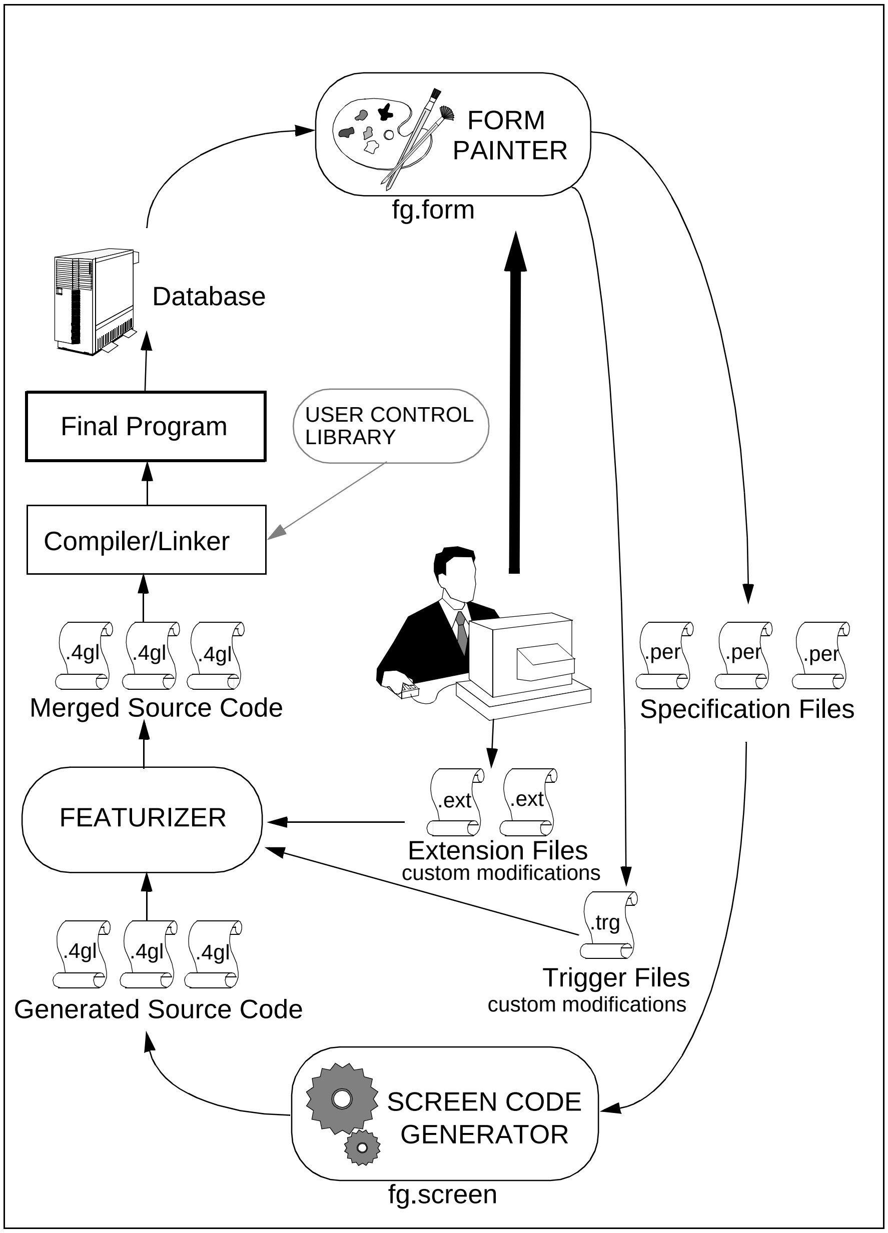
- **UNIX and open systems** - More and more companies are discovering the power and cost effectiveness of the UNIX operating system and open systems. Open systems allow you to displace work done on your expensive mainframes to a network of smaller machines without losing performance. You can say good-bye to those horrendous mainframe maintenance costs.

Overview

Fitrix *Screen* CASE Tools offer a complete solution for creating and maintaining INFORMIX-4GL applications. With Fitrix *Screen*, you can create flexible and feature-rich applications in an incredibly short amount of time. Fitrix *Screen* generated applications also benefit from a variety of useful built-in functions, such as Zoom references, file access, and the ability to lookup information from another table and return data automatically. Applications created with Fitrix *Screen* are not only powerful, but also extremely easy to maintain.

The 4gl code generated by the Fitrix *Screen* Code Generator may be immediately compiled into a functioning data-entry screen or may be modified before compilation.

The following diagram gives you an idea of the process involved when creating an application with Fitrix *Screen*.


FORM PAINTER
fg.form
Database
Final Program
USER CONTROL LIBRARY
Compiler/Linker
.4gl
.4gl
.4gl
Merged Source Code
.per
.per
.per
Specification Files
FEATURIZER
.ext
.ext
Extension Files
custom modifications
.trg
Trigger Files
custom modifications
.4gl
.4gl
.4gl
Generated Source Code
SCREEN CODE GENERATOR
fg.screen

Fitrix *Screen* is comprised of four basic components: the Form Painter, the Code Generator, the Featurizer, and the *CASE* Tools Enhancement Toolkit.

- **Form Painter** - a WYSIWYG environment used to design and create the basic data-entry interface. The Form Painter creates a .per specification file used by the Code Generator to generate the application.
- **Code Generator** - uses the specification file created by the Form Painter to create most of the code necessary to run your application.
- **Featurizer** - this tool merges your custom modifications into base generated code while maintaining regenerability.
- ***CASE* Tools Enhancement Toolkit** - consists of a library of end-user features such as Navigation, Hot Keys, and User-Definable Help Text that automatically enhance any application. The Enhancement Toolkit also consists of a number of developer tools which let you add security and a graphical menuing environment to your applications.

The Form Painter

The Form Painter simplifies the creation of data-entry screens by providing a desktop environment complete with menus and pop-up windows that greatly simplify and shorten the time required to build a data-entry form. The Form Painter serves as an easy-to-use and effective front-end to the Code Generator, further enhancing the automation of data-entry application development.

The Form Painter allows you to create form specification files, also known as .per forms. The Code Generator uses these specification files to create data-entry applications. Form specification files can be built two ways: with an editor such as vi or using the Form Painter. Creating form specification files with a standard editor is much more difficult and time consuming than using the Form Painter. Manual creation of a form specification file involves tasks such as typing in the attributes for each field on the form, carefully defining the screen record(s), and ensuring the correct format and placement of each section of the form. Once these tasks have been performed, you would attempt to compile the .per (form) file into a .frm file. Usually you would have to go through various debugging stages to get the form exactly right without any mistakes.

The Form Painter virtually eliminates mistakes by providing error checking as you create the form. The forms you create with the Form Painter are error free.

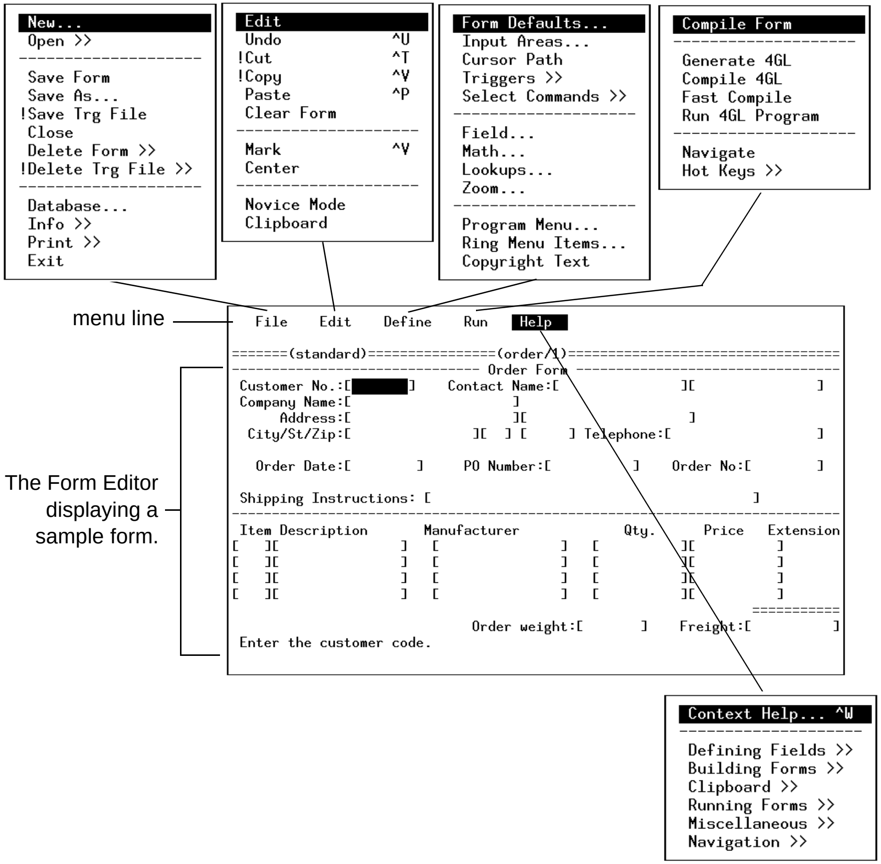
Form Painter pull-down menus provide fast and convenient access to most of the information needed to create .per form specification files.

The File pull-down menu contains options that manage your forms.

The Edit pull-down menu contains options used when creating or revising a form.

The Define pull-down menu contains options used to define the functionality of the fields and the form.

The Run pull-down menu contains various options that allow you to run the Code Generator on your form, compile, and execute the form.



```
   File   Edit   Define   Run   Help
=======(standard)===============(order/1)================================
------------------------------- Order Form --------------------------------
 Customer No.:[       ]   Contact Name:[                ][               ]
 Company Name:[                      ]
      Address:[                      ][                      ]
  City/St/Zip:[               ][  ] [     ] Telephone:[                  ]

   Order Date:[        ]    PO Number:[          ]   Order No:[         ]

 Shipping Instructions: [                                           ]
---------------------------------------------------------------------------
 Item Description      Manufacturer              Qty.      Price  Extension
[   ][               ]  [               ]  [          ][          ]
[   ][               ]  [               ]  [          ][          ]
[   ][               ]  [               ]  [          ][          ]
[   ][               ]  [               ]  [          ][          ]
                                                              ==========
                             Order weight:[       ]   Freight:[          ]
 Enter the customer code.
```

The Help pull-down menu contains various help topics which lead to reference information.

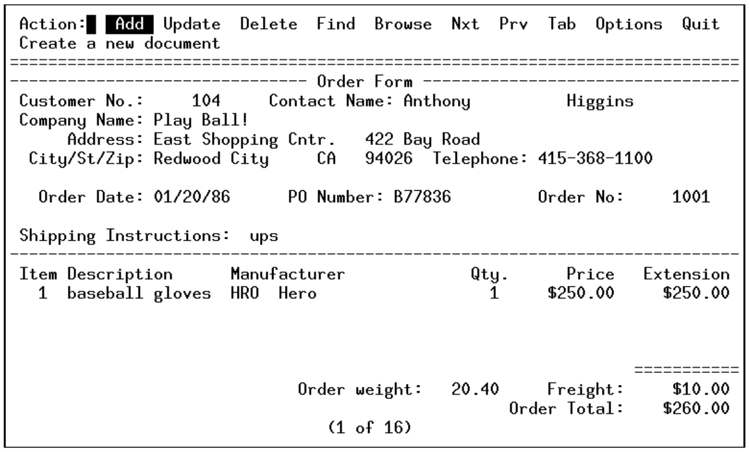
Form Painter Features

- Create error free form specification files.
- Automatically generate a form from the database.
- Specify all aspects of a form specification file.
- Create custom modifications to the generated code via triggers.
- Create, compile, and run a 4GL data-entry application in a matter of minutes.
- Create mathematical equations for fields.
- Create Zooms, which allow the user to view a list of possible entries for a field.
- Define fields that automatically return data (lookups), as well as fields that provide data validation.

The Code Generator

The Code Generator is designed to automatically write the INFORMIX-4GL program needed to produce a sophisticated and consistent data-entry environment. The primary advantage of the Code Generator is that it dramatically reduces the time needed to create 4GL code, compressing several days worth of work into a few minutes. It also produces a source code product inherently more modifiable and maintainable than traditional manually written code. The environment created allows you to Add, Update, Delete, Find, and Browse through documents. The source code follows a predefined and completely documented functional flow in which specific areas of code are designated for specific types of modification.

The following is a sample header/detail type screen created by Fitrix *Screen*.



```
 Action:█  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
 Create a new document
============================================================================
------------------------------- Order Form ---------------------------------
 Customer No.:     104     Contact Name: Anthony          Higgins
 Company Name: Play Ball!
      Address: East Shopping Cntr.   422 Bay Road
  City/St/Zip: Redwood City     CA   94026  Telephone: 415-368-1100

   Order Date: 01/20/86      PO Number: B77836         Order No:     1001

 Shipping Instructions:  ups
----------------------------------------------------------------------------
 Item Description      Manufacturer             Qty.      Price   Extension
   1  baseball gloves  HRO  Hero                  1     $250.00     $250.00



                                                                 ===========
                              Order weight:   20.40     Freight:     $10.00
                                                    Order Total:    $260.00
                                 (1 of 16)
```

Code Generator Features

- Automatically creates most of the code needed to run a data-entry application.
- Reduces application development time.
- Supports multiple languages.
- Simplifies application and program maintenance.
- Triggers let you add custom code to specific logical points in the generated code.
- Generated code is thoroughly commented.
- Add custom code or change any basic generated code through block tags while maintaining regenerability.
- Individual features of your program can be coded separately and easily included or excluded from any product version.

The Featurizer

The Featurizer merges custom code modifications into the generated base code. This allows you to keep your modifications separate from the code generated by the Code Generator for easier maintenance and also makes all programs regenerable. You can also separate your custom features and selectively plug and unplug any feature into any version of your program.

Featurizer Features

- Create 100% regenerable applications.
- Localize all custom modifications.
- Maintain multiple program versions without duplicating code.
- Create individual features that can be plugged or unplugged depending on program version.

Fitrix *Screen* Documentation

Fitrix *Screen* documentation appears in two manuals, a technical reference and a tutorial. The Fitrix Screen *Tutorial* provides an introduction to the Form Painter, the Code Generator, and the User Control Library Features.

The Fitrix Screen *Technical Reference* provides a source of information which can be consulted repeatedly. This book addresses the technical aspects relating to the specification of the data-entry screen image (.per) files, code generation based upon those specifications, RDS issues, and more. The Fitrix Screen *Technical Reference* contain sections for the Form Painter and the Code Generator.

The Fitrix CASE *Tools Enhancement Toolkit Technical Reference* provides documentation on the following features: User Control Library, C Library, Program Control Library, Pull-Down Menus, and Security.

TheFitrix CASE *Tools Training Course Workbook* also provides an excellent medium for learning how to use the major features of Fitrix *Screen*. The Fitrix CASE *Tools Training Course Workbook* contains a number of exercises that teach you how to use Fitrix *Screen* to create a custom application. The Fitrix CASE *Tools Training Course Workbook* is available separately.

The Fitrix Screen *Technical Reference* is organized by section as follows:

Part I—Introduction to Fitrix *Screen*

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

Chapter 2: Getting Started—discusses setting up your environment to run Fitrix *Screen* as well as how to run the various tools.

Chapter 3: Introduction to the Data-Entry Interface—provides an introduction to the basic interface shared by programs created with the Code Generator.

Part II—The Form Painter

Chapter 4: Form Painter Basics—explains the contents of the menus, how to use the menus, and how to move around the menu system.

Chapter 5: Managing Forms—describes form management issues such as saving, opening, printing, closing, and deleting a form.

Chapter 6: Editing Forms—describes how to create and edit forms with the Form Painter.

Chapter 7: Defining Fields—explains how to define fields in both novice and expert modes and also how to change form defaults.

Chapter 8: The Run Menu—explains how to create an application from your painted form.

Chapter 9: Database Administration—describes a program that allows you to modify database tables.

Part III—The Code Generator

Chapter 10: Creating Screen Forms—discusses typical formats used for the data-entry interface. Shows examples of .per forms and resulting data-entry screens.

Chapter 11: Source Code—examines the source code produced by Fitrix *Screen* Code Generator and discusses flow control and cursor handling.

Chapter 12: Customizing the Base Program with Triggers—discusses how source code can be modified via triggers.

Chapter 13: The Featurizer and Blocks—discusses how source code can be modified with block commands.

Chapter 14: Compiling and Running Your Programs—explains how to compile your 4GL source code and how to run your completed programs.

Chapter 15: Advanced Features—explains event handling logic, how to create help for your application, and a number of other features.

Chapter 16: Version Control—covers the concept of Version Control and how you can maintain multiple versions of a program without duplicating the base code.

Chapter 17: Language Translation—explains how to create different versions of your programs in other languages.

Chapter 18: Helpful Techniques—contains a number of common "how-to’s."

Part IV—Appendices

Appendix A: Fitrix *Screen* Utilities—contains a variety of information such as how to use the Tag utility and how to create a demonstration database.

Appendix B: The .per Specification File—looks at the components of the .per file, used to generate the application.

Appendix C: Program Migration—contains information on moving your programs from a development machine over to a production machine.

Appendix D: Fitrix *Screen* Tables—contains a list of the Code Generator tables and a list of reserved terms.

Appendix E: Control Key Defaults—contains a list of the control key defaults, as well as a list of the Form Painter editing keys.

Appendix F: Reserved Terms and Style Guide—contains a list of 4GL reserved terms and a screen form style guide.

Appendix G: Termcaps—describes how to write a termcap as well as setting terminal options.

Documentation Conventions Used in This Manual

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

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

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

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

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

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

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

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

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

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

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

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

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

2

Getting Started

This section covers information about installing Fitrix *Screen*, setting up your environment, and invoking the various products.

- Setting Up Your Environment
- Required Tables
- Backwards Compatibility Issues
- Invoking the Form Painter
- Invoking the Code Generator
- Using `fg.start`

Installation and Preparation

The following must be performed before running the Form Painter or Code Generator on your UNIX system:

Install INFORMIX-4GL version 4.1 or later according to the instructions included with the media. If you are installing the C Compiler version of INFORMIX-4GL, installation must include the C compiler/Development System and the "make" utilities included with the compiler.Install Fitrix *Screen* with `fg.install` as per instructions included with the Code Generator media.

Installing in Different (`$fg`) Directories

Normally when you install an update of Fitrix *Screen* the newer version overwrites the older version. Also, applications created with the Code Generator are typically maintained in the same `$fg` that Fitrix *Screen* is installed in. However, you can maintain your applications and the tools themselves (executables, libraries, etc.) in separate base directories (i.e., in different `$fg` directories. This ability can also be used to install and use the tools on a system without overwriting existing tools. (Note however, that due to changes in the data of some tools tables, you cannot run both sets of tools simultaneously under the OnLine engine.)

Follow the steps below to install and use the tools in a different directory:

1. Before (re)running the installation, set `$fg` to the full pathname of the target directory and (export `$fg`).

2. To run the new tools (and applications using the new libraries and etc., reset `$fg` to point to your existing applications (the old `$fg` directory), and set and export the following variables to the new `$fg` directory:

   ```
   $fgmakedir
   $fglibdir
   $fgtooldir
   ```

3. To make programs use the library *.frm’s from the new tools, put the new `$fglibdir/lib/forms` ahead of the old `$fg/lib/forms` in `$DBPATH`.

4. Add `$fgtooldir/bin` to your `$PATH` and make sure it comes before `$fg/bin`.

5. If using the INFORMIX-SE standard engine rather than the OnLine engine, set `$DBPATH` to include `$fgtooldir/data` *before* `$fg/codegen/data`.

For example, to use tools in `/usr/fourgen2` while the applications you create and run are stored in a different directory, you could put the lines below into your .profile file (or a script to optionally execute—don't forget to use the . to make the settings apply to the current shell):

```
fgtooldir=/usr/fourgen2 ; export fgtooldir
fgmakedir=$fgtooldir ; export fgmakedir
fglibdir=$fgtooldir ; export fglibdir
DBPATH=$fglibdir/lib/forms:$DBPATH ; export DBPATH
```

`$fg`: base directory for tools and applications.

`$fgmakedir`: if set, `fg.make` looks for make files in this directory rather than `$fg` (even though local Makefiles still contain `$fg`).

`$fglibdir`: if set, `fg.make` looks for upper level libraries in this directory rather than `$fg`.

`$fgtooldir`: if set, tools executables, (e.g. the 4gl program executed by calls to the screen generator such as `fg.screen`), are searched for in this directory rather than `$fg`.

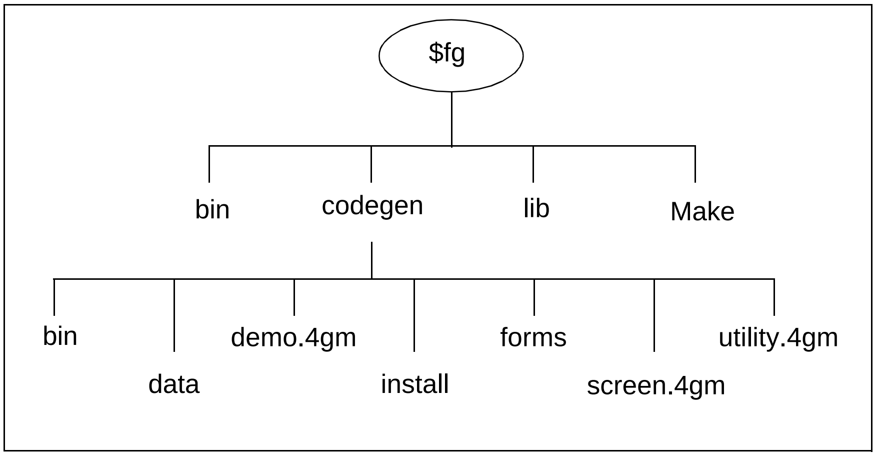
`$DBPATH`: Path to *.frm files. If new `$fg` doesn't precede the old `$fg` in `$DBPATH`, when you run your program it will use the old library's forms. (This is not really crucial on systems where 4.10 is installed in the old `$fg`, since these forms have not changed in the 4.11 upgrade.)

Installation Directory Structure

The following diagrams illustrate the basic directory structure that is created when you install Fitrix *Screen*. Installation of the Code Generator program produces the directory structure outlined in the following pages. The diagrams provide an overview of the location of Code Generator files upon installation. The ellipse indicates that a variable is used to represent part of the path.

Overview of the Directory Structure

The following diagram represents the basic directory structure of Fitrix *Screen*.



These directories are the basis of the Fitrix *Screen* installation directory structure. Each directory is explained in the following pages.

`bin`: This directory contains Fitrix *Screen* executables.

`data`: This directory contains database files used by Fitrix *Screen*.

`demo.4gm`: This directory contains demonstration programs.

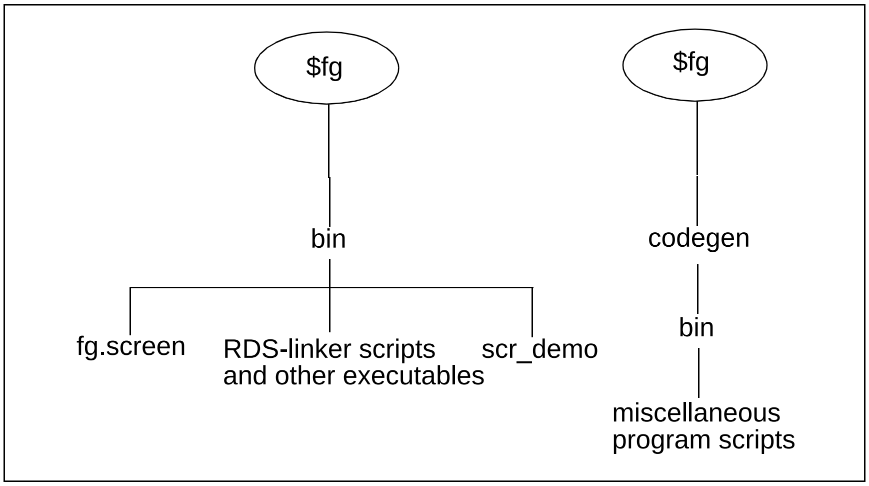
`install`: This directory contains installation files.

`forms`: This directory contains form specification files for forms used by the Form Painter.

`screen.4gm`: This directory contains the executables for Fitrix *Screen* programs.

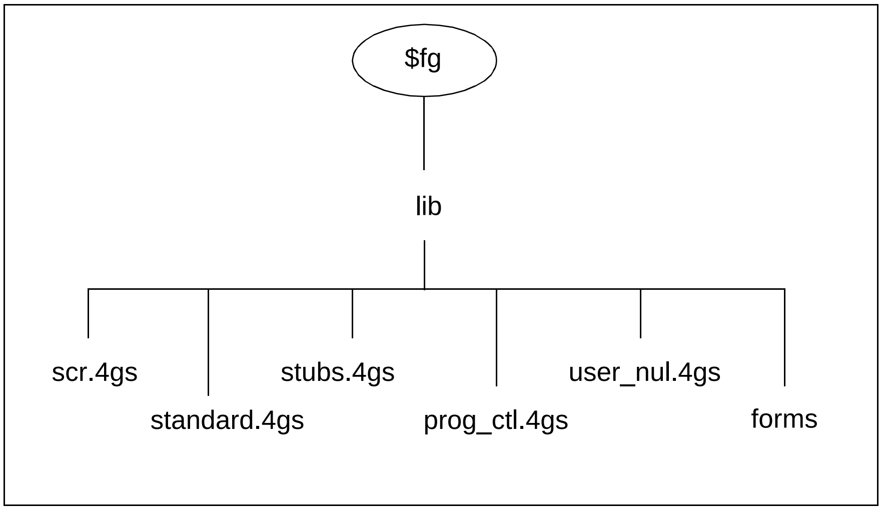
`utility.4gm`: This directory contains source code for the language translation programs.

Executable Files


$fg
bin
fg.screen
RDS-linker scripts and other executables
scr_demo
$fg
codegen
bin
miscellaneous program scripts

The Code Generator is invoked with the `fg.screen` shell script. The Code Generator demo is invoked with the `scr_demo` executable. Other shell scripts perform functions related to the INFORMIX-4GL Rapid Development System (RDS), hypertext tags, and other Code Generator-related matters. The `../codegen/bin` directory contains shell scripts for creating tables required by the Code Generator and for running generated applications.

Library Source Files



The `*.4gs` and `forms` directories contain code used by the Code Generator to build applications.

`scr.4gs`: These library files are used by programs generated with the Code Generator.

`standard.4gs`: These library files are used by both Fitrix *Screen* and the Fitrix *Report* Code Generator.

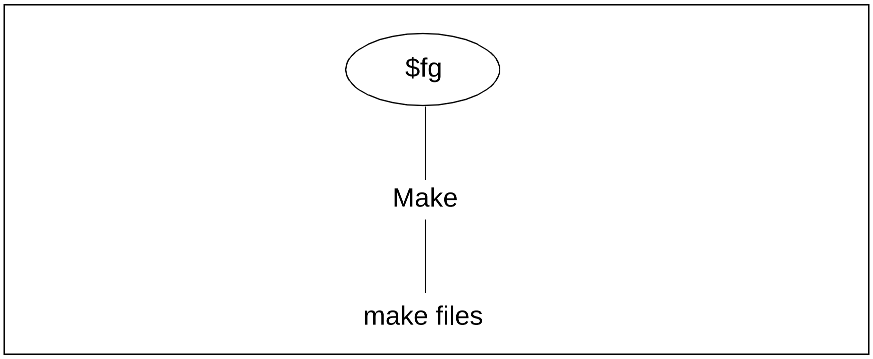
`user_ctl.4gs`: These library files contain additional features which can be used from the Form Painter. These functions are also used by your applications if this library exists on the run-time system.

`prog_ctl.4gs`: This library contains a number of advanced functions which you can incorporate into your own applications.

`stubs.4gs`: This library contains stub functions for the `stubs.4gs` library. This enables your applications to run if the Enhancement Toolkit has not been purchased for the run-time system.

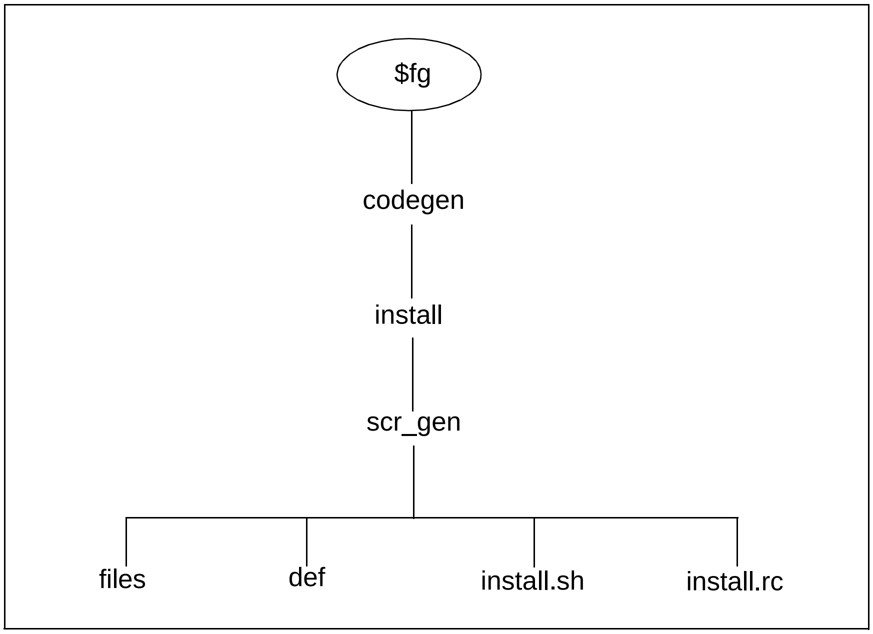
`forms:` This directory contains form specification files used by functions available in code generated with the Code Generator.

Makefile Files



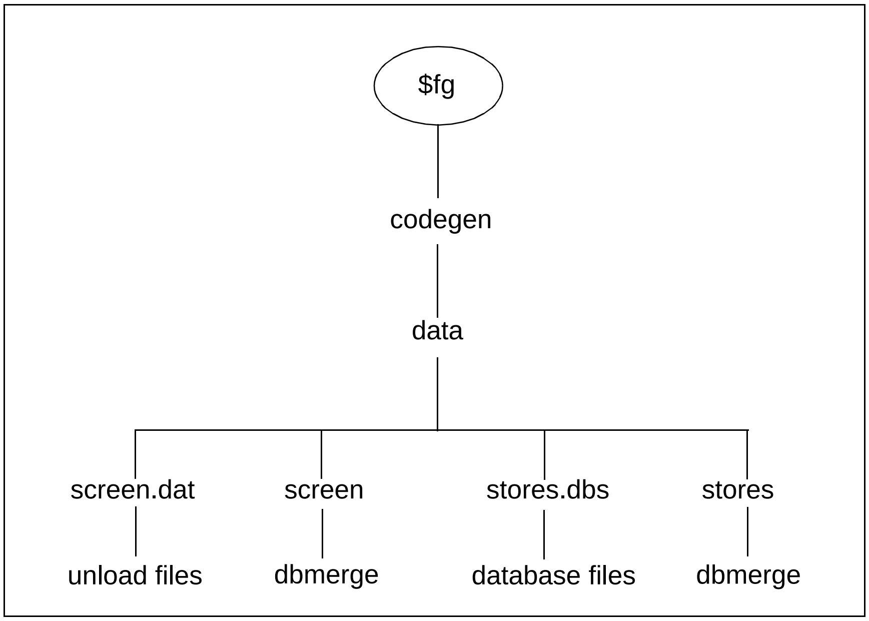
The Makefile files coordinate the compilation of source files into executable program files within a 4GL application. The $fg/Make directory contains the real make files, which use the information found in directories containing Makefile files. Program compilation is discussed in "Compiling Generated Code" on page 14-2.

Install Files



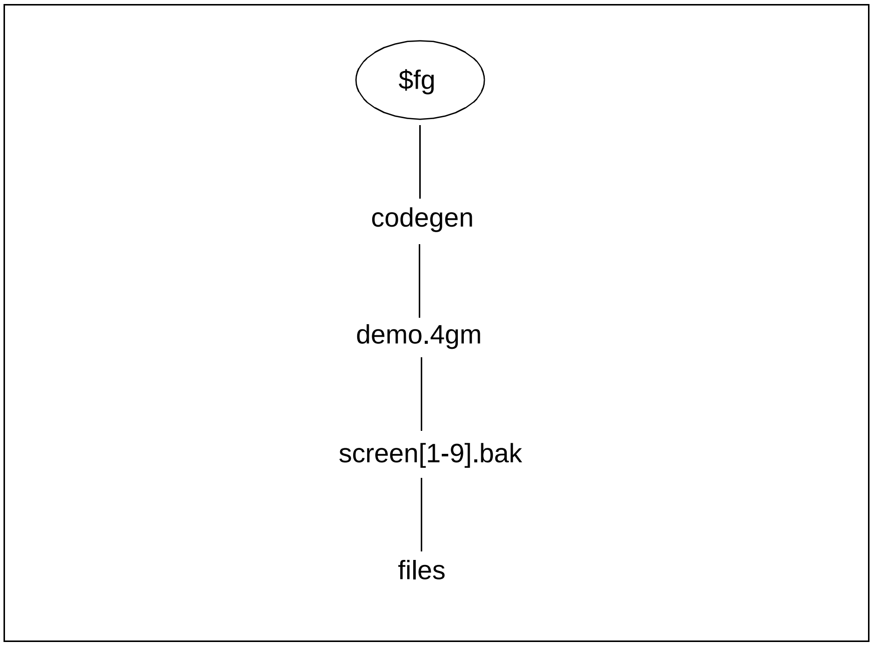
The `files` file provides an installation "blueprint," indicating where files are installed on the system relative to `$fg`. It lets the create script know what file names to pass on to tar and changes ownerships, groups, and permissions. The `def` file contains product information. The `install.sh` file is the actual installation script for the Code Generator program, and uses the setting provided by `install.rc`.

Database Files



The screen`.dat` files provide information (in the form of table unload files) used to run applications off of the `stores` sample database. The `default.dat` directory also contains unload data used by features found in code generated by the Code Generator. The `screen` directory contains a `dbmerge` program which is used to build the feature-required data into a database. The `stores` directory contains a `dbmerge` program which is used to create the stores sample database, which is found in `stores.dbs`.

Demo Files



A number of demonstration directories are installed with the Code Generator. The demo directories contain a variety of different .per forms which you may use to generate sample applications. Each screen directory is installed with a companion `.bak` directory, to ensure that the original .per demo files are not altered. The contents of the `.bak` directory should be copied over to the corresponding `screen` directory to generate and run the demo. The `scr_demo` script copies the files from the .bak directory to a corresponding .4gs directory.

The `screen1` directory contains a simple header-only application.

The `screen2` application is header/detail, and appears with a browse but no Zooms.

The `screen3` application contains a full-featured header-detail application, with a browse and Zooms.

The `screen4` directory is the same as `screen3` but is used for the Form Painter demo.

The `screen5` directory is a complex header/detail application that demonstrates triggers.

The `screen6` directories demonstrate the Featurizer. For information on this demo refer to "The Featurizer and Blocks" on page 13-1.

The `screen7` demo provides a sample program that utilizes extension screens.

The `screen8` directory contains files which allow you to build a sample add-on detail program.

The `screen9` directory contains view-header, view-detail, and query screen type examples.

Setting Up Your Environment

The next step is to ensure that system variables used by the Form Painter and Code Generator are set correctly. The following variables must be set prior to invoking the Form Painter or Code Generator:

`$fg` must point to the installation directory of Fitrix *Screen*.

`$fgtooldir` optional variable that can be set to point to an alternate parent directory containing a `codegen` source directory rather than the one in `$fg`.

`$fgmakedir` optional variable that can be set to point to an alternate parent directory containing a `Make` utilities directory rather than the one in `$fg`.

`$fglibdir` optional variable that can be set to point to an alternate parent directory containing a `lib` upper-level libraries directory used by the generated code rather than the one in `$fg`.

`$INFORMIXDIR` must point to the Informix directory on the system. Typically, this is `/usr/informix`.

`$PATH` must include `$fg/bin` and `$INFORMIXDIR/bin`. If `$fgtooldir` is set, you must include `$fgtooldir/bin` before `$fg/bin`.

`$DBPATH` must include `$fg/lib/forms` and `$fg/codegen/forms`. If `$fglibdir` is set you must include `$fglibdir/lib/forms` and `$fglibdir/codegen/forms` in your $DBPATH.

`$TMPDIR` is a system variable that specifies the directory that temporary files are placed into. If you run into problems where your TMPDIR space is not large enough you may need to specify a new temporary directory location.

`$DBTEMP` is the directory into which INFORMIX-4GL places its temporary files. The default is the `/tmp` directory. If you are using the standard engine you may need to specify a new `/tmp` directory that contains sufficient space.

Note

The $fgtooldir, $fglibdir, and $fgmakedir variables are optional variables that you can set to point to alternate parent directories for certain components of Fitrix *Screen*. These variables allow you to have more than one version of Fitrix *Screen* on your system, and give you the ability to use a particular component from a different version. For example, you could have two versions of Fitrix *Screen* on your system, version A and version B. By setting $fglibdir to point to the directory containing the libraries in version B, you could then run version A of the Code Generator but use the libraries from version B. Refer to "Installing in Different ($fg) Directories" on page 2-2.

Tables Required to Run the Code Generator

Special Code Generator tables need to be present in any database the Code Generator is run against. All of the cg* and stx* tables need to be present in the development database in order to generate or compile programs.

When the Code Generator is installed, these tables are automatically added to your databases. If you run the Code Generator against a database that does not have these tables (i.e., the database was created after the installation of the Code Generator), they are added automatically. Whenever the Code Generator adds tables to a database, the indexes for that database get removed and recreated.

If you want to manually add the Code Generator tables to a database you can run a script called mktables. This script adds the necessary Code Generator tables to the database. For more information on the tables that get installed with Fitrix *Screen* refer to "Code Generator Tables" on page D-5.

Tables Required to Run a Generated Program

When running a program generated by the Code Generator, the stx* tables need to be present in any database the generated program is run against. The stx* tables contain program help and error messages, as well as the User Control Library tables. You can run `mktables` to add the stx* tables to your database. The `mktables` script also adds the cg* tables to a database. Since you may not need the cg* tables on your production database, you can remove these tables to save space. For more information on transferring an application to another system, refer to "Moving Applications to Other Systems" on page C-2.

Standard Database Must Exist in `$DBPATH`

Even though you might be using a development database other than `standard`, a database named `standard` *must* exist on your system. This is because Fitrix *Screen* was compiled against a `standard` database. When run, Fitrix *Screen* tries to open a `standard` database.   If a `standard` database can't be found, an error occurs. You can remove everything from the `standard` database if you wish to use a different development database. You just need to make sure a database named `standard` exists, even if it is empty.

Recommended Application Directory Structure

Since applications tend to grow and expand over time, we recommend the following directory structure, which allows applications to grow in an organized fashion. These directory structure conventions provide a consistency which make applications easier to support and maintain, particularly for those users other than the author.

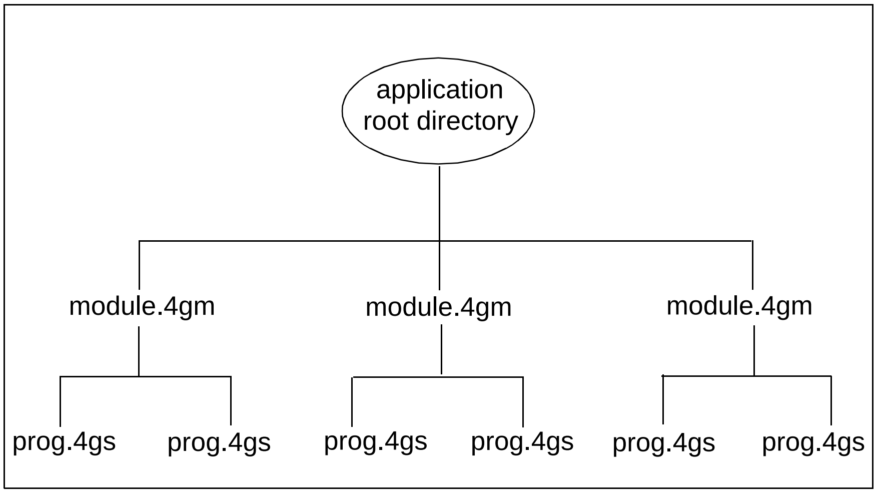
Neither the Form Painter or the Code Generator forces compliance with directory structure conventions. However, Fitrix Fitrix *Screen* does perform best under the organized environment created by the conventions explained in this section.

First, the directory hierarchy is explained, then directory naming conventions are explained.

Directory Hierarchy

Each application should be broken down into a three tier hierarchy. At the first level you have the directory that contains the entire application. This is called the "root directory." Down from the application directory you have the module directories. Module directories contain groups of related programs. The program directories contain most of the code needed to run that particular program.

The following diagram illustrates the directory structure conventions for applications generated with Fitrix *Screen*.



Directory and File Naming Conventions

Application root directory—a descriptive name such as "accounting." The application root directory name should be less than 12 characters.

It may also be helpful to create a UNIX environmental variable to represent the topmost directory in your application hierarchy. This variable generally would represent the installation directory, for example accounting.

Module directory—module directory names must follow the following format: `*module_name*.4gm`. The .4gm extension must be used to denote the module directory level. *`module_name`* must be 8 characters or less.

Under the "application" directory, there are several different types of directories. First are the "module" directories. A module is a free-standing part of the overall application.   A module directory is named for the module and the format is *`module_name`*`.4gm`. For example, `ac_mnt.4gm` or `heat_mnt.4gm` in the case of plant maintenance.

Program directory—under a module directory there are the individual source directories which not only contain the various .4gl programs but also the actual executable itself. These source directories are named with respect to their functions. One convention used is as follows:

`i_` - input screen

`o_` - output report

`p_` - posting report

These are immediately followed by a meaningful name, followed by ".4gs." For example, `i_time.4gs` could mean "an input screen for time cards source directory."

Program names—program names take the first part of their name from the program directory, but they have either a .4ge or a .4gi extension depending on how they are compiled.

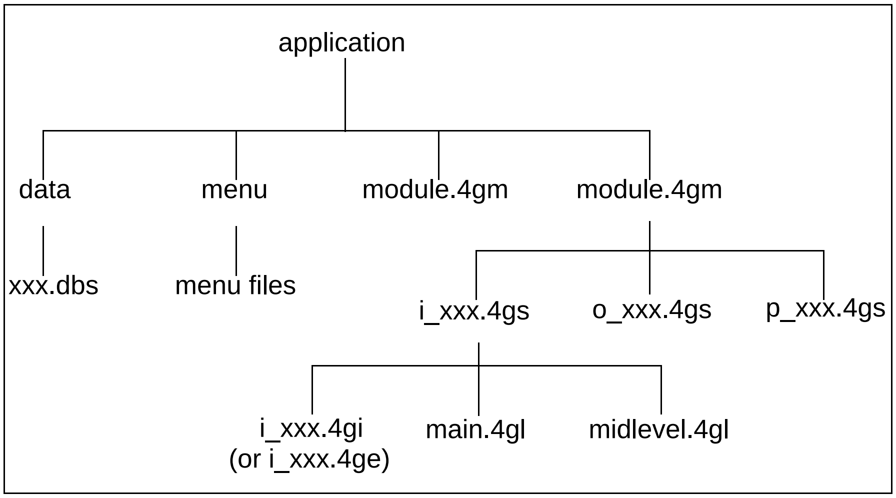
Note

Directory names must be no larger than 8 characters, not including the extension.

Next, under the application directory, there is the directory *data*. This directory contains the database directory or directories (if multiple databases are used). The `$DBPATH` environment variable should be set to point at: *`application_name`*`/data` when using the Standard Informix engine. This directory contains the `.dbs` directories used by Informix. The `$INFORMIXDIR` variable must also be set.

The final recommended directory within an application hierarchy should be the *menu* directory if Fitrix *Menus* is being used. This is the directory in which all the user menus are stored.

Application Hierarchy:



Maintaining Backwards Compatibility—The Options Files

This applies only to users that have generated code with versions of the Code Generator previous to the May 1992 release. The Code Generator now generates `q_` records differently than it has in the past, and certain triggers are handled differently. A `q_` record is a data variable that parallels data elements defined in the screen. If you are a new user of the Code Generator then this section doesn't apply to you.

Two special files have been created allowing you to control how the Code Generator and Featurizer handles some special circumstances. The `fglpp.opt` file contains variables that allow you to specify how the Featurizer handles some of the special triggers, while the `screen.opt` file contains a variable which allows you to control how the Code Generator handles `q_records`.

The Featurizer Options File (`fglpp.opt`)

This file contains options that control how the Featurizer handles special triggers and how it handles missing blocks.

Trigger Controls

In order to maintain backward compatibility, a special options file has been added to the Featurizer which allows you to control how the Featurizer handles the following special triggers:

```
define
static_define
at_eof
switchbox_items
```

The predecessor to the Featurizer, the Trigger Merge Utility, replaced the subsequent occurrences of the above, mentioned triggers. In other words, if you had the directory search path 4gc:4gs, and you had a `define` trigger in both directories, the `define` trigger in the .4gs directory would be merged in first, and then the `define` trigger in the .4gc directory would replace the existing `define` trigger.

The Featurizer now appends these triggers by default to prevent you from having duplicate triggers in your directory. This sounds like a contradiction in terms, but really it isn't. If two identical `static_define` triggers are found when the `define_trig` variable is set to "append," the merge will fail, giving you a duplication error. This allows you to go back and remove one of the duplicate triggers, assuring that each of these special triggers in your application trigger files are unique.

The Featurizer allows you to choose whether you want subsequent occurrences of these special triggers to replace or append existing triggers.

The Code Generator creates a `$fg/codegen/options/fglpp.opt` file with the following contents:

```
define_trig="append";   export define_trig
at_eof_trig="append";   export at_eof_trig
swbox_trig="append";    export swbox_trig
```

These variables allow you to specify "append" if you want the Featurizer to append the associated triggers, or "replace" if you want the Featurizer to replace existing triggers. The default is "append."

fglpp.opt File Variables	Triggers Affected
`define_trig`	`define` `static_define`
`at_eof_trig`	`at_eof`
`swbox_trig`	`switchbox_items`

An `fglpp.back` file has been included in the `$fg/codegen/options` directory. This file contains settings which cause the Featurizer to work like the Trigger Merge Utility. To make the Featurizer replace the special triggers (`static_define`, `define`, `at_eof`, and `switchbox_items`) instead of appending them, copy the backward compatibility option file to `fglpp.opt`.

```
cd $fg/codegen/options
cp fglpp.back fglpp.opt
```

To do this for just one program:

```
cd /path/where/my/program/source/code/is.4gs
cp $fg/codegen/options/fglpp.back fglpp.opt
```

To do this for a module:

```
cd /path/where/my/module/is.4gm
cp $fg/codegen/options/fglpp.back fglpp.opt
```

Missing Blocks

The Featurizer was recently changed to no longer consider missing blocks as fatal errors. This was done to correctly support version control requirements with fglpp. This behavior can be changed by setting the system variable `$fglpp_fatal_warn` to "Y." You can do this using the fglpp options file (`$fg/codegen/options/fglpp.opt` or a local `fglpp.opt`).

The Code Generator Options File (`screen.opt`)

This file contains options that allow you to control how the Code Generator generates certain attributes.

Generating q_ Records

Another options file, `$fg/codegen/options/screen.opt`, contains a special option for the Code Generator. The `screen.opt` file contains the `non_scr_q_elems` option. This option allows you to maintain backwards compatibility with older versions of the Code Generator. This option determines what should be included in the header file's `q_` record, which is generated in the file `globals.4gl`. It also controls what `q_` record elements are assigned values in the `llh_m_prep()` and `llh_p_prep()` functions.

The old way of creating the header `q_` record is as follows:

If you use a field that is not displayed on the screen, you need to create a define trigger to add the field to the q_ record. You also need to use a trigger to add the q_ record to the m_prep() and p_prep() functions. The Code Generator automatically generates the row_id.

You can duplicate the old behavior of the Code Generator by setting the non_scr_q_elems variable to "exclude."

The new way of creating the header q_ record:

If the non_scr_q_elems variable is set to "include," (this is the default), then a q_ record is generated for every column in the table not displayed on the screen. The new way also references these additional q_ record variables into assignment statements within the llh_p_prep() and llh_m_prep() functions.

Here is the default environmental variable settings:

```
non_scr_q_elems="include";   export non_scr_q_elems
```

Note

If you are using extension screens, the non_scr_q_elems variable must be set to "include."

To make Fitrix *Screen* generate code the old way, copy the backward compatibility option file, which resides in $fg/codegen/options/screen.back, to screen.opt.

```
cd $fg/codegen/options
cp screen.back screen.opt
```

To do this for just one program:

```
cd /path/where/my/program/source/code/is.4gs
cp $fg/codegen/options/screen.back screen.opt
```

To do this for a module:

```
cd /path/where/my/module/is.4gm
cp $fg/codegen/options/screen.back screen.opt
```

Controlling the Detail Display Function

This is a backward compatibility option for the detail display function. It can be set to "current_context" or "first_page." If set to "first_page" this will direct the Code Generator to generate the old style `lld_showdata()` function (or equivalent) in a manner to simulate past behavior which always displayed just the first page of detail lines.

The `lld_showdata()` function was redesigned so that it can be called at any time to display the current set of detail lines instead of the first page. This capability requires that the variables p_cur and s_cur are set correctly. Usually this will be the case. However, if p_cur and/or s_cur are used in ways that change their values to not reflect the current detail line context then the new function will behave incorrectly.

By default this function is set to "current_context." To change this to simulate "first_page" behavior you may set `$detl_display="first_page."`

Here are the current option settings as they appear in `$fg/codegen/options/screen.org`:

```
detl_display="current_context";   export detl_display
```

Modifying the Options Files Locally

If you want to change to the settings in the `fglpp.opt` or `screen.opt` files but you do not want to change them for every user on your system, you can create these files in your application or your program directories. For example, say you want to run the Featurizer so that it "replaces," but other developers want to use the default "append." You can't change the system default in `$fg/codegen/options/fglpp.opt.` All you need to do is create an `fglpp.opt` file in your module or program directory and include the settings you want. For example, you might have the following:

```
define_trig="replace";   export define_trig
at_eof_trig="replace";   export at_eof_trig
swbox_trig="replace";    export swbox_trig
```

You could also put an `fglpp.opt` or `screen.opt` file in your program directory.

When the Code Generator and the Featurizer are run, they run the option files in the `$fg/codegen/options` directory first, so those variables get set. Then any .opt file in the application directory is run. Finally, the .opt files in the program directory are run. Each time a variable in these files is exported, it replaces the current setting. In other words, if you had `non_scr_q_elems` set to "include" in the default `screen.opt` file, and it is set to "exclude" in your application directory, then it is set to "include" in the program directory, when you run the Code Generator from the program directory the `non_scr_q_elems` variable is "include."

Running the Form Painter

The Form Painter must be invoked from the directory you wish to develop your program in. The *CASE* Tools are designed to work best in a particular directory structure. This structure allows you to take full advantage of Version Control and prevents you from having to duplicate code. Refer to "Recommended Application Directory Structure" on page 2-15 for a detailed explanation of directory structure and file naming conventions.

Once you are in the proper directory, you are ready to develop .per form specification files and corresponding applications. The following command invokes the Form Painter:

```
fg.form [-dbname database] [-o (0-5)] [-f] [-yes]
[-p perform]
```

-dbname The database can be pre-determined by using the `-dbname` flag and specifying the name of a database.

-o (0-5) The `-o` flag followed by a value (0-5) determines the amount of code that gets displayed during code generation, which subsequently affects the speed of generation. The less output that is directed to the screen, the faster the generation. Invoking the Form Painter with "`fg.form -o 0`" is the fastest form of code generation.

This flag is passed to the Code Generator when the Code Generator is invoked from the Form Painter. The default generation level is 4.

-f The `-f` (fast) flag is similar to the (`-o 1`) output level of Code Generation. This flag is passed to the Code Generator when invoked through the Generate 4GL option of the Form Painter.

-yes The `-yes` flag automatically answers all Code Generator prompts with a yes. This flag is passed to the Code Generator when invoked through the Generate 4GL option of the Form Painter.

-p The -p flag allows you start up the Form Painter and automatically load the specified .per file.

Invoking the Code Generator

There are three ways to invoke the Code Generator: through the Form Painter, manually by typing `fg.screen`, and automatically with a startup program. To invoke the Code Generator manually, a number of conditions must be met prior to invocation. The startup program, `fg.start`, automatically sets up many of these conditions for you. First, the manual method is discussed.

To invoke the Code Generator for any application, you must first change directories to the program directory that contains the .per forms you intend to use to generate code. You must be in the correct directory when invoking the Code Generator.

If you are using Version Control, which allows you to maintain different versions of your applications, you must also run the Code Generator from the appropriate directory. However, with Version Control, all .per forms do not have to be located in the directory in which the Code Generator is run. For more information see "Invoking Programs That Use Version Control" on page 16-20.

File names are created relating to the current directory name. The .4ge and .4gi files are given the name of the directory they were created in.

Once you are in the proper directory, and the variables mentioned previously contain the required values, you are ready to generate the application. The following command invokes the Code Generator on the specified .per file(s):

```
fg.screen [-dbname database] [-o {0-5}] [-f] [-yes|no]
[perform file...]
```

-dbname Specifies the database on which the source code will operate. There is no need to use this flag if the `$DBNAME` environmental variable is properly set.

-o {0-5} Specifies the generation level. The generation level controls the screen display of generated code as it is being created. Level 4 is the default. Level 5 is the slowest—it is artificially slowed for demo purposes. Level 0 displays minimal information to the screen and produces the fastest generation level. The output level can be changed during code generation by pressing [DEL].

-f Specifies a "fast" generation level. This flag works the same as if you specify -o 1.

-yes|no Specifies interactive or non-interactive generation modes. You can also use just `-y`. During normal code generation, different prompts may appear requiring user interaction. Such is the case if you have modified a .4gl. Upon regeneration, a prompt appears requiring entry from a list of actions to take. If you specify the `-yes` flag when invoking the Code Generator, the codegen works silently without prompting for user input. All user prompts are suppressed, and all are automatically answered as if you had typed a Y. This allows for automated batch regeneration.

perform file Specifies .per forms to generate code for.

If you are using Version Control do not specify the .per file names. The Code Generator automatically determines what .per files to use. Refer to "Invoking Programs That Use Version Control" on page 16-20.

The entire generation process takes a few minutes, depending on the number of .per files specified.

Note

If any problems occur during start-up, check to make sure your `$DBPATH` includes `$fg/lib/forms` and `$fg/codegen/forms`.

Using the `fg.start` Startup Script

The second means of executing the Form Painter or the Code Generator is with a special startup program `fg.start`. The `fg.start` program simplify helps you set up your environment before running the Code Generator or the Form Painter.

The `fg.start` program is installed in `$fg/bin`. It does not require any system variables other than `$TERM`. However, `$fg` *must* be set to your Code Generator installation directory. If `$INFORMIXDIR` is other than `/usr/informix` or `/u/informix` then `$INFORMIXDIR` must be set to your INFORMIX installation directory.

Note

`$fg/bin` must be in your `$PATH`.

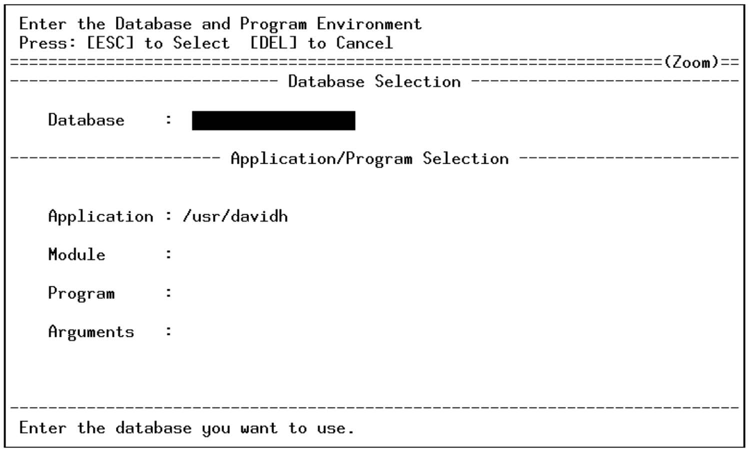
`fg.start` can be run by typing `fg.start`. It accepts two optional arguments, "screen" to invoke the Code Generator instead of the Form Painter and "-dbname database" to pre-select the database. You may also invoke it with "`fg.start help`" to get a usage message. `fg.start` syntax:

`fg.start [-dbname` ***`database`*** **`] [screen]`**

Typing `fg.start` with no arguments defaults to Form Painter start-up. If you specify "screen" then `fg.start` runs the Code Generator.

The program displays a full screen entry form where you can select the database, application directory, module directory name, program directory name, and all of the command line arguments mentioned above in the "manual" invocation discussion (like `*.per` or `-o 2`).

The `fg.start` data-entry form:



```
 Enter the Database and Program Environment
 Press: [ESC] to Select  [DEL] to Cancel
===================================================================(Zoom)==
--------------------------- Database Selection ---------------------------

   Database    :

--------------------- Application/Program Selection ----------------------


   Application : /usr/davidh

   Module      :

   Program     :

   Arguments   :



---------------------------------------------------------------------------
 Enter the database you want to use.
```

Zoom forms are available in the Database, Module, and Program fields. The database Zoom allows you to call up a list of available databases. The database Zoom is based on `$DBPATH` so is not OnLine sensitive. OnLine databases are not displayed. All *.dbs in your `$DBPATH` are listed. The Zooms for module and program provide a list of *.4gms in the application directory and *.4g[sc] in the module directory.

You are prompted to create any databases that do not exist and any directories that do not exist.

Once selection is complete and [ESC] is pressed the program changes directories to the selected program directory and runs either `fg.form` or `fg.screen` on the selected database with the specified command line options. `$fg` and `$INFORMIXDIR` are set by the program if not already set. `$fg/bin` and `$INFORMIXDIR/bin` are appended to `$PATH`. The directory from which `fg.start` was invoked is added to `$DBPATH` along with `$fg/lib/forms`. Once you have run `fg.form` or `fg.screen` and quit, you are returned to the `fg.start` screen to select another program. You may exit this form by pressing [DEL].

"Compiling Generated Code" on page 14-2 contains a description of source code compilation and execution of compiled files. Information under that topic includes program invocation flags, as well as methods for specifying filters, order by clauses, and database names.

Regeneration of Source Code

One of the strengths of Fitrix *Screen* is that it allows you to create regenerable programs. This means that you can regenerate your programs with a newer version of the Code Generator without losing your original modifications.

If you re-run the Code Generator in a directory which already contains generated code, the Code Generator does not assume that you wish to overwrite each source code file in the directory. The Code Generator displays a prompt to determine how you wish to deal with old (existing) source code files. The prompt appears for each source code file that could be overwritten.

As an example of how the system handles duplicate files, consider a situation in which a `Makefile` already exists. The system shows:

```
There currently exists a file called: Makefile
Would you like 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 Makefile.diff, type
a "d" after the selection. example  2d)

Enter Selection:
```

A similar menu appears for the INFORMIX-4GL source code files if they already exist.

- Option 1 causes the old version of the file to be replaced with the new version.
- Option 2 appends the new version to the end of the existing file.
- Option 3 moves the existing file to one with the suffix `.old` appended to the name, then writes the new one.
- Option 4 leaves the existing file as it is and writes the new code to a file with `.new` appended to the name.
- Option 5 skips the creation of the file and goes to the next file.

- Option 6 exits the Fitrix *Screen* Code Generator process without creating any more files.

The option you select from the Duplicate File menu depends upon the modifications, if any, that you have made to the existing file; the modifications resulting from changes to the screen form specification files; and the relative difficulty of replicating or merging the code of the different files. The decision obviously requires some familiarity with the files and code generated by the Code Generator, as does the modification of the files in the first place. If you have not changed the code created by the previous run of the Code Generator, you should select option 1 or 3.

3

The Data-Entry Interface

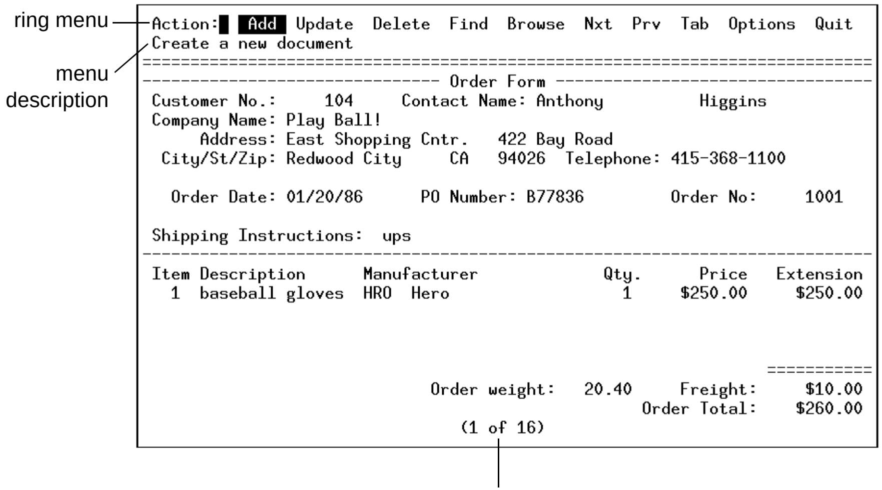
This part of the documentation provides information on the data-entry interface used by programs created with Fitrix *Screen*. This section covers:

- The standard program interface
- Ring menu commands
- Introduction to the Zoom feature
- Introduction to Lookups
- Program Information menu commands
- Default screen attributes

The Basic Fitrix *Screen* Generated Interface

Fitrix *Screen* creates a consistent data-entry interface. A consistent interface not only makes it easy for your end-users to learn and use your programs, but a consistent interface also makes it easier to create your programs and to maintain them.

Example data-entry program created with Fitrix *Screen*:



```
Action:█ Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Create a new document
============================================================================
------------------------------- Order Form ---------------------------------
Customer No.:     104     Contact Name: Anthony          Higgins
Company Name: Play Ball!
     Address: East Shopping Cntr.   422 Bay Road
 City/St/Zip: Redwood City     CA   94026  Telephone: 415-368-1100

  Order Date: 01/20/86      PO Number: B77836        Order No:     1001

Shipping Instructions:  ups
----------------------------------------------------------------------------
Item Description      Manufacturer             Qty.     Price   Extension
  1  baseball gloves  HRO  Hero                  1     $250.00     $250.00



                                                                 ===========
                             Order weight:   20.40     Freight:      $10.00
                                                   Order Total:     $260.00
                                 (1 of 16)
```

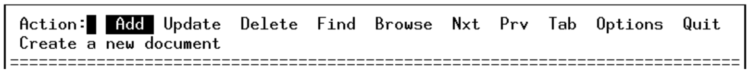
Current document is the first of a selected group of sixteen

The Data-Entry Ring Menu

The standard data-entry ring menu can be found on most programs created with Fitrix *Screen*. If you have the *CASE* Tools Enhancement Toolkit you can specify an optional ring-menu which consists of pull-down menus. These Pull-Down Menus offer greater flexibility as well as more commands. For more information about the Pull-Down Menus refer to the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

The standard ring menu consists of ten default commands. These menu commands give you the ability to perform a variety of operations on your records (documents).

The standard data-entry ring menu:



```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Create a new document
============================================================================
```

Add: Adds a new record.

Update: Modifies an existing record.

Delete: Deletes a record.

Find: Queries for a single record or group of records.

Browse: Views a summary of all selected records.

Nxt: Pages to the next record in the current group.

Prv: Pages to the previous record in the current group.

Tab: Views detail lines in current record.

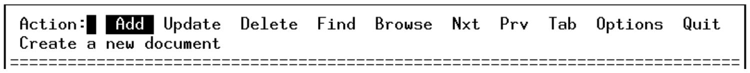
Options: Contains custom menu options.

Quit: Quits the program.

Commands are executed from the ring menu by moving the highlight over the command and pressing [ENTER], or by typing the first letter of the command name. Case is not significant in running the commands; both lowercase and uppercase characters work. There are additional keys for executing some commands. For example, the Quit command may be run with X or E in addition to Q. These extra keys are noted in the descriptions that follow. In addition, operating system commands may be executed from the ring menu command prompt by preceding the UNIX command with an ! (exclamation mark).

The items of the Fitrix *Screen* data-entry ring menu may be modified to work differently in an application. Usually these customizations to the ring menu check conditions of the data before allowing use of the menu item. For example, a screen for entering customer data may not allow the user to use the Delete option if there are orders for that customer.

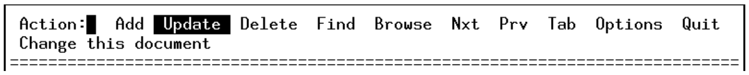
The Add Command



```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Create a new document
============================================================================
```

The Add command on the data-entry ring menu is used to add new rows to the table. When you select Add by pressing an A or [ENTER] while the command is highlighted, the system loads an empty row and takes you to the first column on the screen. When you have filled one column you may move to the next or previous column with the [ENTER] and arrow keys. When you have completed column data-entry, press the [ESC] key to store the data. If you press the [DELETE] key before pressing [ESC] the entry is aborted and the new record is not saved.

The Update Command

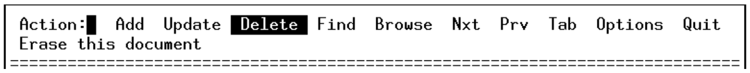


```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Change this document
============================================================================
```

The Update command is used to modify data in existing rows. Once you have located the row you wish to modify with the Find, Next, and Prev options, you may change the data by typing U or pressing [ENTER] while Update is highlighted on the ring menu. The Update command moves the data-entry cursor to the first column on the screen. You may enter new column data by typing over the current contents and can move the cursor to the next or previous column with the [ENTER] and arrow keys just as in the Add command.

When you have completed updating the row press the [ESC] key to store the data and return to the data-entry ring menu. If you press the [DELETE] key before [ESC] the data reverts to the form it had before the Update command was run. This is convenient when you accidentally change the contents of a column you did not wish to change and do not wish to retype the original data. Pressing the [DELETE] key returns the data to what it was when the last [ESC] was pressed.

The Delete Command



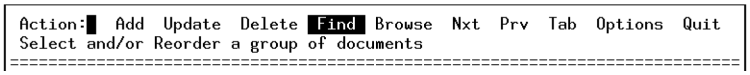
```
Action:█  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Erase this document
==========================================================================
```

The Delete command is used to remove rows from a table. It may be executed with the highlight or by typing a D. When you select the command from the ring menu the system prompts:

```
Delete:  Verify document deletion
Erase this document? (Y/N)
```

Answering the prompt with an n or N aborts the delete and returns to the ring menu without removing that row. Answering with a y or Y removes the row and returns to the ring menu above the now empty screen.

The Find Command



```
Action:█  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Select and/or Reorder a group of documents
==========================================================================
```

The Find command lets you locate rows by searching for certain patterns or ranges of data. When you select Find by moving the highlight over the command and pressing [ENTER] or by typing F at the data-entry ring menu, the screen presents a "Query by Example" form. This screen looks like the regular data-entry screen but allows you to type patterns and special "operators" into columns in order to locate a row or set of rows.

If you select the Find command and then press [ESC], all rows are selected. If you want to select a specific document or group of documents you can enter query patterns into columns. Using query patterns to locate information allows the user the flexibility to quickly access any row or group of rows in the table.

Relational operators such as > and < allow you to find rows with column values greater than or less than an entered value. The : operator lets you find a range of column values. The | helps you to locate rows by a list of acceptable column contents. Using the * and ? wildcard characters allow you to match complex patterns in "char" type columns.

The available operators and their functions are listed in the table below.

Operator	**Relationship**
=	Equal to
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
<>	Not equal
:	Range
|	Or list
?	Single character wildcard
*	Multi-character wildcard

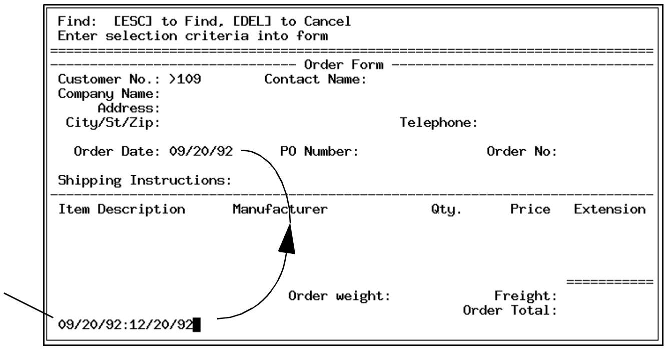
All operators may be used for any column with the exception of the wildcard operators, which work only in "char" type columns. Query searches are made by entering an operator and a pattern into one or more columns.

For example, to find all rows with last names that follow "Smith" in the alphabet you would type `>Smith` into the Last Name column.

When you have completed entering search criteria into the columns of the Find Query by Example screen, press [ESC] to perform the search. Find searches the entire table and displays the first row to match the criteria on the screen. The indicator at the screen bottom tells you how many rows in the table match your find criteria. If you see that your query is returning more rows than expected, you can interrupt the query by pressing the [DEL] key. All of the rows found up to the point at which [DEL] was pressed are displayed.

Sometimes the patterns used to locate rows exceed the physical column delimiters on the screen. When you reach the end of a column while entering find criteria on a query screen your cursor jumps to the lower left corner of the screen allowing you to keep entering characters and building your selection criteria.

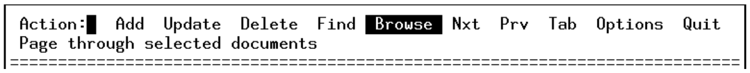
The following example shows patterns entered into a Find screen to locate all orders placed between September 20, 1986 and December 20, 1986 (09/20/86:12/20/86) where the customer number exceeded 109. The size of the field places no limitation on the criteria you can specify for a Find. In such cases the field continues at the lower left hand part of the form.



```
Find:  [ESC] to Find, [DEL] to Cancel
Enter selection criteria into form
==============================================================================
------------------------------- Order Form -----------------------------------
Customer No.: >109        Contact Name:
Company Name:
     Address:
 City/St/Zip:                               Telephone:

  Order Date: 09/20/92     PO Number:                Order No:

Shipping Instructions:
------------------------------------------------------------------------------
Item Description      Manufacturer             Qty.      Price   Extension



                                                                 ===========
                             Order weight:             Freight:
                                                   Order Total:
09/20/92:12/20/92
```

Entering selection criteria that is larger than the visible field continues on the bottom left portion of the form.

The Browse Command

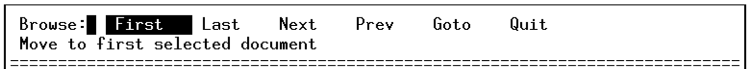


```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Page through selected documents
=============================================================================
```

The basic purpose of the Browse command is to display a summary of all selected documents on a Browse form. The Browse form is a scrolling array which allows you to quickly pick out the document you want to view or modify.

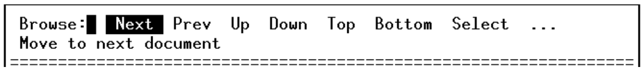
In order to use a Browse form with a scrolling array, a `browse.per` form must exist in the program directory. The `browse.per` form is created much like the main data-entry .per form. Even if you do not create a Browse form for your program the Browse function is still available, but only one document can be displayed at a time.

If your program does not have a `browse.per,` the Browse option changes the ring menu to display the following commands:



```
Browse:  First    Last    Next    Prev    Goto    Quit
Move to first selected document
==========================================================================
```

Programs that do include a Browse screen show a slightly different list of commands:



```
Browse:  Next  Prev  Up  Down  Top  Bottom  Select  ...
Move to next document
=================================================================
```

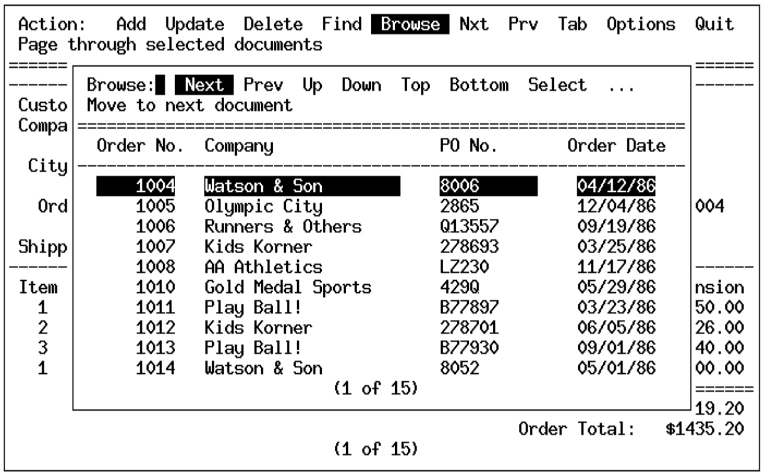
The ellipsis (…) indicate that other commands exist on the command line—they cannot all be shown at once. The commands not shown in the graphic above are accessed by moving the highlight to the far right, or by pressing the up or down arrow keys. Commands not shown in the graphic above are Goto and Quit.

The next two sections describe the commands found on the browse form and the browse commands available when a browse form is not used.

Browsing With a Browse Screen

When you first select Browse from within a program with a Browse Screen, the system opens the browse window and displays the first *n* rows in the table (where *n* is the number of lines on the browse screen) beginning with the current row shown on the data-entry screen. If you have previously selected a set of rows with the Find command, the window displays only rows from the active set.

The sample below shows a Browse screen for the `orders` table of the `stores` database detailed in the INFORMIX-4GL manuals:



```
 Action:   Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
 Page through selected documents
======┌──────────────────────────────────────────────────────────────┐======
------│ Browse:   Next  Prev  Up  Down  Top  Bottom  Select  ...      │------
 Custo│ Move to next document                                        │
 Compa│==============================================================│
      │  Order No.  Company                 PO No.      Order Date   │
  City│--------------------------------------------------------------│
      │      1004   Watson & Son            8006          04/12/86   │
   Ord│      1005   Olympic City            2865          12/04/86   │004
      │      1006   Runners & Others        Q13557        09/19/86   │
 Shipp│      1007   Kids Korner             278693        03/25/86   │
------│      1008   AA Athletics            LZ230         11/17/86   │------
 Item │      1010   Gold Medal Sports       429Q          05/29/86   │nsion
   1  │      1011   Play Ball!              B77897        03/23/86   │50.00
   2  │      1012   Kids Korner             278701        06/05/86   │26.00
   3  │      1013   Play Ball!              B77930        09/01/86   │40.00
   1  │      1014   Watson & Son            8052          05/01/86   │00.00
      │                         (1 of 15)                            │======
      └──────────────────────────────────────────────────────────────┘19.20
                                                 Order Total:   $1435.20
                                (1 of 15)
```

The top row is highlighted. You can move the highlight and scroll through rows with the browse commands. Browse commands are executed like all Fitrix *Screen* ring menu commands: by moving the highlight over the command and pressing [ENTER] or by typing the first letter of the command. After you exit the Browse option, the current row on the data-entry screen is the one that was highlighted on the browse screen. If you press the [DEL] key to exit you return to the original row (the one loaded when you selected Browse) rather than going to the one highlighted on the browse window.

You must exit Browse before you may update data.

Next: Moves the highlight to the next sequential row.

Prev: Moves the highlight to the previous row.

Up: Moves up through the preceding rows one "page" at a time.

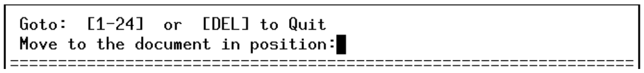
Down: Moves downward through succeeding rows one page at a time.

Top: Loads the first page of the active set (all rows of the table if the Find command has not been run) into the browse window.

Bottom: Displays the last page of rows.

Select: Selects the row currently highlighted. The selected row becomes the current document.

Goto: Access a row by typing the number of its order in the sequence of selected records (the active set). When you run Goto, the system prompts you for the number with a message like:



```
Goto:  [1-24]  or  [DEL] to Quit
Move to the document in position:█
=================================================================
```

After you type in a number between 1 and 24 and press [ENTER] the program displays that row highlighted in the first line on the browse form, and fills in remaining browse lines with the subsequent rows.

Quit: Closes the browse screen and returns to the data-entry form without selecting a row. The document current prior to selecting the Browse command remains current.

Browsing Without a Browse Screen

Since programs without a Browse screen use the original data-entry screen for the Browse option, only one row is displayed at a time. The Browse menu options also differs slightly.

First: Loads the first row of the selected group of rows.

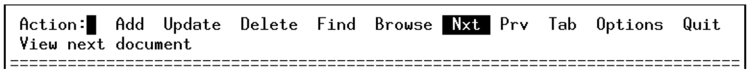
Last: Loads the last row of the selected group of rows.

Next: Loads the next row in the sort order sequence.

Prev: Loads the preceding row in the sequence.

Goto: Displays specified row. Selecting this option prompts for the number of the row to load. Enter the number of the row in the sort sequence for the table and press [ENTER] to load the row into the screen. The row specified is displayed in the Browse form.

The Next Command

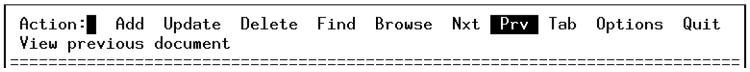


```
Action:█  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
View next document
===========================================================================
```

Using the Next command loads the next sequential row into the data-entry screen. Use the highlight or N key to select the Next command. You determine the sequence of rows for the individual program when you run the Code Generator. If you have used the Find command to select a set of rows, Next takes you to the next sequential row in that active set.

When the current row is the last one in the active set, Next loads the first row in the set.

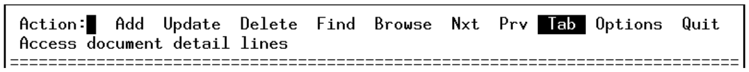
The Prev Command



```
Action:█  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
View previous document
===========================================================================
```

Prev is the opposite of the Next command. Selecting Prev loads the previous row of the defined sequence. Like Next, Prev displays only those rows in the active set. If you have run the Find command, Prev finds the previous record matching the Find search criteria. Running Prev when the current row is the first row in the active set loads the last row of the set.

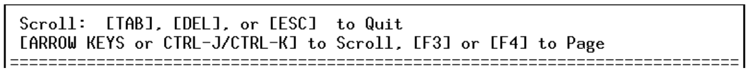
The Tab Command



```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Access document detail lines
===========================================================================
```

The Tab command is used in header/detail files to move from the header portion of the screen to the detail portion to view detail lines. Tab may be executed from the data-entry ring menu by highlighting the command and pressing [ENTER], typing a T, or by typing [CTRL]-[i].

While the screen cursor is in the detail line window, special detail line commands are available:



```
Scroll:  [TAB], [DEL], or [ESC]  to Quit
[ARROW KEYS or CTRL-J/CTRL-K] to Scroll, [F3] or [F4] to Page
===========================================================================
```

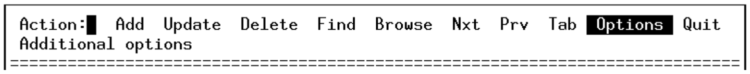
These commands are described in "Detail Line Commands" on page 3-13.

When the screen cursor exits the first column in the detail lines (with the [UP ARROW] key), the rows shift downward one line. If there are no more rows preceding the top line, the [UP ARROW] key causes an error message to appear:

```
There are no more rows in the direction you are going.
```

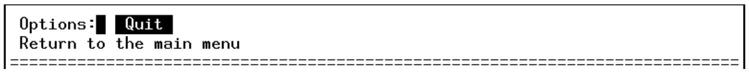
The same message appears when you press the [DOWN ARROW] key on the line of the last row for that header (if you are not updating data).

The Options Command



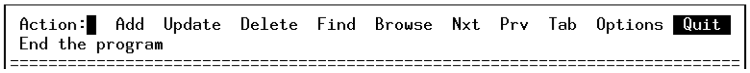
```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Additional options
===========================================================================
```

The Options command leads to a ring menu which may contain additional commands. This Options ring menu is typically the place where you would integrate new commands into the existing data-entry ring menu. In the absence of additional commands, the display resembles the following:



```
Options:  Quit
Return to the main menu
===========================================================================
```

The Quit Command



```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
End the program
==========================================================================
```

The Quit command of the data-entry ring menu ends the program. In addition to the highlight, you can quit by typing an E, Q, or an X.

Detail Line Commands

You may access the detail lines from the data-entry ring menu or with the [TAB] key while updating the screen. Tab is executed from the ring menu by highlighting the command and pressing [ENTER], pressing the [t] key, or pressing the [TAB] key.

When you access detail lines from the ring menu you are not allowed to update the detail line columns. When you press [TAB] while in Add or Update mode, you are allowed to both modify and add rows to the detail lines.

Although the detail line commands display only when you have run Tab from the menu, they are also available when you are updating the lines. The arrow keys move the screen cursor to the preceding or following line respectively. Typing an [UP ARROW] while on the top line of the screen displays the previous "page" of lines and, similarly, pressing a [DOWN ARROW] while on the last line of screen displays the following "page" of lines.

Pressing an [UP ARROW] while on the first detail row associated with the header row produces the error message below:

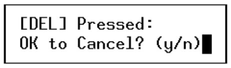
```
There are no more rows in the direction you are going.
```

If you are not updating lines, pressing [DOWN ARROW] or [ENTER] while on the last row produces the same message.

You may also "page" detail lines with your function keys. The [F3] key displays the next page and [F4] displays the preceding page. Paging in a direction with no more rows produces the aforementioned error message.

Saving Your Data

While adding or updating a document you press [ESC] to store data or [DEL] to abort the changes and return to the previous data. When [DEL] is pressed, the following prompt appears:



```
[DEL] Pressed:
OK to Cancel? (y/n)█
```

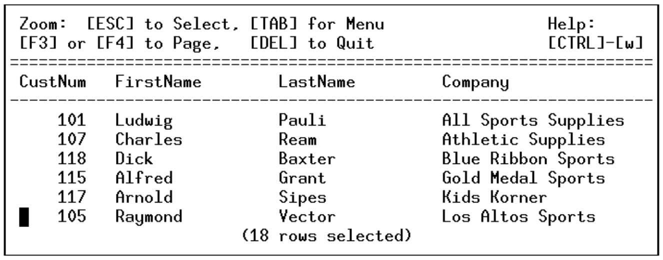
The prompt clarifies that the [DEL] command cancels only those changes made since the document was last stored. If you respond with N, the changes remain and you continue in the Add or Update mode. If you respond with Y, the changes since the last store ([ESC]) are deleted, and you are returned to the ring menu.

The Zoom Form

You can assign a zoom form to any field in a document. The Zoom function allows you to call up a form displaying information about the field you are currently in, or to call up a default form from another zoom form. The Zoom function is useful in situations when you must enter valid data that has already been defined, such as a customer number. By Zooming into the customer number field, you can select a valid customer number from a list.

Whenever a zoom form is available, the word (Zoom) appears on the upper right corner of the main data-entry form. The zoom form can be accessed by pressing [CTRL]-[z].

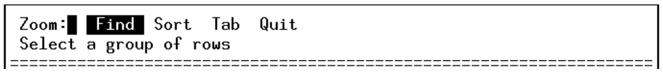
The zoom form:



```
Zoom:  [ESC] to Select, [TAB] for Menu                Help:
[F3] or [F4] to Page,   [DEL] to Quit                 [CTRL]-[w]
==================================================================
CustNum   FirstName        LastName         Company
------------------------------------------------------------------
    101   Ludwig           Pauli            All Sports Supplies
    107   Charles          Ream             Athletic Supplies
    118   Dick             Baxter           Blue Ribbon Sports
    115   Alfred           Grant            Gold Medal Sports
    117   Arnold           Sipes            Kids Korner
█   105   Raymond          Vector           Los Altos Sports
                       (18 rows selected)
```

Calling up a zoom form first displays a selection criteria form which allows you to limit the selection of data to display on the Zoom. For example you could choose to display only customers with last names that begin with an s by typing "s*" in the `Lastname` column and pressing [ESC].

Notice the commands at the top of the zoom form. Pressing [TAB] displays the following menu:



```
Zoom:  Find  Sort  Tab  Quit
Select a group of rows
==================================================================
```

Each of these commands may be executed like any other ring menu command: by highlighting it and pressing [ENTER] or by typing the first letter of the command. For more information on the Zoom feature refer to "The Zoom Form" on page 3-14.

The Find Command (Zoom)

The Find command allows you to search for a specific piece of information. For instance, if your zoom form displays information from your customer table and you do not remember the customer code of the customer you want to select off the top of your head, you could enter the first few letters of the customer code. If you enter HAF*, then all of the customer codes that begin with HAF are displayed on your zoom form. You can then move the highlight to the customer you want to select.

All of the find criteria available to the main data-entry form can be used to help limit your selection of items.

The Sort Command (Zoom)

The Sort command allows you to easily specify the field by which the selections are sorted. Typing S displays the following ring menu:

```
Sort:  [ESC] when done, [DEL] to Cancel
Enter ANY valid data into the column to sort by.
```

To select the columns you want your selection sorted by, use the [TAB], [ENTER], and arrow keys to move the highlight to the appropriate column, then type any valid character. For example, typing an a in the First Name column causes the items to be sorted alphabetically.

The Tab Command (Zoom)

When the cursor is in the zoom form’s ring menu, use the Tab command to return to selection mode where you can continue viewing the items and make your selection.

In select mode, when the cursor is in the detail lines, use the [TAB] key to return to the Zoom ring menu.

The Quit Command (Zoom)

The Quit command causes you to leave the zoom form without selecting anything.

The AutoZoom Feature

Another feature of the Code Generator is the AutoZoom. AutoZoom is a quick version of the regular Zoom feature. AutoZoom allows you to bypass function keys ([CTRL]-[z]) and the normal selection criteria form. AutoZoom is automatically enabled in all character fields in which the regular Zoom is enabled. You execute the AutoZoom when you enter an asterisk in a character field that supports Zoom functionality. This selects all documents that match the data in the field and displays them on a zoom form. For example, the AutoZoom enables you to call up a list of all the customer orders whose manufacturers’ names begin with H. This gives you very rapid access to the information you need even if you don’t remember a manufacturer’s full name. It also reduces the amount of typing you need to do. For more information on the AutoZoom feature refer to "Creating a Permanent Zoom Filter" on page 10-62.

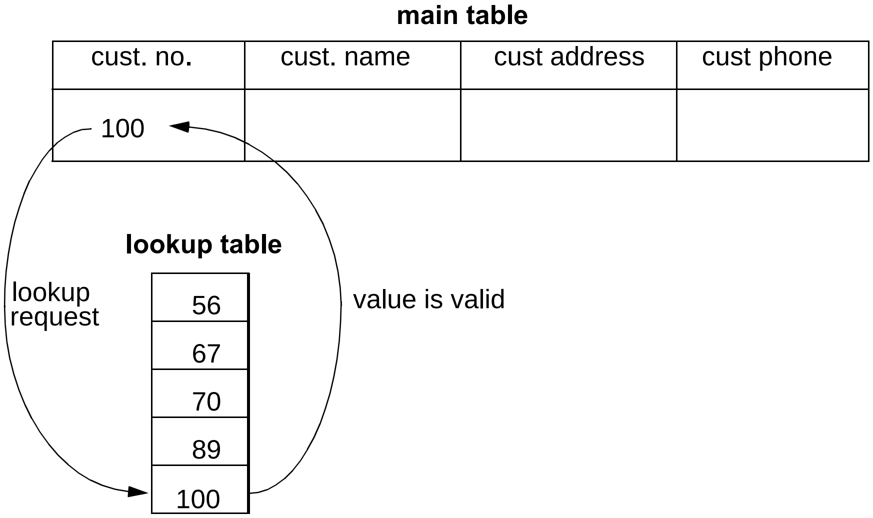
Introduction to Lookups

Lookups evaluate the data that is entered into a field and match that data against data in any table. A lookup placed on a field serves two purposes: verification, and to retrieve related data from a table.

1. **Verification** - lookups verify that a value entered in a field exists in another table. For example, on an order entry screen, the customer field has a lookup on it to verify that the value entered exists in a customer table. If the user entered a value that did not exist in the customer table, the lookup would cause the following error message to occur:

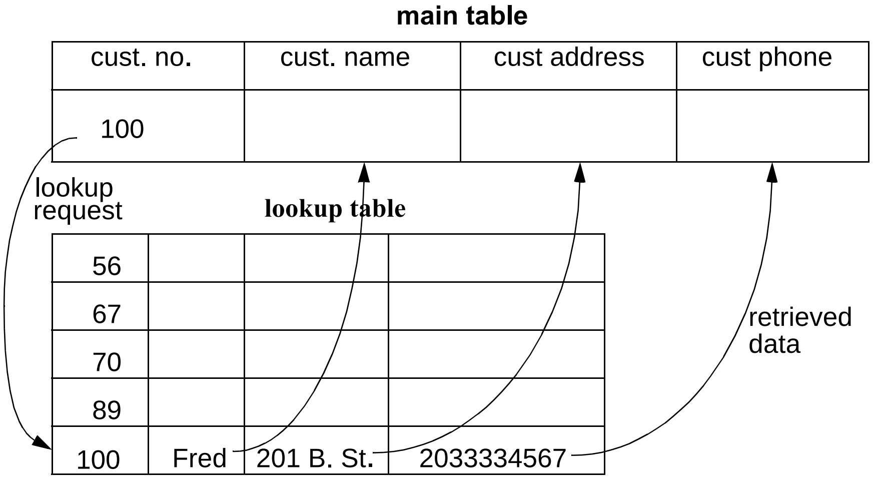
   ```
   "Value is not in the list of valid data"
   ```

   Following diagram illustrates a verification lookup.



2. **Looking up data** - lookups are used to pull data out of tables. When the user enters a value into a field, a lookup can pass that value to a table and retrieve a corresponding value into an adjacent field. For example, when the user enters a value into a customer code field on an order entry screen, a lookup can take that value and "lookup" the customer name in the customer table and place the customer name into a customer name field.

The following diagram illustrates a data retrieval lookup.


main table

cust. no.	cust. name	cust address	cust phone
100			

lookup table

56			
67			
70			
89			
100	Fred	201 B. St.	2033334567

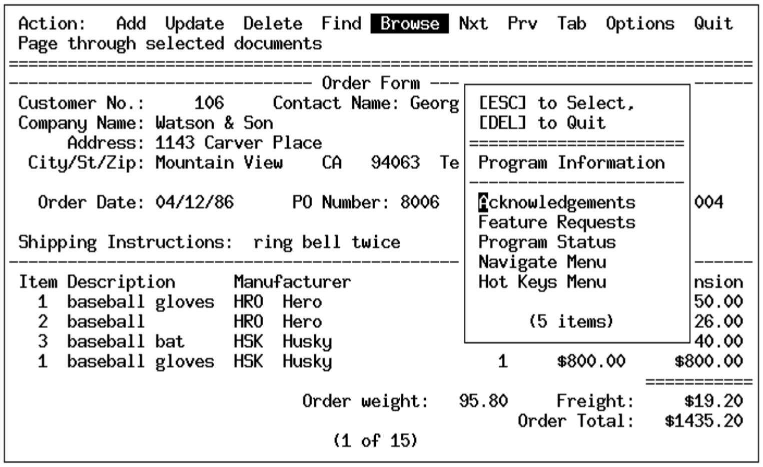
For more information on lookups refer to "Defining Lookups" on page 7-23.

Program Information Menu

All data-entry applications generated by the Code Generator include a Program Information menu. This menu contains options which provide additional functionality as well as program-level information.

The Program Information menu is accessed via the default Hot Key setting [CTRL]-[y]. Another method for displaying this menu is discussed in "Hot Keys Menu" on page 3-22

Once selected, the Program Information menu appears on top of the current form as follows:



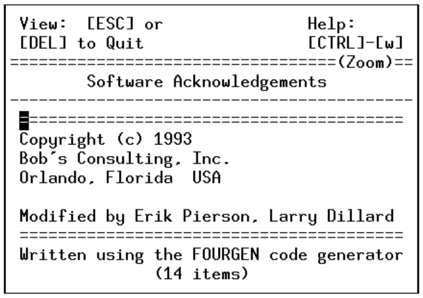
```
 Action:   Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
 Page through selected documents
==============================================================================
------------------------------- Order Form --- ----------------------- ------
 Customer No.:     106     Contact Name: Georg | [ESC] to Select,      |
 Company Name: Watson & Son                    | [DEL] to Quit         |
      Address: 1143 Carver Place               |=======================|
  City/St/Zip: Mountain View    CA   94063  Te | Program Information   |
                                               |-----------------------|
   Order Date: 04/12/86      PO Number: 8006   | Acknowledgements      | 004
                                               | Feature Requests      |
 Shipping Instructions:  ring bell twice       | Program Status        |
---------------------------------------------- | Navigate Menu         | ------
 Item Description      Manufacturer            | Hot Keys Menu         | nsion
   1  baseball gloves  HRO  Hero               |                       | 50.00
   2  baseball         HRO  Hero               |       (5 items)       | 26.00
   3  baseball bat     HSK  Husky              |_______________________| 40.00
   1  baseball gloves  HSK  Husky                  1     $800.00    $800.00
                                                                =========== 
                             Order weight:   95.80    Freight:     $19.20
                                                  Order Total:   $1435.20
                                (1 of 15)
```

Five options appear on the menu. Use the arrow keys to position the cursor over the desired option. The [ESC] key selects the option. These options are explained next.

Viewing Program Acknowledgements

The first option on the Program Information menu is titled Acknowledgements. This option displays the Software Acknowledgements form, which displays all acknowledgements pertaining to the particular program.

The Software Acknowledgements form.



```
View:  [ESC] or               Help:
[DEL] to Quit                 [CTRL]-[w]
=================================(Zoom)==
       Software Acknowledgements
-----------------------------------------
=======================================
Copyright (c) 1993
Bob's Consulting, Inc.
Orlando, Florida  USA

Modified by Erik Pierson, Larry Dillard
=======================================
Written using the FOURGEN code generator
              (14 items)
```

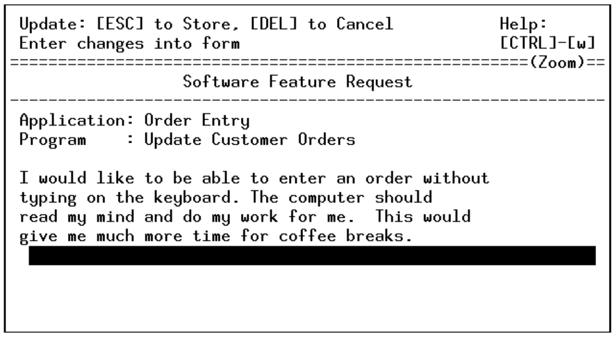
This form displays the default Code Generator comments as well as any acknowledgements added to the default forms by the programmer.

The window that displays the acknowledgements varies in size depending on the text shown.

Entering Feature Requests

The Code Generator provides applications with the ability to allow users to store feature request ideas without significantly disrupting the flow of work.

When the Feature Request option is selected, a form appears allowing you to enter a request.



```
Update: [ESC] to Store, [DEL] to Cancel          Help:
Enter changes into form                          [CTRL]-[w]
=====================================================(Zoom)==
                  Software Feature Request
-------------------------------------------------------------
Application: Order Entry
Program    : Update Customer Orders

I would like to be able to enter an order without
typing on the keyboard. The computer should
read my mind and do my work for me.  This would
give me much more time for coffee breaks.
```

The Feature Request form is the interface through which user comments are transferred to the local `errlog` file established in the current directory. All comments stored on this form are written to the `errlog` file as soon as the user exits the application. The `errlog` file is also written to automatically by fatal errors, and deliberately by users logging the text of an error message through the Errors form. The system administrator or programmer can read through the local `errlog` file to gather information on error messages or features requested by users.

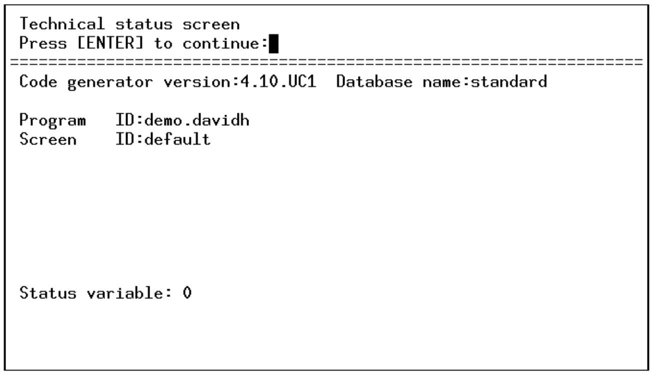
Like all unstructured text-entry forms created with the Code Generator, this form allows usage of the Informix edit keys.

The Zoom feature is available from the Software Feature Request form. The Zoom leads to the Default Software Feature Request form, which provides you or the system administrator an opportunity to specify prompts or messages for those who enter feature requests. Information on the Default Software Feature Request form appears on the Software Feature Request form. The user may edit, modify, or delete the default text prior to storing the feature request.

Program Status

The Program Status option on the Program Information menu provides users with a snapshot of current program information. When selected, this option displays a form containing the Code Generator version number, database name, program ID, screen ID, current field, and the current value of the `status` variable.

The Technical Status form resembles the following illustration:



```
Technical status screen
Press [ENTER] to continue:
=================================================================
Code generator version:4.10.UC1  Database name:standard

Program   ID:demo.davidh
Screen    ID:default






Status variable: 0
```

The Technical Status form is view-only. This form can also be viewed (with more specific information) from the Error Detail ring menu.

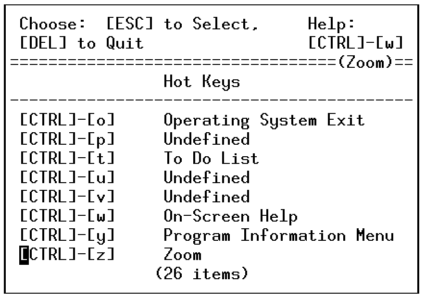
Navigate Menu

The Code Generator creates code that handles actions in the form of events, separate from the keys used to invoke them. Due to this structure, events can be assigned (and re-assigned) to a particular key or keystroke combination. Known events are listed on the Navigate menu.

Navigation is a User Control topic; the Navigate menu only appears when the Enhancement Toolkit is present on the system. If you do have the Enhancement Toolkit, refer to the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

Hot Keys Menu

The Hot Keys option on the Program Information menu provides an instant reference to the default Hot Key settings. When this option is selected, the Hot Keys form is displayed. The Hot Keys form contains a list of events currently associated with keys or keystroke combinations. The form appears as follows:



```
 Choose:  [ESC] to Select,      Help:
 [DEL] to Quit                  [CTRL]-[w]
==================================(Zoom)==
                Hot Keys
------------------------------------------
 [CTRL]-[o]     Operating System Exit
 [CTRL]-[p]     Undefined
 [CTRL]-[t]     To Do List
 [CTRL]-[u]     Undefined
 [CTRL]-[v]     Undefined
 [CTRL]-[w]     On-Screen Help
 [CTRL]-[y]     Program Information Menu
 [CTRL]-[z]     Zoom
               (26 items)
```

The user can scroll through the definitions with the arrow keys or page with the [F3] (page down) and [F4] (page up) keys.

If the user presses [ESC] to select a hot key definition, the event (if any) associated with that key is carried out.

Applications installed with the Enhancement Toolkit allow users to redefine existing Hot Key definitions. In the absence of the Enhancement Toolkit package, users have access to several predefined Hot Key settings that cannot be modified.

Default Screen Attributes

Code generated with Fitrix *Screen* is given default screen display attributes. These attributes serve as display conventions, and enhance the consistency of applications.

Attribute Conventions

Windows:

- First window (border only)—blue (dim).
- All other windows (and their forms)—white (regular).

Display statements:

- Program information (Press [ENTER], etc.)—never has attributes (i.e., shares the attribute of the current window).
- Data—display in attribute(red) (all display and display array statements have attribute(red)). This prints as bold on non-color terminals.

During Input:

- When entering a data field—display with attribute(reverse).
- When leaving the data field—display with attribute(red).

During Input Array:

- The input array does not follow the last displayed attribute (like input) and it changes the attribute of all data to that of the window. The convention is to show the data in attribute(red) while not in the input array statement, and show the data without an attribute during input array.
- Entering a data field—display with attribute(reverse) (like input).
- Leaving a data field—re-display without an attribute. (i.e., same attribute as the current window).
- Leaving input array statement—re-display screen array in attribute(red). This restores the screen array to the original color before the input array statement.

Part Two

The Form Painter

4

Form Painter Basics

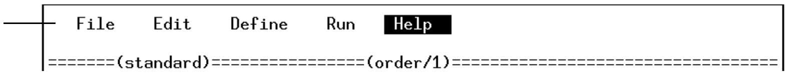
This section of the documentation covers:

- Pull-down menus in the Form Painter
- Active/inactive menu options
- Symbols displayed with menu options
- Using on-line help in the Form Painter

Form Painter Menus

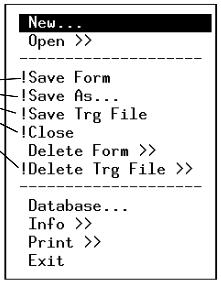
The Fitrix *Screen* Form Painter interface provides pull-down menus relating to form specification. The appearance and usage of these menus is the subject of this section of the documentation. The purpose of the individual options on pull-down menus is the subject of the next section in this manual.

The Form Painter contains five pull-down menus that you can access through commands on the menu line. Each pull-down menu can be displayed by using the arrow keys to highlight the appropriate command and then pressing [ENTER], or by typing the first letter of the command (e for the Edit pull-down menu). Immediately below the menu line is the Prompt line, which displays a brief description of the highlighted command.


File
Edit
Define
Run
Help
=======(standard)================(order/1)==================================

menu line

The appearance of the options listed on the pull-down menus varies depending on the current situation. The system indicates whether an option is accessible—an option that is not accessible appears dimmed, with a preceding exclamation mark. For instance, if no form has been made current (the editor screen is blank), a "save" option is meaningless. The save option then appears on the pull-down as follows:

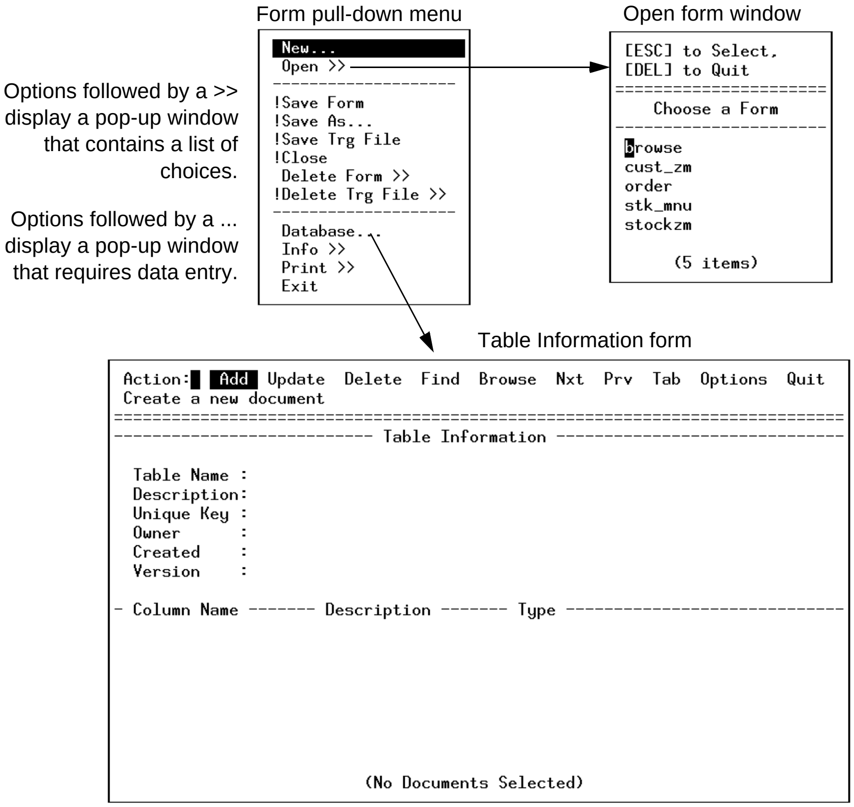


options marked with an !
are currently unavailable

The exclamation point is added in case a terminal does not have dim/bright attributes. In short, if an option appears bright on the terminal screen (colored red if you have a colored terminal) *without* a preceding exclamation point, it is currently accessible.

Some options appear with symbols appended to them. Options followed by ellipses (`...`) lead to a pop-up window requiring data-entry. An example is found on the Form pull-down menu. The New... option leads to a pop-up window with a prompt for you to specify the name of the form being created.

Options that display the "greater than" symbols (>>) display a pop-up window containing a list of possible selections. For instance, the Open >> option opens a window and displays a list of existing form names for selection.



Form pull-down menu

```
 New...
 Open >>
--------------------
!Save Form
!Save As...
!Save Trg File
!Close
 Delete Form >>
!Delete Trg File >>
--------------------
 Database...
 Info >>
 Print >>
 Exit
```

Open form window

```
[ESC] to Select,
[DEL] to Quit
======================
    Choose a Form
----------------------
browse
cust_zm
order
stk_mnu
stockzm

      (5 items)
```

Options followed by a >> display a pop-up window that contains a list of choices.

Options followed by a ... display a pop-up window that requires data entry.

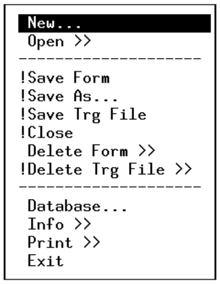
Table Information form

```
 Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
 Create a new document
============================================================================
--------------------------- Table Information ------------------------------

  Table Name :
  Description:
  Unique Key :
  Owner      :
  Created    :
  Version    :

- Column Name ------- Description ------- Type -----------------------------





                        (No Documents Selected)
```

The File Pull-Down Menu

The File command on the menu line is used to access options that relate to managing forms. Select the File command to view a list of options contained on the File pull-down menu.

The File pull-down:

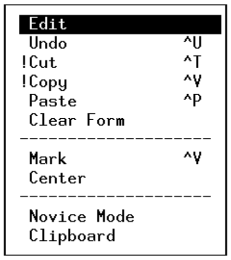


An option's accessibility on the File pull-down menu depends on whether there is a current form. That is, if there is no current form, options relating to saving or closing a form are irrelevant and cannot be accessed even though they are visible on the menu.

The Edit Pull-Down Menu

The Edit command leads to a pull-down containing options commonly used when revising forms. The Edit pull-down menu also contains an option used to toggle between Novice and Expert modes. For more information on modes, see the section "Defining Fields and Forms."

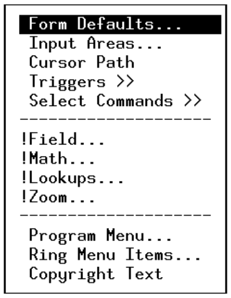
The Edit pull-down menu:



The Define Pull-Down Menu

The Define command leads to a pull-down containing options used to define fields and forms. Since options on this menu are form-specific, they are inaccessible until a form is made current. The only exception is the Copyright Text option, which is used to modify global information regarding source code control and copyright(s).

The Define pull-down menu:

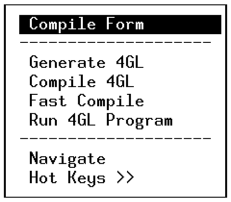


The Run Pull-Down Menu

The Run command displays a pull-down containing options for compiling forms, generating code, compiling code, and running applications.

The Run pull-down menu also contains the Navigate option. Navigation is a powerful feature for carrying out pre-established events on the system without losing your current place. Navigation becomes even more useful when you assign navigation events to Hot Keys. Both the Navigate and Hot Keys features are documented in the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

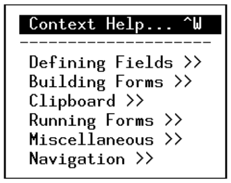
The Run pull-down menu:



The Help Pull-Down Menu

The Help command allows you to access options that provide reference information on aspects of the Form Painter. The Help pull-down menu also includes an option for context-sensitive help information.

The Help pull-down menu:

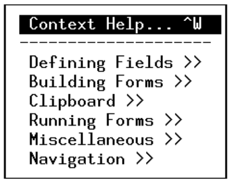


Using Form Painter Online Help

Online help is available to those working with the Form Painter. Help can be accessed by topic through the Help pull-down menu. In addition, context-sensitive help can be accessed by pressing [CTRL]-[w] from any point in the Form Painter.

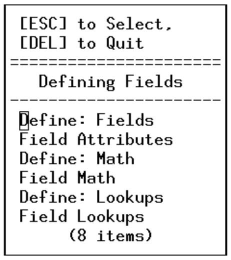
Form Painter Topic Help

In order to view help text for a topic regarding the Form Painter, select the Help command. The Help pull-down menu contains main topics under which the topic-based help text is organized. The Help pull-down menu appears as follows:



```
 Context Help... ^W
--------------------
 Defining Fields >>
 Building Forms >>
 Clipboard >>
 Running Forms >>
 Miscellaneous >>
 Navigation >>
```

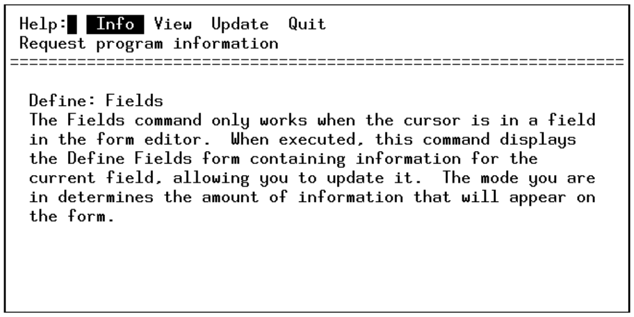
The greater than signs (>>) following each option on the Help pull-down menu indicate that selection leads to a picker list containing subtopics. For example, if you select the option Defining Fields >>, you see the following picker list of help text subtopics:



```
 [ESC] to Select,
 [DEL] to Quit
======================
   Defining Fields
----------------------
Define: Fields
Field Attributes
Define: Math
Field Math
Define: Lookups
Field Lookups
     (8 items)
```

To scroll through the list of subtopics for which help text is provided, you can use the up and down arrow keys, or page with the INFORMIX-defined [F3] and [F4] paging keys. Use the [ESC] key to select a topic. Once you select a topic, the Help form appears, displaying the help text defined for the selected topic.

The Help form appears as follows:



```
Help:  Info  View  Update  Quit
Request program information
================================================================

 Define: Fields
 The Fields command only works when the cursor is in a field
 in the form editor.  When executed, this command displays
 the Define Fields form containing information for the
 current field, allowing you to update it.  The mode you are
 in determines the amount of information that will appear on
 the form.
```

The commands on the command line of the Help form are explained below:

Info: Leads to the Program Information Menu, which contains a list of five selections. For information on the Program Information Menu refer to "Program Information Menu" on page 3-19.

View: Used to scroll through the text displayed on the Help form. The INFORMIX-defined cursor movement keys (arrow keys, [F3], and [F4]) are available while viewing the text.

Update: Selected to enter or modify help text. To store text entered on the form, use the [ESC] key.

Quit: Exits the Help form and returns you to your position prior to entering the Help form.

Context-Sensitive Help Text

Context-Sensitive help text is available from any place in the Form Painter by pressing [CTRL]-[w]. A Help form appears containing information pertinent to your current location.

5

Managing Forms

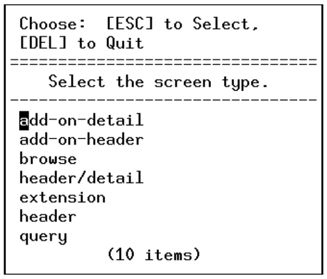
This section covers the following:

- Creating a new form
- Opening a form
- Saving a form
- Automatic Saves
- Closing a form
- Deleting a form
- Establishing software acknowledgements
- Printing a form

Creating a New Form

The New option on the Form pull-down menu lets you create a new .per form.

The Form Painter has two operating modes: Expert and Novice. The operating mode is determined by toggling the Expert/Novice option on the Edit pull-down menu. In Novice mode, the number of choices you have for various options is limited. The operating mode in which you are currently operating has an effect on the actions that take place following the selection of the New option. If you are operating in Expert mode, you first see a pop-up list. The list contains the types of forms that can be created.:

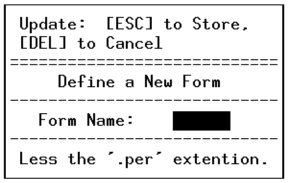


```
Choose:  [ESC] to Select,
[DEL] to Quit
================================
   Select the screen type.
--------------------------------
add-on-detail
add-on-header
browse
header/detail
extension
header
query
         (10 items)
```

If you operate in the Novice mode you do not see this list; Novice mode is used strictly for header-only forms. More information about Expert and Novice modes can be found in "Novice and Expert Modes" on page 7-2.

Form types are discussed in "Creating Screen Forms" on page 10-1.

After selecting the type of screen, a dialog box appears prompting for the new form name. The form appears as follows:



```
Update:  [ESC] to Store,
[DEL] to Cancel
============================
     Define a New Form
----------------------------
  Form Name:
----------------------------
Less the '.per' extention.
```

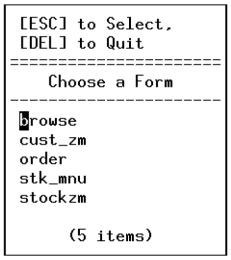
You can enter a form name up to seven characters long. This length is an established naming convention. The extension ".per" is appended to all form names when the file is saved.

Note

Use the AutoForm feature to quickly create a new form using all columns in a table. Refer to "Using the AutoForm Feature" on page 9-6.

Opening a Form

The Open option on the Form pull-down menu lets you open an existing .per form. The "greater than" symbols next to the name of the option indicate that selection leads to a pop-up window containing a list box of existing file names.



```
[ESC] to Select,
[DEL] to Quit
=====================
    Choose a Form
---------------------
browse
cust_zm
order
stk_mnu
stockzm

      (5 items)
```

The list box displays forms in the current directory less their .per extension.

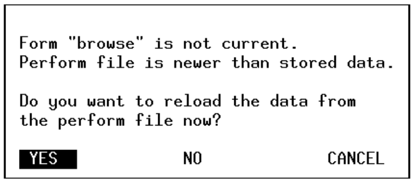
The cursor can scrolled with the arrow keys. You can use the INFORMIX-4GL paging keys ([F3] and [F4]) to page through the current list of forms. If [F3] and [F4] don’t work, you have a problem with the termcap settings for your terminal. Please refer to "Writing Termcap Entries" on page G-6.

Database vs. Disk Copies of a Form

The Form Painter stores form file information in the database. This constitutes a separate source of data (apart from the disk .per file) from which you can open a .per form.

When you open an existing form file, the information is normally read in from the database, not from the .per file on disk. If the information in the .per file on disk is newer than the information in the database, a prompt appears asking whether the disk version should be reloaded, thus updating the database copy.

A dialog box appears as follows:



```
Form "browse" is not current.
Perform file is newer than stored data.

Do you want to reload the data from
the perform file now?
```

Selecting YES causes the system to parse in the data from the disk version of the form file, overwriting the database copy. Selecting NO causes the system to load in information from memory, not from the disk .per file. Selecting CANCEL cancels the process of opening a file.

The benefit of storing form file information in both disk and database formats is that if the disk version of the .per somehow becomes corrupted or lost, you can restore the form from the information stored within the database.

.per Form Requirements

In order for a .per form to load correctly into the Form Painter a number of conditions must be met. Read this section if you want to load .per files into the Form Painter that were not created by the Form Painter.

A .per form needs to contain the following attributes:

- It must conform to the INFORMIX-4GL form style.
- The Form Painter requires .per forms to have the following five sections: DATABASE, SCREEN, TABLES, ATTRIBUTES, and INSTRUCTIONS. (INFORMIX form specification files do not require an INSTRUCTIONS section.) In addition, a FGSS section is required any time you want to use Math, Zoom, Lookup, or Triggers.
- If the .per contains a FGSS section the word FGSS must be located at the beginning of this section and the F in FGSS must appear in the first column.
- Field tags must be less than 6 characters long.

For more information on the specific attributes that compose a .per form, refer to "The .per Specification File" on page B-1.

Troubleshooting a Non-Generated .per File

If you are loading a .per file created with vi into the Form Painter to work with, and you get the following error:

```
-4500
 A numeric conversion error has occurred due to
 incompatibility between a calling program and
 its function parameters or between a variable
 and its assigned expression.
```

You need to go through the .per file that is causing the problem and make sure *all* field tags defined in the screen section are also *all* defined in the attribute section.

Saving a Form

Forms can be saved at any stage of development; you can save forms as incomplete "fragments" or as complete .per files ready for compilation.

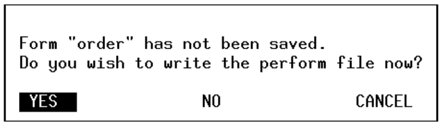
It is important to know that the Form Painter stores form file information in the database—this constitutes a separate source of data from which a .per form can be written. Unless you modify the .per file outside of the Form Painter program, the database always contains the most current copy. When you exit a file, your only decision is whether to update the disk copy (the actual .per file) with the information from the database. That is, if the form file is incomplete, you may not want to spend time saving it as a complete .per file. In that case, you can choose not to save it as a .per file, and then exit without deleting the database copy.

If the .per file becomes corrupted or deleted, the database information for that file can serve as a backup, restoring the form to the state in which it was last written to disk.

The deliberate method of saving a form file as a .per file involves using the Save option on the Form pull-down menu. Once you select it, the Form Painter composes the form information into a structured .per form specification file, ready for code generation.

A second method for saving the current form file into a .per file is provided when you exit the current file without having saved it prior to the last edit. That is, as you close a file, attempt to open a new file, or attempt to exit the Form Painter, a prompt appears providing an opportunity to save the current form as a .per file.

The prompt appears as follows:



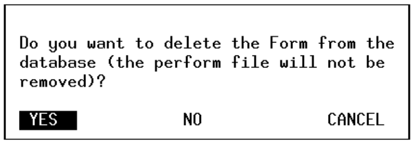
Form "order" has not been saved.
Do you wish to write the perform file now?

YES: Writes the completed .per file.

CANCEL: Cancels the operation and returns control to the Form pull-down menu without saving the current file.

NO: Displays a new prompt providing the opportunity to retain the incomplete file—it is not written as a .per file but is retained in the database.

This prompt appears as follows:

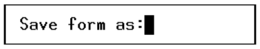


Do you want to delete the Form from the database (the perform file will not be removed)?

The Form Painter maintains a database copy of files you create with the Form Painter. The previous prompt determines whether the file should be removed from the database (YES) or simply retained as an incomplete file (NO).

Saving a Form Under a New Name

A separate option on the Form pull-down menu allows you to save the current file under a new name. This option is titled Save As. After selecting this option, the Form Painter produces a pop-up window containing a prompt for a new file name. The window appears as follows:



Save form as:

When the new name is entered, the Form Painter composes and saves the file under the new name. Once the save is completed, the cursor reappears in the Form Editor. Although you save the information to a new file, you continue to work with the "old" file.

Automatic Save

The Form Painter does not require that the .per file on disk be the master copy of the form. The generator, compiler, and printer, on the other hand, require that the .per file be the master copy of the form.

The current form is saved automatically and composed into a structured .per form specification file when you select the Compile Form option on the Run pull-down menu.

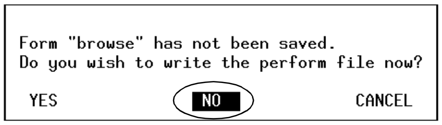
When you execute the Compile Form option, the Form Painter checks to determine whether the file has been modified since the last save. If there have been changes since the last save, the form file is automatically composed and saved prior to compilation.

Likewise, the Generate 4GL option performs an automatic save if you have made changes more recent than the modification date for the file.

The Print option on the Form pull-down menu also causes the database information to be written out to the .per file, depending on whether the database information is newer.

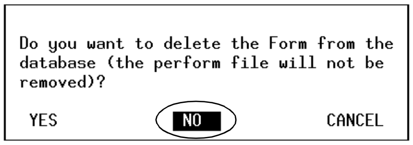
Saving an Incomplete Form

An incomplete form is one that has not been written as a complete .per file. There is only one method for saving an incomplete form. You must first select one of the following options: Close, Exit, Quit, or New. At the first prompt, choose not to save the file as a .per file.



Form "browse" has not been saved.
Do you wish to write the perform file now?

Next, choose not to delete the database information for the form file.



Do you want to delete the Form from the database (the perform file will not be removed)?

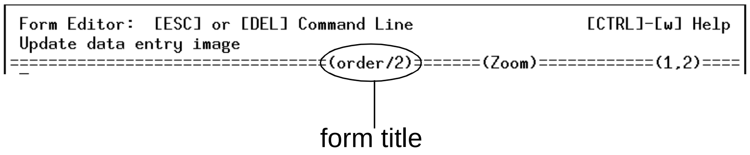
This way, the incomplete information in the form file remains in the database, but is not written out to disk. At the next editing session, you can simply select the form file by name; the system loads the form file from the database.

Closing a Form

Use the Close option on the Form pull-down menu to close the current file. Files get closed when loading another file or exiting the Form Painter.

As long as a file is current, its name appears centered on the line below the Prompt line.

Example:



```
Form Editor:  [ESC] or [DEL] Command Line                [CTRL]-[w] Help
Update data entry image
==============================(order/2)======(Zoom)============(1,2)====
_
```

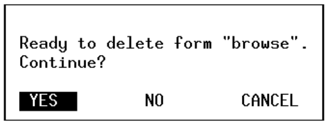
The Close option clears out text on the Form Editor and removes the filename display. If you have modified the file since the last time you saved it, a prompt (similar to the one displayed previously in this section) sequence appears to ask you for storage instructions.

When the Close option appears on the Form pull-down menu preceded by an exclamation mark (and/or dimmed), there is no current file and, hence, no file to close.

Deleting a Form

The Delete option on the Form pull-down menu allows you to delete existing .per form specification files. Once the Delete option is selected, a picker list appears, displaying each .per file in the current directory. To delete a file displayed in the picker list, simply select that file. A verification prompt next appears, requesting confirmation on the delete command. Answering yes deletes the selected file. The .per file is deleted from both the disk and the database.

The Delete Form prompt:



Ready to delete form "browse". Continue?

Recovering a Deleted File

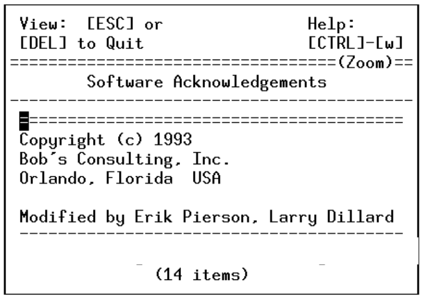
When you delete a form with the Delete Form option, the physical .per file is not actually removed from the disk but rather it is moved from the current directory to the `/tmp` directory. This means that if you accidently delete a form, you may be able to recover it by copying the .per form from the `/tmp` directory back to the original directory.

Establishing Software Acknowledgements

You can establish software acknowledgements for applications you create and generate with the Form Painter. While the acknowledgement text does not appear on the .per form, it is attached to the application code during generation with the Fitrix *Screen* Code Generator.

There are three stages to creating acknowledgements:

1. The Software Acknowledgements form.



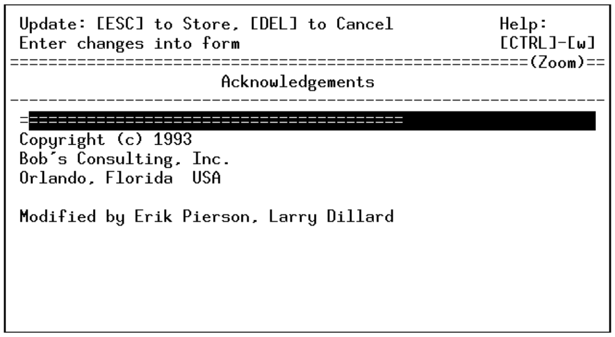
```
 View:  [ESC] or               Help:
 [DEL] to Quit                 [CTRL]-[w]
=================================(Zoom)==
        Software Acknowledgements
-----------------------------------------
 ========================================
 Copyright (c) 1993
 Bob´s Consulting, Inc.
 Orlando, Florida  USA

 Modified by Erik Pierson, Larry Dillard
 ----------------------------------------

               (14 items)
```

This view-only form is displayed when the end-user selects the Acknowledgements option from the Program Information Menu. This form displays the default Code Generator comments as well as any acknowledgements added to the following two forms.

The window that displays the acknowledgements varies in size depending on the text shown.

Pressing [CTRL]-[z] displays the following Acknowledgements form.

2. The Acknowledgements form.

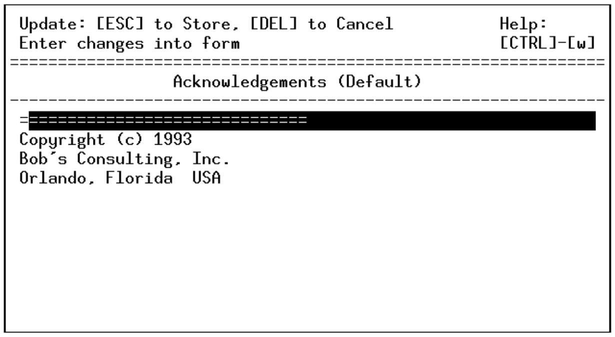


```
 Update: [ESC] to Store, [DEL] to Cancel          Help:
 Enter changes into form                          [CTRL]-[w]
======================================================(Zoom)==
                      Acknowledgements
--------------------------------------------------------------
 ========================================
 Copyright (c) 1993
 Bob´s Consulting, Inc.
 Orlando, Florida  USA

 Modified by Erik Pierson, Larry Dillard
```

This form enables you as the developer to add any type of acknowledgements you wish to your programs. Text added on this form shows up on the Software Acknowledgements form for this particular program only.

Pressing [CTRL]-[z] displays the following defaults form.

3. The Acknowledgements (default) form.



```
 Update: [ESC] to Store, [DEL] to Cancel          Help:
 Enter changes into form                          [CTRL]-[w]
==============================================================
                 Acknowledgements (Default)
--------------------------------------------------------------
 ==============================
 Copyright (c) 1993
 Bob´s Consulting, Inc.
 Orlando, Florida  USA
```

This form allows you, as the developer, to add default acknowledgement text to every Software Acknowledgement form in the application.

Printing a Form

Use the Print option on the Form pull-down menu to print an existing .per form specification file. The Print option uses the value currently found in `$SPOOLER`; the default is `lp`. If `lp` is unsuitable for your system, you need to adjust the value of `$SPOOLER`.

If any changes have been made since the last time you saved the form, the Form Painter automatically saves the form prior to printing. The Print option leads to a window containing a picker list, which allows you to specify the file you want to print.

Use the INFORMIX-defined [F3] and [F4] paging keys to scroll through the list. Select a form by moving the cursor to the proper row and pressing [ESC].

6

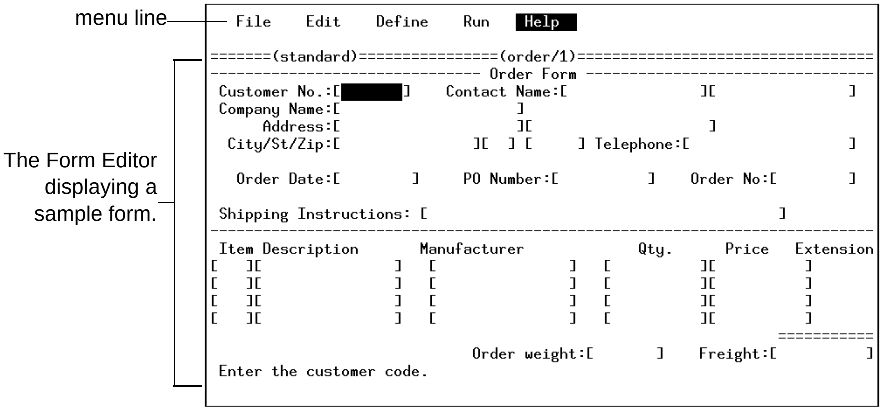
Editing Forms

This section covers:

- The Form Editor
- Editing keys
- Undoing edits
- Centering text
- Using the text Clipboard
- Marking, cutting, and copying text blocks

The Form Editor

The bottom section of the Form Painter interface is called the Form Editor. The Form Editor displays the image of the form. It is here where the image of a form is created.



menu line

The Form Editor displaying a sample form.

```
 File    Edit    Define    Run    Help

=======(standard)================(order/1)=================================
------------------------------- Order Form ---------------------------------
 Customer No.:[       ]    Contact Name:[               ][               ]
 Company Name:[                    ]
      Address:[                    ][                      ]
  City/St/Zip:[                ][  ] [     ] Telephone:[                 ]

   Order Date:[        ]     PO Number:[          ]    Order No:[        ]

 Shipping Instructions: [                                        ]
----------------------------------------------------------------------------
 Item Description       Manufacturer             Qty.      Price   Extension
[   ][               ]  [               ]  [          ][           ]
[   ][               ]  [               ]  [          ][           ]
[   ][               ]  [               ]  [          ][           ]
[   ][               ]  [               ]  [          ][           ]
                                                                 ===========
                             Order weight:[       ]    Freight:[           ]
 Enter the customer code.
```

The Form Painter Edit Commands

The only time the cursor appears in the Form Editor portion of the screen is when a form file is current. When a form is current, the cursor can be toggled back and forth between the Form Editor and the menu line by pressing the [ESC] key. You can also switch from the menu line to the Form Editor by executing the Edit Form option, found on the Edit pull-down menu.

When the cursor is in the Form Editor, you have access to a number of INFORMIX-defined editing keys. The Form Painter uses additional keys that enhance the "painting" environment. The table shown in this section lists Form Editor keys and their significance.

Keys	Action
[CTRL]-[a]	toggle between insert and overstrike
[CTRL]-[x]	delete character
[CTRL]-[d]	delete to the end of the line
[CTRL]-[u]	undo an edit
[CTRL]-[v]	mark/copy
[CTRL]-[t]	cut
[CTRL]-[p]	paste
[CTRL]-[w]	context help
[F1]	insert a blank line
[F2]	delete a line
[ENTER]	move to the beginning of the next line
[HOME]	move to the top left corner of the form
[	define a new field
]	lengthen an existing field
[ESC]	toggle between command and edit mode
[DEL]	go to command mode

It is worth noting here that the INFORMIX termcap definitions are used. If you find that the keys shown in the previous table do not operate as expected, the problem is likely to be in the termcap definition used for your particular terminal. For information regarding the specification of termcap definitions, please see "Writing Termcap Entries" on page G-6.

Undoing the Previous Edit

The Form Painter editor allows you to "undo" your previous edit with the Undo option on the Edit pull-down menu. The Form Painter interprets an edit as the most recent single change—typically the result of one command. For instance, the following list of edits are all considered individual, and could be reversed with the undo command:

- deleting text to the end of a line ([CTRL]-[d]). The undo command restores *all* characters deleted by this action.
- adding a single, contiguous line of text to a form. The undo command applies to all text entered on the same line since the last time the [ESC] key was pressed.
- using [CTRL]-[x] (within a defined field) to shorten the length of a field. The undo command returns the right delimiter (]) to its original position.
- reversing uninterrupted deletions using [CTRL]-[x]. The undo command, when used immediately, restores contiguous characters deleted successively.
- undoing a cut replaces an entire block of text if it was marked and cut.
- undoing a paste replaces a block of text that was pasted into the form from the Clipboard.
- undoing an "undo" reverses the effect of the undo command just executed.

The undo command only affects the latest edit. For example, assume the undo command (when first pressed) removes the last word typed. If executed again (with no subsequent changes), undo restores the removed word.

Centering Text on the Form

The Edit pull-down menu offers an option enabling you to center any line specified on a form. First, place the cursor on an existing line of text. Next, select the Edit pull-down menu, and then the Center option. The system automatically provides the required number of spaces to the left of the text line so the line is centered on the form.

Working with the Clipboard

The Fitrix *Screen* Form Painter offers a number of options for working with large, contiguous blocks of text. This part of the documentation explains the use of the Clipboard as well as the Mark, Cut, and Paste features.

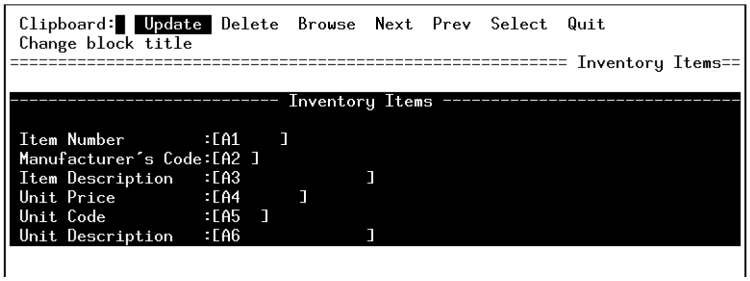
Using the Clipboard

The Form Painter allows you to mark any current block of text on the Form Editor and cut or copy it to the Clipboard where you can later retrieve and use it. Think of these individual blocks of text as Clipboard "pages." The discussion first focuses on how to use the pages on the Clipboard. Next, the focus turns to methods for adding pages to the Clipboard.

Data stored in the Clipboard by any given user is retained in the form of pages. The most recent 100 titled pages are preserved in a stack for later use. When you store a new page on the Clipboard, the oldest page is "pushed off" the end of the stack. The stack of titled clipboard pages is maintained for use in future editing sessions. In order for a Clipboard page to remain on the stack from one Form Painter session to the next, you must give it a title. The topic of titling pages is addressed later in this section.

The Clipboard is most effective as a time-saver when you perform repetitive tasks. Instead of typing several identical lines, you can type one line, copy it to the Clipboard, then paste it several times. You can also use the Clipboard to copy sections from existing data screens, then paste those sections into a new screen.

To display the Clipboard, select the Clipboard option on the Edit pull-down menu. The clipboard page at the top of the stack is displayed in a full-sized window on the editor form. The cursor appears on the Clipboard command line, allowing you to select from a number of Clipboard-related commands. The following diagram illustrates the general format.



```
Clipboard:  Update  Delete  Browse  Next  Prev  Select  Quit
Change block title
========================================================= Inventory Items==

--------------------------- Inventory Items ------------------------------

 Item Number        :[A1    ]
 Manufacturer´s Code:[A2 ]
 Item Description   :[A3            ]
 Unit Price         :[A4      ]
 Unit Code          :[A5  ]
 Unit Description   :[A6            ]
```

The Clipboard command line offers seven commands for use with the individual pages.

Think of clipboard pages as documents in a file; they can be titled (Update), viewed (Next, Prev, Browse), deleted (Delete), or selected for pasting into the current document (Select). The commands are outlined below:

Update: Name or rename the current page. You must title blocks of text (pages) you plan to use in the future. When you exit the Form Painter, all untitled Clipboard pages are automatically removed. When you select the Update command, the prompt line changes to the following:

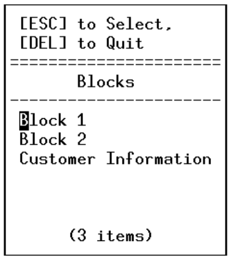
```
Enter title:
```

Enter the title you intend to give the text page currently shown in the clipboard window. Press [DEL] instead of [ESC] if you decide against storing the text page under the title specified at the prompt. The standard INFORMIX-defined edit keys are available. For more information on how the Form Painter Clipboard titles pages refer to the discussion of the Cut option on page 6-11.

Delete: Delete a page in the Clipboard stack.

Browse: View a list of pages. This command draws a Browse window over the form containing a list of pages currently stored on the clipboard. The list of items consists of all the titled and untitled pages.

The Browse window appears as follows:



```
[ESC] to Select,
[DEL] to Quit
=====================
       Blocks
---------------------
Block 1
Block 2
Customer Information



      (3 items)
```

Next: View the next page in the stack. When used at the bottom of the stack, the Next command loops around to display the first page in the current stack.

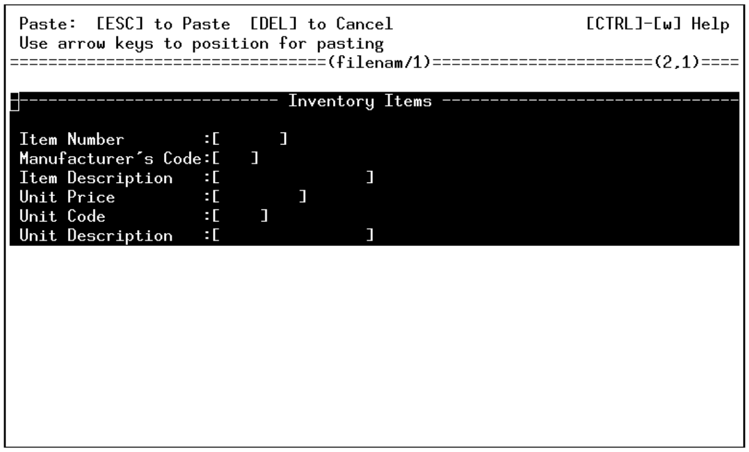
Prev: View the previous page in the stack. This command is the opposite of the Next command.

Selecting an item (page) in the Browse window makes that item current in the clipboard window. In order to paste the page into the current form, select it from the clipboard window with the Select command.

Select: Select the page for pasting into the current form. Once you select a page, it appears on the Form Editor at the cursor position from which the Clipboard was called up. The page text is displayed in reverse video, with the cursor located in the upper left corner of the page.

Quit: Quit the Clipboard and return to the Form Editor. Use this command to return to the Form Editor without pasting a page from the Clipboard.

The following illustration provides an indication of how a selected clipboard page might look on the Form Editor prior to pasting.



```
 Paste:  [ESC] to Paste  [DEL] to Cancel                [CTRL]-[w] Help
 Use arrow keys to position for pasting
===============================(filenam/1)======================(2,1)====

---------------------------- Inventory Items ------------------------------

 Item Number        :[       ]
 Manufacturer's Code:[   ]
 Item Description   :[                ]
 Unit Price         :[         ]
 Unit Code          :[    ]
 Unit Description   :[                ]
```

Notice that the Clipboard command line is replaced with the Paste command line at the top of the screen. Use the arrow keys to move the highlighted page of text into the desired position on the form. When it appears in the proper location, press [ENTER] or [ESC] to paste the page into the current form. Once the page has been pasted into the form, it becomes part of the form and can be edited like any other text. Clipboard pages can be re-used for forms that contain identical areas, such as detail lines.

The information on the text page erases existing text and fields on the form (characters and fields), if pasted on top of existing text or fields. Unless you want to delete existing definitions, it is not a good idea to position and paste clipboard text on top of existing text/fields.

Note

Some fields on the Clipboard page may not transfer to new forms. Newer versions of 4GL have data types that older versions don’t recognize. When you paste a page created in a newer 4GL version into an "older" form, a warning pops up to tell you that you can’t paste the incompatible data types into the form. See "Engine/4GL Compatibility" on page E-4.

If you decide not to paste selected text from the Clipboard, simply press [DEL] to return to the Form Editor.

Using the Paste Command

The Edit pull-down menu contains the Paste option, which is used to automatically paste the top (newest) page of text from the clipboard.

By using the Paste option, you do not have to use the Clipboard command line. The Paste option automatically selects the newest page of text from the Clipboard.

The Paste option is designed to be a time-saver for those occasions when a certain block of text appears repeatedly on the screen form. By simply marking and copying the text to the Clipboard, you can re-use it quickly and easily. The default Hot Key setting for the Paste option is [CTRL]-[p].

Note

The text pages stored on the Clipboard can consist of any block of text entered onto a Form Editor. This includes the definitions for fields included in the text block. When you cut or copy a block containing fields, the block retains the definitions specified in the original field.

Marking Text Blocks

The preceding part of the documentation explained how you can benefit by using text pages stored on the Clipboard. This part focuses on how to store pages of text blocks on the Clipboard. In short, there are two ways to store pages in the Clipboard: by cutting (Cut), and by copying (Mark).

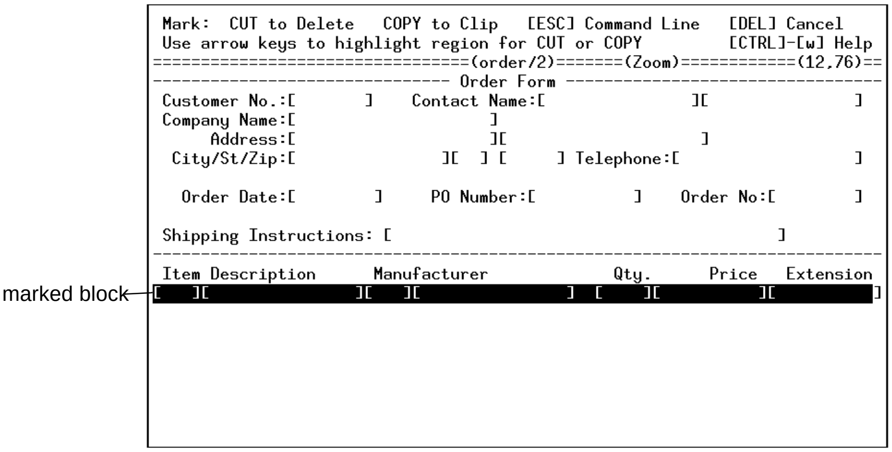
Both methods of storing text to the Clipboard rely on the Mark option, which lets you "pin down" the ranges of the text block. Once text is "marked," it is ready to be cut or copied to the Clipboard.

To mark a block:

1. **Position the cursor in one of the corners of the block to be marked.**
2. **Select Mark from the Edit pull-down menu or press [CTRL]-[v].**
3. **Stretch the highlight until it covers the entire area to be cut or copied.**

The Edit pull-down menu contains the Mark option. Mark can also be selected by pressing [CTRL]-[v]. The cursor must be positioned in a corner (upper left or lower right) of the text block before the Mark option is executed. This is important because once the Mark option has been executed, the cursor is "anchored." That is, the cursor cannot be moved from the anchor point; the only function of the cursor movement keys is to stretch the highlight to cover the block intended for storage on the Clipboard. The block can be stretched in a rectangular shape away from the anchor point; it can range in size from a single character to an entire form.

The following diagram depicts a marked block.



```
 Mark:  CUT to Delete   COPY to Clip   [ESC] Command Line   [DEL] Cancel
 Use arrow keys to highlight region for CUT or COPY         [CTRL]-[w] Help
=================================(order/2)=======(Zoom)============(12,76)==
------------------------------- Order Form ---------------------------------
 Customer No.:[       ]    Contact Name:[               ][               ]
 Company Name:[                    ]
      Address:[                    ][                    ]
  City/St/Zip:[               ][  ] [     ] Telephone:[                  ]

   Order Date:[        ]     PO Number:[          ]    Order No:[        ]

 Shipping Instructions: [                                        ]
----------------------------------------------------------------------------
 Item Description      Manufacturer             Qty.      Price   Extension
[   ][               ][   ][               ]  [    ][          ][          ]
```

In the previous example, the marked text block consisted of just one line. Prior to marking, the cursor could have been over the "C" in Cut, or over the right field delimiter. The text block, indicated by the highlighted area, can be expanded or contracted prior to cutting or copying.

Cutting Text Blocks

Once a mark has been placed on the form (anchoring the cursor), you can cut or copy the text into the Clipboard. In order to cut text into the Clipboard, you must execute the Cut option on the Edit pull-down menu or press [CTRL]-[t].

For example, assume you have marked an area of text you want to cut and retain in the clipboard. The next step is to execute the Cut option. You can do this by accessing the Edit pull-down menu and selecting Cut. This must be done while the text is still highlighted—if you execute the Cut option without having highlighted text on the Form Editor, there is no effect.

Once you execute the Cut option, the highlighted text on the Form Editor is moved from the Form Editor to the Clipboard.

Keep in mind that text pages stored in the Clipboard are titled generically until you deliberately title them through the Clipboard Update command discussed previously under "Using the Clipboard." The generic title is `Block n`, with n being a number incremented each time a new page is stored to the Clipboard. The most recent untitled text page is automatically titled `Block 1`. It is strongly suggested that a new text page be given a descriptive title as soon as it appears on the Clipboard—a Clipboard text page is not saved from one Form Painter session to the next unless it has been given a title. For instructions on titling Clipboard text pages, see the discussion of the Clipboard Update command on page 6-6.

Copying Text to the Clipboard

The second method for storing text pages to the Clipboard is by copying text. To copy text, you first have to mark the text to be copied with the Mark command ([CTRL]-[v]), then execute either the Copy or the Mark option after the block is highlighted. The Copy hot key is the same as the Mark key, [CTRL]-[v].

The primary steps are the same as those you used to cut text to the clipboard. On a current form, you must first use the Mark option to anchor the cursor at a corner of the text block you want to copy to the Clipboard. Then expand the highlight to encompass the desired text. Execute the Copy command or the Mark command a second time. Once a text block is copied, the highlight disappears and the demarcated text becomes the top page of the Clipboard stack.

As with text pages cut to the Clipboard, copied pages should be given descriptive titles as soon as possible.

Creating Detail Arrays

The copying and pasting features of the Form Painter make it easy to create detail field arrays in your forms.

To create a detail array (used in the detail section of a header/detail form):

1. **Make sure you are in input area 2.**

2. **Create the first line of detail fields.**

3. **Copy the detail line.**

4. **Paste the detail line multiple times.**

Field definitions are retained.

Copying Between Input Areas

When you cut or copy a field and then paste it back to the form, the pasted field becomes part of the current input area. All input areas must be defined before you copy fields to them.

To copy a field from input area 1 to input area 2:

1. **While in input area 1, copy the field.**

2. **Switch to input area 2.**

3. **Paste the new field where you want it.**

7

Form Definition

This chapter addresses the definition of individual fields and forms. The information in this section is based on the options found on the Define pull-down menu. The information in this chapter covers:

- Novice and Expert Modes
- Defining Form Defaults
- Defining the Input Area
- Defining Fields
- Defining the Cursor Path
- Defining Math for Fields
- Defining Lookups
- Defining a Zoom Field
- Defining Triggers
- Selecting Commands for the Ring Menu
- Short Cuts to Define Options
- Defining Copyright Text for Applications

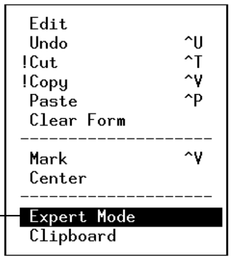
Novice and Expert Modes

You enter field definitions onto data-entry forms in the Form Painter. Each field on a definition form used to enter table name/column name information offers the Zoom feature as well as input validation. For example, any time you need to enter a join statement or a unique key, you can use the Zoom to select a valid entry.

The first aspect to consider when defining information is your operating mode. Two modes are available in the Form Painter: Novice and Expert. The system default is Expert mode.

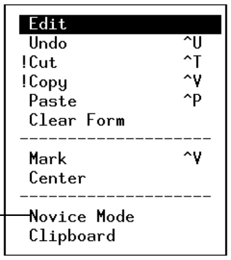
The Edit pull-down menu offers a toggling option that switches the mode under which you operate. The two modes, Novice and Expert, determine which type of forms you may create as well as the amount of detail you can define for each field.

When Novice mode is in effect, a toggling option on the Edit pull-down menu appears as follows:



This option is a toggle. By selecting the Expert Mode option, you will switch to expert mode. This also indicates that you are currently in novice mode

To change to expert mode, you need to select the Expert Mode option. When the expert mode is active, the option is displayed differently:

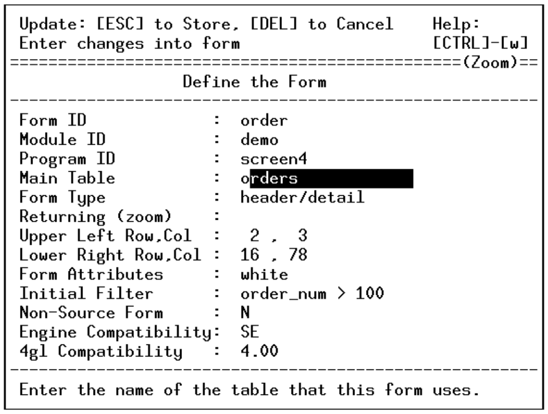


Selecting the Novice Mode option switches you to novice mode. This also indicates that you are currently in expert mode

If the Novice Mode option is visible, selecting that option makes the novice mode active. Creating forms in novice mode is easier because system defaults are used, which denies you some of the more complicated details associated with screen form specification. It can be used only for painting header-only forms (flat files). You cannot use the novice mode to build header/detail, zoom, or browse forms. Pop-up windows used to define fields and forms are less detailed when running under the Novice mode; not as many characteristics are available.

Defining Form Defaults

The next step in defining a data-entry application with the Form Painter is to establish form defaults. You have access to the Define the Form form through the Form Defaults option on the Define pull-down menu. Alternatively, you can access the form by pressing [CTRL]-[z] in the Form Editor when the cursor is anywhere but inside a defined field. Either way, you must first be working with a current screen form. The Define the Form form:



```
 Update: [ESC] to Store, [DEL] to Cancel    Help:
 Enter changes into form                    [CTRL]-[w]
=============================================(Zoom)==
                  Define the Form
-----------------------------------------------------
 Form ID            :  order
 Module ID          :  demo
 Program ID         :  screen4
 Main Table         :  orders
 Form Type          :  header/detail
 Returning (zoom)   :
 Upper Left Row,Col :   2 ,  3
 Lower Right Row,Col :  16 , 78
 Form Attributes    :  white
 Initial Filter     :  order_num > 100
 Non-Source Form    :  N
 Engine Compatibility:  SE
 4gl Compatibility  :  4.00
-----------------------------------------------------
 Enter the name of the table that this form uses.
```

A number of default values appear on the form automatically. The fields Form ID, Module ID, and Program ID are NOENTRY; they take their values from the form name and directory structure in which you are developing the screen form.

Form ID: The Form ID is established when the current form is first created, and contains the name of the form.

Module ID: The Module ID is derived from the parent directory name (less the .4gm extension).

Program ID: The Program ID name comes from the name of the present working directory (less the .4gs extension).

Main Table: The Main Table field is required if you use a browse or zoom form with a header, header/detail, or add-on header form. You can use the Zoom feature to display a picker list of existing table names in the default database. Validation occurs on this field although you are not required to enter an existing value.

Form Type: The Form Type field contains the type of form you selected after naming the form. You may change the type of form by entering a new type in this field.

This field offers the Zoom feature, available by pressing [CTRL]-[z]. The Zoom feature draws a picker list of valid screen types. For more information refer to "Form Types" on page 10-4.

Returning (zoom): This field is bypassed unless the form type is zoom. Since a zoom form can be used to return selected data, the field name for returned data must be listed here. The Zoom feature is available in this field to help you select a valid entry. Validation occurs on this field although you are not required to enter an existing value.

Upper Left Row, Col: These fields each contain two values that position the data-entry form on the terminal screen. The field labeled Upper Left Row, Col stores the beginning row and column number, respectively, for the screen form. The default values 2, 3 correspond to those typically assigned to a header or header/detail form. You will most likely change the values in these fields for browse and zoom screens, which tend to be displayed across a portion of the main program screen form.

Lower Right Row, Col: This field maps the lower row and ending column values. If specified, these values can "extend" the boundaries of the form beyond the column or row (given the limits on the form size). If left blank, the size of the form defaults to the lower row and right-most column entered onto the form. The benefit of specifying values for the Lower Right Row, Col field is that the form can be easily resized.

When creating browse or zoom screens, the Lower Right Row, Col field really has no effect. When you save your form, the right edge of the form defaults to the right-most character on the form. Therefore, in order to center your columns on your form, you may need to use a dashed line to determine the width.

Form Attributes: Default attributes can be assigned through the Form Attributes field. The default attributes (border, white) are consistent with code generated with the Fitrix *Screen* Code Generator.

Initial Filter: This field controls the initial selection of records from the database. The default filter 1=0 evaluates to false for each record, indicating that no automatic selection of records takes place. The filter 1=1 automatically selects all records since the program always evaluates the filter statement as true. The special words "all" and "ALL," when you enter them into the Initial Filter field, have the same significance as 1=1. The words "none" and "NONE" are interpreted in this field as 1=0.

The entry in this field is checked for syntactical correctness. If irregular syntax is detected, a warning prompt appears. The system does not, however, require that valid syntax be specified.

Non-Source Form: The Non-Source Form field is a yes/no field that determines whether or not the Code Generator generates source code off this form. If you answer Y, then the flag `non_source_form` is written to the first line following the copyright heading of the .per file. The non-source form statement prevents the Code Generator from generating code for this form. This allows you to have multiple `main .per` forms in working directories, which may be necessary when you generate code off of one particular form but you use another form for display purposes.

The last two fields allow you to choose the engine and 4GL compatibilities for the form. These fields circumvent incompatibilities between engine and 4GL versions. Certain data types available in the 4.10 4GLs run only on the OnLine engine.

For more information refer to "Engine/4GL Compatibility" on page E-4.

Engine Compatibility: The default engine is Standard Engine (SE). Zoom is available.

4GL Compatibility: The default 4GL version is 4.10. Zoom is available.

Defining the Input Area

The definition of input areas becomes important when you create a complicated screen form such as the header/detail form. When creating a header/detail application, you must define two distinct input areas, each with its own main table, unique key, filter, etc.

Note

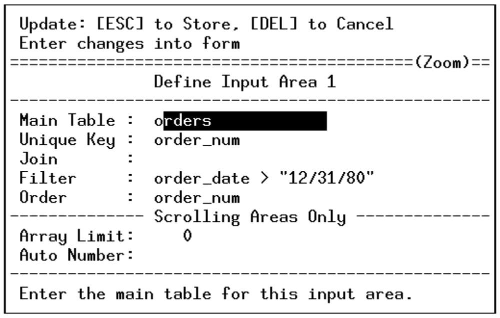
You must define input areas before creating, copying, or pasting fields.

This part of the documentation explores the characteristics that you define at the input area level. The discussion is based on the Input Areas option, found on the Define pull-down menu and on the picker list that appears when you press [CTRL]-[z] in the Form Editor. Since you must operate in Expert mode in order to develop a header/detail screen form, the discussion assumes that you are in expert mode. Later, the abbreviated Define Input Areas form used for Novice mode is discussed.

The input area is automatically determined by the main table you declare for the latest field you define. That is, you enter the table name for each field you define on your form. When the table name for the current field differs from that of the most recent field, a new input area is established. All subsequent fields sharing the new table name form part of a common input area. Typically, this lets you distinguish between the header and detail portions of the form. In the .per file, input areas are titled inputN, with N being a number.

You enter and modify some of the input area characteristics through the Define Input Area form. When you select the Input Areas option while working with a header/detail form, a picker list helps you determine the particular input area to define. After you select the desired input area, the Define Input Area form appears.

The Define Input Area form:



```
Update: [ESC] to Store, [DEL] to Cancel
Enter changes into form
=======================================(Zoom)==
              Define Input Area 1
-----------------------------------------------
Main Table :  orders
Unique Key :  order_num
Join       :
Filter     :  order_date > "12/31/80"
Order      :  order_num
-------------- Scrolling Areas Only -----------
Array Limit:     0
Auto Number:
-----------------------------------------------
Enter the main table for this input area.
```

Entry into the fields on this form is determined by the input area chosen and by the type of form you are developing (form type).

Main Table: This field stores an entry which defines the main table for this input area. There can be only one main table for each input area. The Zoom feature is available in this field to help you select a valid entry. Validation occurs on this field although you are not required to enter an existing value.

Unique Key: The Unique Key field stores a list of fields that comprises the unique key for the main table in the main (non-scrolling) section of the screen. The system uses this information to key "secondary" data to the main table. This secondary data includes Freeform Notes and User Defined Fields. If the key is not defined, you do not have access to certain User Control Library features such as Freeform Notes or User Defined fields.

The Unique Key field scrolls so the entry into this field can exceed the visible length shown on this form. The Zoom feature is available in this field to help you choose a valid entry. Validation occurs on this field although you are not required to enter an existing value.

Join: The Join field defines the criteria for joining the scrolling input area to the main input area. The Join field is only specified in the scrolling input area (input area 2); therefore, this field is not specified unless the form type is header/detail. The Zoom feature is available in this field to help you select a valid entry. Validation occurs on this field although you are not required to enter an existing value.

The following is an example:

```
items.order_num = orders.order_num
```

In this instance, the scrolling section main table (items) is joined to the main section main table (orders).

Filter: The Filter field stores the hard-coded filter used in every query. This filter is joined with your query by example filter and the filter that may be passed via the command line. The Zoom feature is available in this field to help you choose a valid entry. Validation occurs on this field although you are not required to enter an existing value.

This field’s default is 1=1 (no hardfilter). The special words "all" and "ALL", when entered into the Filter field, have the same significance as 1=1. If you enter the words "none" and "NONE" in this field, they are interpreted as 1=0.

For example, if you *only* wanted to see customers with a `customer_num` greater than 1000 in this program, you would specify in the Filter field:

```
customer_num > 1000
```

Order: The Order field stores the field names by which you wish to order the selection of documents. Enter the desired "order by" clause in this field. The "order by" clause can be made descending or ascending. The Zoom feature is available in this field to help you select a valid entry. Validation occurs on this field although you are not required to enter an existing value.

Like other variable-length fields, the Order field stretches to accommodate an entry that exceeds the visible length of the field on the Define Input Area form.

The last two fields appearing on the Define Input Areas form are accessible only when you define the detail input area. To enter these fields, you must define a header/detail screen form, and you must select Detail on the input area picker list.

Array Limit: The Array Limit field stores an integer value determining the number of internal program array elements you wish to provide space for in the scrolling input area. It only is used for detail and Zoom type input areas. The default value for this field is 100.

Auto Number: The Auto Number field specifies a detail field that the system applies unique line numbers to, for maintaining the order of detail rows. This allows each detail line to be unique. Any detail field specified in Auto Number identifies an item entered on the first detail line as "item on line number 1." Then, whenever the detail lines are re-displayed, they are displayed in the same order they were entered.

If you want to maintain the order of your detail lines you must specify an Auto Number field; if you do not, detail lines appear in unpredictable order. Do not list the line number field (that contains the actual line numbers) in the form image because it is not maintained during input. It is maintained only upon disk writes. There is no default value.

The Zoom feature is available in this field to help you select a valid entry. Validation occurs on this field although you are not required to enter an existing value.

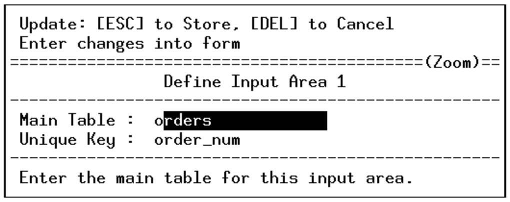
Switching Between Input Areas

When you work in the Form Painter you can toggle between input areas by pressing [CTRL]-[n]. Pressing [CTRL]-[n] moves the cursor to the first field in the next input area. The current input area appears at the top of the form next to the form name. The title `customer/1` indicates that you are working in input area one of the current form.

Defining the Input Area in Novice Mode

You can define the input area while operating in Novice mode, although the form used is limited to two fields: Main Table and Unique Key. While in Novice mode, you can only create header-only screen forms, so the additional fields described previously are not applicable.

The Novice Define Input Area form appears as follows:



```
 Update: [ESC] to Store, [DEL] to Cancel
 Enter changes into form
==========================================(Zoom)==
                Define Input Area 1
--------------------------------------------------
 Main Table :  orders
 Unique Key :  order_num
--------------------------------------------------
 Enter the main table for this input area.
```

The fields appearing on the form are discussed in "Defining the Input Area" on page 7-7.

Defining Fields

Fields on the Form Editor are delimited by square brackets. Square brackets ([ ]) have special significance when entered into the Form Painter. When the left bracket ([) is pressed while the cursor is on a blank space, the Field Definition form is displayed. The [ ] keys have no effect when entered on top of existing text. The right square bracket appears on the form automatically once a field has been defined.

You may resize any field by typing a] near an existing field delimiter. For example, if you want to expand the width of a field by two characters, move the cursor two characters to the right of the right bracket and type]. The old field delimiter disappears and the field is lengthened by two characters.

You can also shorten a field by moving the cursor in-between the field delimiters and pressing [CTRL]-[x]. This moves the right field delimiter one space to the left.

Sometimes you may not have enough room on your form to use square brackets as delimiters between two adjacent fields. In this instance you can use the pipe symbol "|" as a delimiter between fields. Say you have two 10 character fields that need to be located next to each other but you only have one space between them to specify a field delimiter:

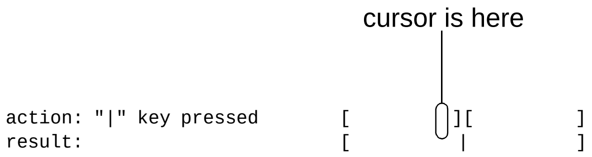
```
Customer Name: [          ][          ]
```

If the above does not fit, you could use the pipe delimiter like this:

```
Customer Name: [          |          ]
```

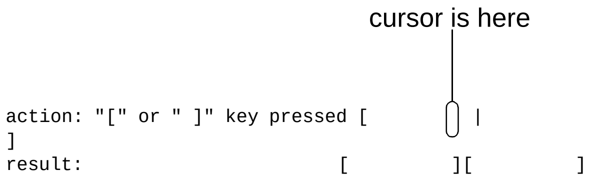
Adjacent brackets][can be converted to pipe "|" delimited fields by moving the cursor to either the ending] of the first field or the starting [of the second field and pressing the | key.

If the pipe (|) is pressed when the cursor rests on the] of the first field, the end] of the first field is converted to a pipe and second field is moved left with its starting [removed.



```
action: "|" key pressed       [        ][         ]
result:                       [         |         ]
```

You can also disconnect fields that use the pipe delimiter in the same way. With the cursor on the pipe, type a right or left square bracket "]" or "[".

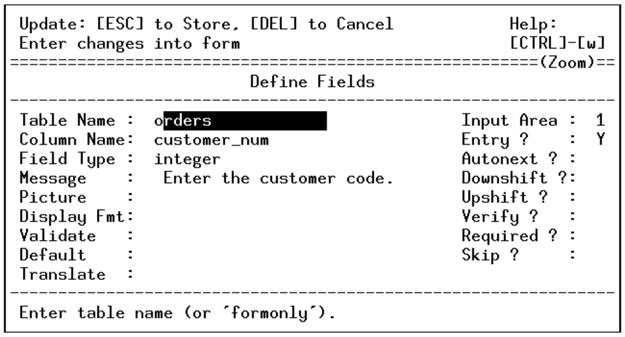


```
action: "[" or " ]" key pressed [       |
]
result:                       [         ][        ]
```

If there is not enough room to perform this operation an error message is displayed.

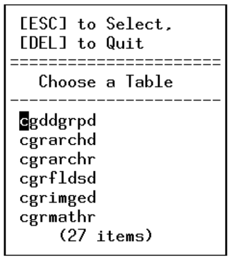
Defining Fields in Expert Mode

Expert mode, which is required for all screen form types other than header-only, provides you with additional control over the field definition. When the left square bracket ([) is pressed on a blank part of the Form Editor screen in Expert mode, the Expert Define Fields form appears.



```
Update: [ESC] to Store, [DEL] to Cancel             Help:
Enter changes into form                             [CTRL]-[w]
=======================================================(Zoom)==
                         Define Fields
---------------------------------------------------------------
Table Name :  orders                         Input Area :  1
Column Name:  customer_num                   Entry ?    :  Y
Field Type :  integer                        Autonext ? :
Message    :   Enter the customer code.      Downshift ?:
Picture    :                                 Upshift ?  :
Display Fmt:                                 Verify ?   :
Validate   :                                 Required ? :
Default    :                                 Skip ?     :
Translate  :
---------------------------------------------------------------
Enter table name (or 'formonly').
```

Table Name: This field identifies the table this field accesses. Pressing [CTRL]-[z] in the Table Name field displays a selection criteria prompt that allows you to narrow the list of available table names. After entering criteria and pressing [ENTER] the picker list appears as follows:

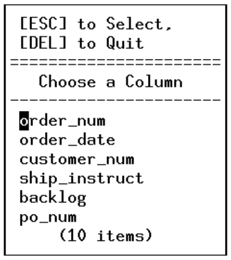


```
[ESC] to Select,
[DEL] to Quit
=====================
   Choose a Table
---------------------
cgddgrpd
cgrarchd
cgrarchr
cgrfldsd
cgrimged
cgrmathr
    (27 items)
```

As with other picker lists used throughout the system, pressing the [ESC] key selects the row on which the cursor appears. The [DEL] key returns control to the Define Fields pop-up window without returning any information. The INFORMIX-defined paging keys, [F3] (page down) and [F4] (page up), allow you to page through long lists.

Column Name: This field lets you specify the name of the column this field affects. The column name should be listed as part of the table you select in Table Name. Press [CTRL]-[z] to view a picker list of the columns found in the named table.

The picker list appears as follows:



```
[ESC] to Select,
[DEL] to Quit
======================
   Choose a Column
----------------------
order_num
order_date
customer_num
ship_instruct
backlog
po_num
    (10 items)
```

The Zoom on Column Name only operates if you have specified a valid table name for this field. Validation occurs on these fields although you are not required to enter an existing value.

Input Area: This field determines which input area the field belongs to. The input area is defined by the main table. For more information on input areas refer to "Defining the Input Area" on page 7-7.

Entry: This is a yes/no field that determines whether the field is to be NOENTRY. If you restrict entry for a field, you establish it as display-only. Display-only fields are generally used to return information by a lookup.

If you indicate that the field is NOENTRY (N), the cursor does not enter the field on the compiled data-entry form. When Entry? is set to N, the Required? and Message fields do not need to be specified.

Field Type: Field type is automatically displayed based on the table and column description for the field. If Table Name is specified as formonly, you are able to specify the field type.

There are two field types that allow you to enter text files, such as a spread sheet, or byte files, such as graphics, sound, or video clips. These field types are called BLOBs (Binary Large OBjects).

If you try to create a BLOB field and your engine or 4GL are not compatible, a warning message appears. For more information refer to "Creating BLOB Fields" on page 7-18.

If the field type is defined "like" an existing table.column in the current database, the Zoom feature can be used to select a valid table.column entry. Validation occurs on this field although you are not required to enter an existing value.

Message: This field stores a comment or description line that appears on the compiled program whenever the cursor enters the field being defined. When a user runs the application created from your painted form and enters a field, message text defined for that field appears at the bottom of the form. Also, whenever your cursor enters a field while you are in the Form Editor, text from the Message field is displayed at the foot of the Form Editor. Message text must be limited to 74 characters.

Picture: This field can be used to establish a character pattern determining how data-entry on the compiled form appears. Perhaps the most obvious usage of this attribute is to format the input of telephone numbers. An example for area code and phone number:

```
(###) ### - ####
```

As you enter digits in the compiled form, they appear in place of the pound signs (#), which serve as placeholders. The pound sign is used for numeric entry; the character A is used for alphabetic characters; the character X is used for alphanumeric entries. Do not "quote" your pattern in the Picture field attribute.

Display Fmt: This field serves as a hybrid attribute for the INFORMIX attributes FORMAT and DISPLAY LIKE. These attributes are mutually exclusive; you can use the entry in Display Fmt for either of the two INFORMIX attributes. Examples of proper entries:

```
mm/dd/yy
###-##-####
like stxcntrc.company
```

If the display format is defined "like" an existing table.column in the current database, you can use the Zoom feature to select a valid table.column entry. Validation occurs on this field although you are not required to enter an existing value.

Validate: This field is similar to Display Fmt in that it covers two INFORMIX attributes that are mutually exclusive. In this case, the attributes are INCLUDE and VALIDATE LIKE. Validate stores INCLUDE information unless the first word specified is "like" (in which case, the field attribute behaves as if it were a VALIDATE LIKE entry). The following are examples of proper entries:

```
"Y", "N"
1 to 50, 200 to 422
like customer.fname
```

If the entry is defined as "like" an existing table.column in the current database, you can use the Zoom feature to select a valid table.column entry. Validation occurs on this field although you are not required to enter an existing value.

Default: This field enables you to enter any data that appears in a field by default when the program is run. A user running the program could then press [ENTER] to accept the default value or to enter new data. When specifying default data for character fields you must surround your data with quotes (" ").

Notes about Default:

1. Defaults are limited to 30 characters in length. The default line can contain many default values for fields, with each default value having a maximum length of 30 characters.
2. Defaulting is not performed in input area 1, the header, unless all of the variables in the input 1 program `p_record` are null.
3. Defaulting is not performed for a specific row in the detail input array unless all the program `p_record` variables for a given row are null.

You can easily create a field that defaults to the current date. Simply put "today" in the default field. The "today" keyword works with date columns only.

Translate: This field lets you enter translation context. If you are using language translation you must first set up your translation contexts by populating the `stxlangr` table. Anything you enter in this field must be defined in the `stxlangr` table. For more information on language translation refer to "Translating Values Used in Data Entry" on page 17-7.

Autonext: This field determines whether the cursor automatically transfers to the next field on the compiled data-entry form when the current field is full. If the entry in Autonext? is Y, the cursor automatically jumps to the next field when the current field is filled. This is useful for fields typically filled with a constant number of characters, such as department codes.

Downshift: This field converts uppercase characters to lowercase upon display. A Y (Yes) entry converts all uppercase data-entry characters into lowercase.

Upshift: This field is the opposite of Downshift?; it converts lowercase characters into uppercase for screen display.

Verify: This field is available as a means by which data-entry accuracy can be enhanced. If this field stores the value Y (Yes), end-users are required to make an identical entry into the defined field twice.

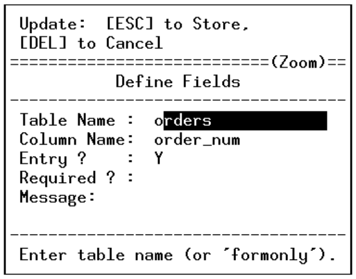
Required: If the current field is enterable, you can designate whether this field is required. When the compiled application is run, required fields must be filled with a valid entry before the document can be saved. The Required? field stores a value of either Y or N. Detail fields cannot be made required.

Skip: A Y in this field causes skip logic to be generated for this field. Refer to "Creating Skip Field Logic" on page 15-36.

Defining Fields in Novice Mode

Novice mode displays an abbreviated field definition form shown in the following illustration. Since the Novice mode can be used only to build header-only screen forms, the required information is limited. If you operate in Expert mode, please see the previous section "Defining Fields in Expert Mode" on page 7-13.

In Novice mode, the Define Fields form appears as follows:



```
Update:  [ESC] to Store,
[DEL] to Cancel
==========================(Zoom)==
           Define Fields
----------------------------------
Table Name :  orders
Column Name:  order_num
Entry ?    :  Y
Required ? :
Message:

----------------------------------
Enter table name (or 'formonly').
```

For an explanation of these fields see the previous section.

Modifying Existing Field Definitions

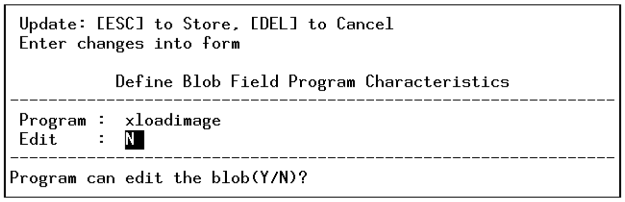
The preceding information explains how to define fields. You must use the left square bracket ([) to initially define a field. To modify an existing field definition, you must first place the cursor in the field (the entire field appears in reverse video). Next, execute the Fields option on the Define pull-down menu, or press [CTRL]-[z] from within the Form Editor and select Fields from the picker list.

After selecting this option, the Define Fields form appears. You can then modify any attribute you wish. The appearance of the Define Fields form depends, of course, on the mode in which you are operating.

Creating BLOB Fields

If you are running INFORMIX-4GL 4.10 or higher and the OnLine engine, you can create an application that uses BLOB (Binary Large Object) technology. A BLOB can be a text file, a graphics file, a sound file, or another application. For more information on BLOBs refer to "Creating BLOBs" on page 15-31.

To create a field that uses BLOBs (Binary Large Objects), you must select a column that has been set up as a "text" or "byte" field. Once you specify a text or byte column in the Column Name field on the Define Fields form a pop-up window appears into which you enter the program and edit permission for the BLOB.



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

          Define Blob Field Program Characteristics
---------------------------------------------------------------
 Program :  xloadimage
 Edit    :  N
---------------------------------------------------------------
Program can edit the blob(Y/N)?
```

The following lists the default values:

For a text field:

```
Program: vi
Edit   : Y
```

For a byte field:

```
Program: xloadimage
Edit   : N
```

When you specify BLOBs and compile the form, the .per file lists the BLOB fields in the attributes section. If the BLOBs are columns in a table, the input 1 section of the .per contains one or more blobdef lines—one per BLOB. The blobdef lists the column name, the specified program, and the edit permission.

An example: `sp_sheet` is the column name of a text BLOB. Wingz is the program name and the permission to edit is set to Y. After you save the form, the blobdef line appears as follows:

```
blobdef - sp_sheet, Wingz, y
```

When running the final application created with a form that uses a BLOB field, if the BLOB field on the screen is blank, the field is empty. If the field on the screen contains an asterisk, there is data in the BLOB field. Zoom on the asterisk to view (or run) the BLOB file.

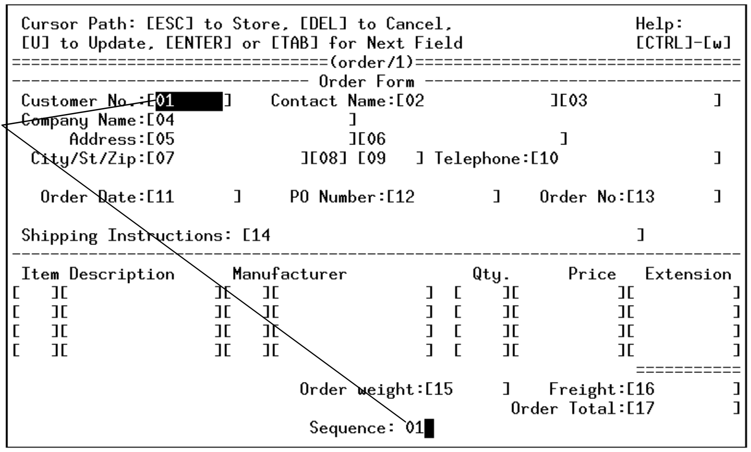
More information on creating BLOBs can be found in "Creating BLOBs" on page 15-31.

There are several other field types in addition to BLOBs that depend on engine and 4GL compatibility. Refer to "Engine/4GL Compatibility" on page E-4 for more information.

Defining the Cursor Path

You may choose to have the cursor proceed from field to field in an order different from the order in which the fields on the screen form were defined. The sequence can be modified quickly and easily with the Cursor Path option on the Define pull-down menu. Alternatively, you can access the Cursor Path option by pressing [CTRL]-[z] while in the Form Editor, outside a defined field.

Once you select the Cursor Path option, a picker list appears allowing you to determine the input area you wish to work with. Once you select an input area, the field tags for the defined fields in the input area appear. The field tags are automatically numbered according to the order in which the fields were specified. The cursor path for the header section of a screen form might appear as follows:



These numbers represent the order of the field in the cursor path

```
 Cursor Path: [ESC] to Store, [DEL] to Cancel,                  Help:
 [U] to Update, [ENTER] or [TAB] for Next Field                 [CTRL]-[w]
=================================(order/1)==================================
------------------------------- Order Form ---------------------------------
 Customer No.:[01     ]    Contact Name:[02             ][03             ]
 Company Name:[04                  ]
      Address:[05                  ][06                  ]
  City/St/Zip:[07             ][08] [09   ] Telephone:[10                ]

   Order Date:[11      ]     PO Number:[12        ]    Order No:[13      ]

 Shipping Instructions: [14                                      ]
----------------------------------------------------------------------------
 Item Description      Manufacturer             Qty.      Price   Extension
[   ][               ][   ][               ]  [    ][          ][          ]
[   ][               ][   ][               ]  [    ][          ][          ]
[   ][               ][   ][               ]  [    ][          ][          ]
[   ][               ][   ][               ]  [    ][          ][          ]
                                                                 ===========
                              Order weight:[15     ]    Freight:[16        ]
                                                    Order Total:[17        ]
                               Sequence: 01
```

The previous example consists of a header/detail form, with the cursor path displayed for the header input area. The detail input area cannot be changed at the same time; the cursor path is modified one input area at a time. For this reason, the field tags for the detail input area fields are not displayed in the previous example.

The Cursor Path option changes the command line prompt to provide information about updating the existing cursor path. Press U to update the cursor path for the current screen form field. The order in which the cursor moves depends on the order of the field tags specified for the fields in the input area. Therefore, simply enter a new value for fields that you intend to reorder. The field tags are modified one at a time.

Defining Math for Fields

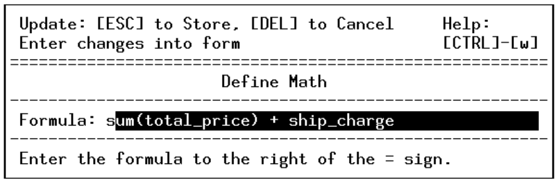
The Fitrix *Screen* Code Generator can automatically generate the logic for mathematical calculations in fields. To take advantage of this capability, you must define the calculation for the field by using the Math option on the Define pull-down menu.

To define math for a field:

1. **Define the field.**
2. **Move the cursor into the field on the Form Editor.**

   The entire field should be in reverse video.
3. **Select the Math option on the Define pull-down menu (or on the picker list appearing after you press [CTRL]-[z] while you are in a defined field).**
4. **Enter the math formula.**

The Define Math form appears on top of the existing data-entry form as follows:



```
Update: [ESC] to Store, [DEL] to Cancel     Help:
Enter changes into form                     [CTRL]-[w]
=======================================================
                     Define Math
-------------------------------------------------------
Formula: sum(total_price) + ship_charge
-------------------------------------------------------
Enter the formula to the right of the = sign.
```

The Define Math form contains only one field, the Formula field. This field stores the required mathematical equation. Entries can be longer than the visible length of the Formula field; the field scrolls to accommodate longer equations.

Sample entry for the Formula field:

```
sum(total_price) + ship_charge
```

Defining Lookups

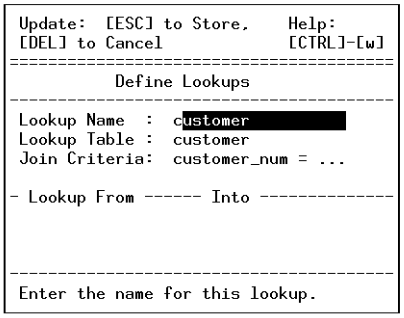
Lookups evaluate the data that is entered into a field and match that data against data in any table. A lookup placed on a field serves two purposes: verification and data retrieval.

See "Introduction to Lookups" on page 3-17 for illustrations of the two types of lookups.

The Define Lookups Form

The Define Lookups form lets you create lookups for fields on your .per form. The Define Lookups form is displayed by selecting a field in the Form Editor and then choosing Lookups from the Define pull-down menu. You can also display this form by pressing [CTRL]-[z] in a field and then selecting Lookups from the Define Fields picker.

The Define Lookups form appears as follows:



```
 Update:  [ESC] to Store,    Help:
 [DEL] to Cancel             [CTRL]-[w]
========================================
           Define Lookups
----------------------------------------
 Lookup Name  :  customer
 Lookup Table :  customer
 Join Criteria:  customer_num = ...

- Lookup From ------ Into --------------



----------------------------------------
 Enter the name for this lookup.
```

Lookup Name: The Lookup Name field stores the name of the lookup. Uniquely naming lookups allows you to have multiple lookups for the same field. An entry in the Lookup Name field is required, and must contain a unique name. Generally, the Lookup Name contains the name of the lookup table except when multiple lookups are performed on the same table; then you must use a unique name for each lookup.

Lookup Table: The Lookup Table field stores the table that is being looked up. That is, this field contains the table name storing the looked-up values. The Zoom feature helps you select a valid table name from the current database.

Join Criteria: The Join Criteria field lets you enter the "where" clause for the join statement used in the lookup. This field scrolls to accommodate entries larger than the visible size of the field. The Zoom feature can be used in this field to select valid column names; the Zoom appends selected data to the existing data in this field. Validation occurs on this field although you are not required to enter an existing value. For example:

```
stock_num = $stock_num
```

In this instance, the where clause matches the row in the stock table where the `stock_num` column value is equal to the `stock_num` field value on the data-entry form. Variables appear on the right side of the equals sign.

Lookup From/Into: If you do not enter anything into these fields, then the Code Generator automatically returns looked up data into all NO-ENTRY fields that belong to the same table as the field that was looked up.

You must fill in the Lookup From and Into fields when either of the following items are true:

1. The field name on the screen has a different name from the column in the table that you are looking up.

2. There is more than one lookup to the same table.

The Lookup From column contains the name of the column in the table being looked up, while the Lookup Into column contains the name of the field on the form.

Use the [TAB] key to enter the Lookup From/Into section. The [TAB] key also lets you exit this section.

The Zoom feature is available in each column of this section to help you select valid column names based on the lookup table specified. Entries in this section must correspond to column.table names defined on this screen form and in the lookup table. This is a scrolling set of columns; you may enter up to 50 destination column names.

The following example demonstrates a lookup in the stock.stock_num column and returns a corresponding description into the description column on the screen. This is how this lookup would appear in the .per file.

```
lookup  = name=stock_num, key=stock_num, table=stock,
            filter=stock_num = $stock_num, into=description
```

Creating a Data Retrieval Lookup

Data retrieval lookups are keyed from a field in the main table for this input, and they retrieve information from another table to place into destination fields. Destination fields must be NOENTRY type (Entry? set to N). The default destination (Lookup Into) fields are all NOENTRY fields in the input area that have a table name matching the lookup table name.

Lookups are defined through the Lookup option found on the Define pull-down menu (also found on the picker list appearing when you press [CTRL]-[z] from within a defined field). The lookup is defined in the key field, not in the fields receiving data returned by the lookup.

To define a data retrieval lookup:

1. **Create the key lookup field.**

   The key field is the field that triggers the lookup. For example, if you specify the Customer Code field as the key field, then whenever a value is entered here, the lookup occurs.

2. **Display the Define Lookup definition form.**

3. **Name the Lookup.**

   Lookups are generally named after the table being looked up.

4. **Enter the Lookup table name.**

   This is the name of the table the lookup is querying.

5. **Enter the where clause for the lookup key.**

   This is the join statement that matches the data in the lookup field to data in the database.

6. **Enter the column to be looked up from and the field to be returned into.**

   You only need to complete this step if the name of the column being looked up differs from the name of the column on the screen.

7. **Save the Lookup form by pressing [ESC].**

8. **Create each destination field that will display lookup information, defining each of them as NOENTRY.**

Creating a Data Validation Lookup

Typical verification lookups simply check the value entered by the user and do an SQL query on a table for it. If the value is not found, the standard error message "Value is not in the list of valid data" appears and the user is kept in the field until a value that is in the table is entered.

Data validation lookups differ from data retrieval lookups in that validation lookups don’t return information into any fields. Therefore when setting up a validation lookup, you don’t specify an Into field. If an Into field is not specified and no field on the screen shares the same table as the lookup field, information is not returned to any field.

The following example uses the customer entry form in Fitrix *Screen* demo 1 (`$fg/codegen/demo.4gm/screen1.4gs`), a header-only screen for entry into the customer table of the `stores` database. On this screen there is a field for the state code "State:" but no field for the state description. This example adds a verification lookup to ensure that the state code entered is a valid value in the state table of the `stores` database.

To create a data validation lookup:

1. **Enter the field you want to place the validation lookup on.**

2. **Call up the Define Lookups form.**

3. **Give the lookup a descriptive name, like `state_lk`.**

   The Lookup Table is the table checked for a valid value.

4. **Enter the join criteria to match the column in the table with the field on the screen.**

For example:

```
state.code = $state
```

Where `state.code` is the column in the table and `$state` represents the field on the screen. In the resulting program, the value entered by the user is matched against the column values in the state table.

When a user runs the program and enters a state value that does not exist in the state table, the "Value is not in the list of valid data" message appears.

Refer to "Error Handling Functions (fg_err and lib_error)" on page 15-44 for information on creating custom error messages.

Deleting Lookups

Lookups can be deleted by calling up the lookup definition form for the unwanted lookup and pressing [CTRL]-[d] to delete the lookup name. When you press [ENTER], you are prompted to verify whether you want this lookup deleted. Answering Yes deletes the lookup.

Defining Multiple Lookups

If you call up the lookup form on a field that has already had a lookup defined, the Select a Lookup Name form appears displaying the name of all lookups defined for that field, and an option for Add a Lookup. The Add a Lookup option allows you to define multiple lookups per field.

Each lookup must have a unique name.

Lookup Dependencies

Lookups must appear in the .per file in the order they are needed. If a lookup depends upon another, you need to list the lookups in the .per form in the order that they will be performed.

Examples of Lookup Usage

The following describes three examples of lookup usage:

1. If there is no `into` statement, the Code Generator searches the screen record for definitions of the same table as the table name of `table=tablename`.

   ```
   screen record s_pvendr (stpvendr.vend_code, stpvendr.bus_name,
   stpvendr.terms_code, stptermr.terms_desc)
        ...
        ...
        lookup = name=term_lookup, key=terms_code,table=stptermr,
        filter= stptermr.terms_code = $terms_code
   ```

   The Code Generator will find `stptermr.terms_desc` in the screen record therefore defaulting the `into=terms_desc`.

   If the Code Generator cannot find an associated table, then the lookup is defined as a validation only lookup (a lookup that returns no data).

2. If you use the `into` statement, all `into` statement's must be specific. You cannot use the `into` statement for some fields and expect the Code Generator to default the other ones.

   The `into=column` must be a column in the lookup table. It does not have to be a screen record field. If your screen record field has the same name as the column then the lookup returns data into that field otherwise it puts that data into a parallel record.

   ```
   screen record s_acct (stpinvce.acct_no, formonly.acct_desc)
        ...
        ...
        lookup = name=acct_lookup, key=acct_no, table=stxchrtr,
        into=acct_desc, into=incr_with_crdt,
        filter= stxchrtr.acct_no = $acct_no
   ```

   The Code Generator puts the `acct_desc` into `p_pinvce.acct_desc` and `incr_with_crdt` into `q_pince.incr_with_crdt`.

   The `p_` record is associated with the screen and the `q_` records are parallel to the `p_` records.

3. If you want to assign a lookup where the column selected is not the same name as the field you want to put it into, you can use the `from_into` syntax.

   ```
   screen record s_acct (stpinvce.acct_no, formonly.james_desc)
   ```

```
...
...
lookup = name=acct_lookup, key=acct_no, table=stxchrtr,
from_into=acct_desc james_desc, from_into=incr_with_crdt
          is_it_a_credit,
filter= stxchrtr.acct_no = $acct_no
```

The Code Generator puts the `acct_desc` `into p_pinvce.james_desc` and `incr_with_crdt` into `q_pince.is_it_a_credit`.

The `lookup()` Function

Lookups are initiated by a change in data of a field. In `llh_a_field` (the function that gets called after every entry field to perform after-field logic), the function `llh_lookup` is called as follows:

```
# After data_changed logic
if data_changed
then
    case
      when scr_fld = "state"
        # Perform Lookups
        if llh_lookup("state_lk",true) = false and
          length(this_data) != 0
        then
            let nxt_fld = "state"
            return
        end if
    end case
end if
```

The first argument to `llh_lookup` is the lookup name. The second argument is a "must find" condition, indicating what to do if the value entered is not found. `llh_lookup` behaves as follows if "true" or "false" is the second argument:

true: call the standard error message "Value is not in the list of valid data."

false: continue on with no call to an error message.

Fitrix *Screen* creates the second argument as "true" by default. The function `llh_lookup` itself returns "false" if the value is not found. Thus we have specific logic that is performed when `llh_lookup` returns a "false" condition:

```
if llh_lookup("state_lk",true) = false and
  length(this_data) != 0     #   ^^^^^ returns false condition
then                        # <------ so this logic is performed
    let nxt_fld = "state"
```

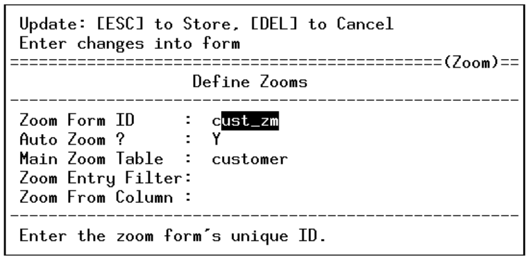
By default, Fitrix *Screen* sets the variable `nxt_fld` to the current field name to leave the user in the field until a valid value is entered.

Defining a Zoom

Zooms allow users of an application to call up a form to display a list of valid data for a field. The user can then select an entry from the zoom form which is then automatically returned to the original field. The Zoom function is invoked when running the compiled application by pressing [CTRL]-[z] when in a field with Zoom functionality.

The Define Zoom Form

If you intend to add Zoom functionality to the application, you need to specify the relationship between the current screen form and the zoom form. The Form Painter uses the Define Zooms form to prompt for information concerning the Zoom from a given field. The Define Zooms form is accessed through the Define pull-down menu (also found on the picker list appearing when you press [CTRL]-[z] from within a defined field). The Define Zooms form:



```
 Update: [ESC] to Store, [DEL] to Cancel
 Enter changes into form
=============================================(Zoom)==
                   Define Zooms
----------------------------------------------------
 Zoom Form ID      :  cust_zm
 Auto Zoom ?       :  Y
 Main Zoom Table   :  customer
 Zoom Entry Filter:
 Zoom From Column :
----------------------------------------------------
 Enter the zoom form´s unique ID.
```

Zoom Screen ID: This field stores the name of the zoom screen form without the .per extension. As with all screen form names, the name should be limited to seven characters. The Zoom feature is available to help select an existing form.

Auto Zoom: This field is a Y/N field that determines whether the Code Generator creates AutoZoom code for this field. When running the compiled application, if you place an asterisk anywhere in a field with AutoZoom capability, the Zoom logic uses that data to build the matches clause, then automatically calls the zoom screen. For example, if you enter an "s*" into an AutoZoom field, a zoom form automatically displays listing all of the values that begin with an s.

Do not specify AutoZoom if you need to have an asterisk as a piece of data in a Zoom key field, or if the Zoom key field is a non-char (numeric) type field that cannot use the MATCHES clause in an SQL query.

Main Zoom Table: This field lets you specify the main table for the Zoom. It is required only if you wish to have AutoZoom. The system requires it to build the MATCHES clause for AutoZoom.

Zoom Entry Filter: This field stores the "where" clause used when you enter the Zoom. By specifying a filter, you can determine what is displayed when the user executes the Zoom function. If a Zoom filter is specified, it uses that filter instead of placing the user into the query by example screen. The Zoom feature can be used in this field to select valid column names for the filter; the Zoom appends selected data to existing data (if any) in this field.

The special words "all" and "ALL," when entered into the Zoom Entry Filter field, have the same significance as 1=1 (select all records). The words "none" and "NONE" are interpreted in this field as 1=0 (select no records).

Validation occurs on this field although you are not required to enter an existing value.

Note that the initial filter specified here is used only upon the initial display. Once the user uses the Find command on the zoom, the initial filter is lost. You can set up a permanent zoom filter that stays even when the user selects the Find command. For information refer to "Creating a Permanent Zoom Filter" on page 10-62.

Zoom From Column: This field is required when the following items are true:

1. The table.column name being Zoomed into is different from the name of the column on the screen.
2. The screen field you are Zooming from is a character field.
3. Auto-Zoom is enabled. Zooms with the "noautozoom" keyword or a filter do not require the "from" keyword.

For example, if the field on your screen is named `customer_num` and the field being Zoomed into is called `cus_num`, you would enter `cus_num` into the Zoom From Column field.

Creating a Zoom

To attach a Zoom function to a field:

1. **Create the field to utilize Zoom functionality.**
2. **Display the Define Zooms definition form.**
3. **Enter the Zoom Form ID which is the name of the zoom .per form that this Zoom function calls.**
4. **Specify whether the field will allow the Auto-Zoom function.**
5. **Enter the Main Zoom Table name.**
6. **Enter the selection criteria used for the initial selection in the Zoom Entry Filter.**
7. **If table.column name being Zoomed into is different from the field name on the form, specify it in Zoom From column.**
8. **Store this Zoom function by pressing [ESC].**
9. **Create the Zoom .per form with the Form Painter.**
10. **If data is to be returned to the field on the main form, specify the returning column on the Define the Form form for the Zoom.**

Deleting a Zoom

You can delete a Zoom by calling up the Zoom Definition form for the field you want to delete, then blanking out the Zoom Form ID field. After removing the entry in the Zoom Form ID field either with [CTRL]-[d] or by pressing [SPACEBAR] until the field is erased. Pressing [ESC] displays a prompt warning you that the Zoom will be deleted. The Zoom for this field is deleted if you answer "yes" to this prompt.

Defining Triggers

The Form Painter permits you to add triggers as you are developing your data entry screen. Triggers allow you to localize form specific custom 4GL code modifications and additions in a single file, known as a triggers file. When you run the Code Generator on a form, the instructions placed in the triggers file are interpreted and incorporated into the resulting source code. Two benefits of triggers are:

1. You can make powerful and fast modifications to the resulting program without having to be intimately familiar with the source code.
2. You can add to and re-generate programs without losing any of your changes.

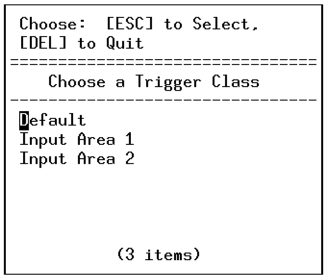
To add triggers with the Form Painter, select a trigger from a list of available triggers, then enter the program code for the trigger into a trigger definition form. When you save your form, a triggers file is created with your form name plus the .trg extension. When you run the Code Generator on your form, the triggers file is parsed, and the resulting generated program contains all the custom modifications you created with your triggers.

You can select the triggers feature from several different locations within the Form Painter. When you select the Triggers function, your location determines what type of trigger selection list will appear.

The Triggers function can be accessed by selecting the Triggers option from each of the following places:

Selecting Triggers from this location	**Displays the following form**
Define pull-down menu	Choose a Trigger Class
Define Form picker	Choose a Trigger Class
Define Field picker	Choose a Trigger

The Choose a Trigger Class form appears if you call up Triggers while the cursor is in the Form Editor and *not* in a field or from the Define pull down.



```
Choose:  [ESC] to Select,
[DEL] to Quit
================================
    Choose a Trigger Class
--------------------------------
Default
Input Area 1
Input Area 2




           (3 items)
```

This form allows you to select which class of triggers you want to work with.

- Default triggers handle custom modifications to `main.4gl`
- Input Area 1 triggers manage modifications to `header.4gl`.
- Input Area 2 triggers manage modifications to `detail.4gl`.

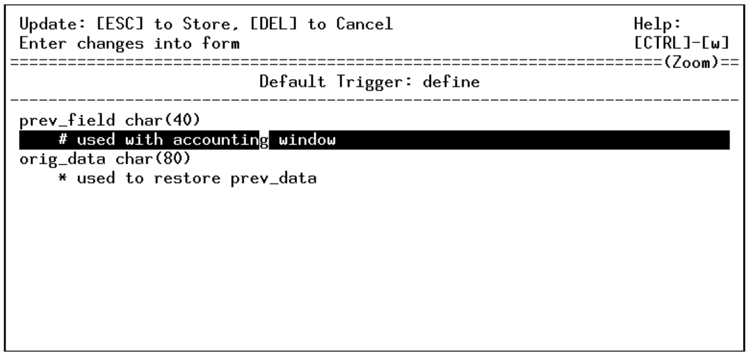
For more information on trigger classes, refer to "The Trigger File" on page 12-4.

After selecting a trigger class, the Choose a Trigger list appears. The Choose a Trigger list displays all the available triggers for the selected class. You can select a trigger by moving to the trigger and pressing [ESC] or [ENTER]. Selecting one of these triggers displays an entry form into which you may enter the code for that trigger.

Note

When you call the Triggers function from within a field, the Choose a Trigger form displays only those triggers that are available from within the current input area.

A Trigger Definition form appears as follows:



```
Update: [ESC] to Store, [DEL] to Cancel                            Help:
Enter changes into form                                            [CTRL]-[w]
=====================================================================(Zoom)==
                           Default Trigger: define
-----------------------------------------------------------------------------
prev_field char(40)
    # used with accounting window
orig_data char(80)
    * used to restore prev_data
```

Into this form you may enter 4GL code for the particular trigger. When entering triggers through the Form Painter you do not need to type the name of the trigger. The trigger name is automatically written to the triggers file. Also, a semicolon is automatically appended to the end of each trigger definition when the triggers file is created, so you do not need to put semicolons in.

A trigger can be deleted by deleting all the lines in the trigger with [CTRL]-[d].

For more information about each specific trigger refer to "The Triggers" on page 12-8.

Editing the Trigger

While you are in the Trigger Definition form, you can press [CTRL]-[z] to Zoom into your favorite editor such as vi. The Trigger Zoom adds to the flexibility of the Form Painter, allowing you to create triggers in the environment you are most comfortable with. Pressing [CTRL]-[z] in the Trigger Definition form puts you in a temporary file containing whatever was in the Trigger Definition form. If the form was empty when you pressed [CTRL]-[z], you will see an empty file. In the temporary file, create your trigger. When you are finished, write and quit like you would normally quit your editor. If you are using vi then you would perform a ":wq." Upon quitting the editor, you will return to the Trigger Definition form. Whatever you entered into your temp file will appear in the Trigger Definition form.

Saving the Trigger File

The Form pull-down menu provides an option to Save Trg File. This option allows you to save your triggers to a file named after your current form with a .trg extension. The Form Painter creates two files: a .per file and a .trg file. When you save the form with the Save Form option, a .per file is created along with a .trg file if you have defined any triggers. The Save Trg File option is useful when you open a form and only modify the triggers. This way you only have to save the part of the form that has been modified.

Deleting the Trigger File

The Delete Trg File option allows you to delete any trigger file in the current directory. Deleting a trigger file is convenient when you need to heavily modify a form. Rather than changing each individual trigger, you can delete the trigger file and rebuild them from scratch. The Delete Trg File option displays a picker list containing all trigger files in the current directory. Select the file you wish to delete.

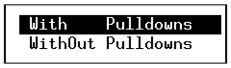
Recovering a Deleted Trigger File

If you use the Delete Trigger File option in the Form Painter, you may be able to recover an accidently deleted file. Instead of removing a trigger file with the `rm` command, the Delete Trigger File option copies the trigger file to the `/tmp` directory. If you need to recover a deleted file, all you need to do is retrieve the file from the `/tmp` directory.

Selecting Commands for the Ring Menu

The Select Commands option on the Define pull-down menu modifies the ring menu of the application being created. The Select Commands option may only be selected if a header or header/detail is currently open.

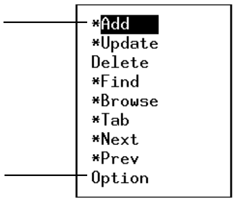
When the Select Commands option is invoked a sub-menu appears that allows you to choose whether the ring menu for the program you are creating will have pull-down menus or the standard ring menu without pull-downs. If you have User Control Library installed you will get two options: With Pulldowns and Without Pulldowns.



The `scradv.a` library, which is part of the User Control Library, must be used in order to have pull-down menus. If you do not have the User Control Library, then you cannot create ring menu items with pull-downs.

For more information on the scradv.a library and using pull-down menus refer to the CASE *Tools Enhancement Toolkit Technical Reference*.

After selecting either With Pulldowns or Without Pulldowns a selection box appears:



Options marked with a * appear on the final program's ring menu

Options without a * do not appear on the final program's ring menu.

Inside of this selection box are the available menu items of the application ring menu. An asterisk next to the menu item indicates that the item is enabled and will appear on the application ring menu. A menu item without an asterisk indicates that the menu item is disabled and will not appear on the application ring menu.

The up and down arrow keys position the cursor and the [ENTER] key toggles the state of the menu item. The [ESC] key accepts the selection and the [DEL] key aborts the selection.

Information is handled differently depending on whether the User Control Libraries are present.

1. If using the With Pulldowns option the information is stored in either the cgmcmndr (if all menu items are present) or cgmmenud (if only a subset of the menu items are present).

2. If you are using the Without Pulldowns option and you choose a subset of the ring menu items, a local detl_menu() or head_menu() function is generated. If you define such a ring menu using the With Pulldowns option, no local menu function is generated but scradv.a is added to the Makefile. Custom ring menus that utilize pull-downs require scradv.a to be linked in.

If you have both With and Without Pulldown the scradv.a version is given precedence (with pull-downs) and you won't get a local menu function.

If you have defined a custom ring menu that does not utilize all of the standard ring menu items, and then revert back to include all of the normal options, the custom menu logic will no longer be generated locally.

If you use the Program Menu option to define a custom menu, scradv.a will be included in the Makefile if you choose any menu but Old_ring as your "Get Ring" value. If you choose Old_ring a local menu function will be generated for the program you are setting up.

The With Pulldowns option indicates that the menu name is "Mainring." The Without Pulldowns option indicates that the menu name is "Old_ring."

It is not possible to add menu items using the With Pulldown or Without Pulldown options. These options only enable or disable the "Mainring" or "Old_ring" menu items. You could modify these menus to add whatever items you want.

If the scradv.a menus are used, two programs can be used to modify their behavior. These programs are invoked from the Program Menu option and the Ring Menu Items option on the Define menu. The Ring Menu Items option builds and maintains the menus and the Program Menu option allows you to customize a defined menu for a specific application.

If you really want scradv.a to manage the standard ring menu without pull-downs you will have to use a do_not_generate trigger to suppress the generation of the local menu function and you will need to either add scradv.a to the LIBFILES in the Makefile or compile with fg.make -L scradv.a. If you do not have the User Control Library and try to use the scradv.a menus, it won't work: the link phase of the compile will fail.

An Example:

In this example, the User Control Library is present. The goal is to change the scr_demo 3 application so that the ring menu items Update and Delete do not appear on the order form ring menu.

1. **First, change directories to `$fg/codegen/demo.4gm/screen3.4gs`.**
2. **Start the Form Painter by typing `fg.form`.**
3. **Open up the order.per form.**
4. **Select Define.**
5. **Choose Select Commands.**
6. **Choose With Menus.**
7. **Select Update and Delete.**

   The asterisk disappears next to these items.
8. **Compile the forms and then generate 4gl, compile 4gl, and run the program.**

If you run the program, the Update and Delete items do not appear on the applications ring menu. To achieve this, a `detl_menu()` function is generated in the `main.4gl`.

The Program Menu Option

Use this option if you have the Enhancement Toolkit and you want to customize the optional pull-down ring menu for your programs.

For an introduction to the Pull Down Menus feature, see the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

Detailed documentation on the Program Menu option can be found in the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

The Ring Menu Items Option

Use this option if you have the Enhancement Toolkit and you want to create new menu items for your ring menus.

For an introduction to the Pull Down Menus feature, see the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

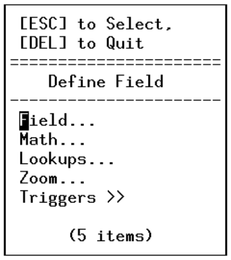
Detailed documentation on the Program Menu option can be found in the Fitrix CASE *Tools Enhancement Toolkit Technical Reference.*

Short Cuts to Define Options

There are two ways you can call up the Field, Input Area, Lookup, Zoom, and Form Defaults forms:

1. Exit the Form Editor and select the appropriate option on the Define pull-down menu.
2. Press [CTRL]-[z] to provide definitions without exiting the Form Editor.

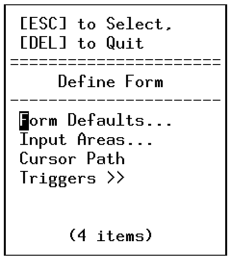
When you press [CTRL]-[z] in the Form Editor, a picker list appears containing a set of relevant define options. The contents of the picker list vary depending on the location of the cursor when [CTRL]-[z] is pressed. That is, if the cursor is currently within a field, the picker list contains options defined at the field level. The Define Field pop-up menu:



```
 [ESC] to Select,
 [DEL] to Quit
======================
    Define Field
----------------------
 Field...
 Math...
 Lookups...
 Zoom...
 Triggers >>

      (5 items)
```

Use the arrow keys to move the cursor to the option you wish to define and press [ESC] to select it. Press [DEL] to quit and return to the form editor. Selecting an item on the picker list is equivalent to selecting the corresponding option on the Define pull-down menu—the same data-entry appears.

If the cursor is currently outside of a defined field in the form editor, pressing [CTRL]-[z] will lead to a picker list containing topics defined at the form or input level. The Define Form pop-up menu:

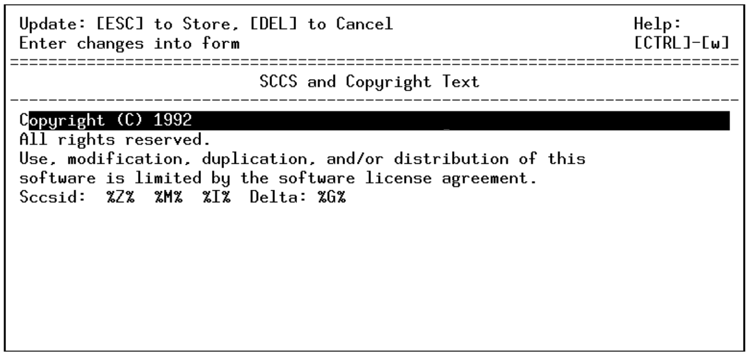


```
[ESC] to Select,
[DEL] to Quit
=====================
     Define Form
---------------------
Form Defaults...
Input Areas...
Cursor Path
Triggers >>


      (4 items)
```

When you quit the selected define form, the program returns to the form editor.

Defining Copyright Text

The Form Painter provides an option that allows you to establish and maintain copyright and SCCS (source code control system) information for any screen form file painted. The option is titled Copyright Text, and is found on the Define pull-down menu. Once selected, the Copyright Text form appears on the screen as follows:



```
Update: [ESC] to Store, [DEL] to Cancel                       Help:
Enter changes into form                                       [CTRL]-[w]
===========================================================================
                         SCCS and Copyright Text
---------------------------------------------------------------------------
Copyright (C) 1992
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%
```

Any information entered and stored on this form will appear as commented header text in all .4gl files generated with the Code Generator.

8

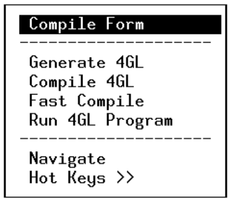
The Run Menu

This section covers the options found on the Run pull-down menu in the Form Painter. The following items are discussed:

- Compiling the Form
- Creating 4GL
- Running the 4GL Application
- Using Hot Keys
- Using Navigation

Run Pull-Down Menu

The topics discussed in this section all relate to options found on the Run pull-down menu. The Run pull-down menu appears as follows:



All the options except Navigate and Hot Keys relate to compiling, generating, or running code. Once the screen form is developed, the final steps to creating your application are as follows:

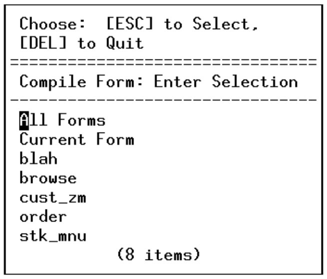
1. Compile the form.
2. Generate the 4GL code from the form specifications.
3. Compile the generated 4GL code.
4. Run the compiled 4GL application.

This section discusses each step in addition to the Navigate and Hot Keys features.

Compiling the Screen Form

The Compile Form option found on the Run pull-down menu runs the screen form through the INFORMIX `form4gl` compiler. The compiler checks syntax and compiles and names error-free forms with .frm extensions. For example, a screen form named `routes.per` would be compiled into a form named `routes.frm`.

Selection of the Compile Form option leads to a picker list, prompting the user to specify which form(s) to compile at the present time. The picker list appears as follows:



```
Choose:  [ESC] to Select,
[DEL] to Quit
================================
Compile Form: Enter Selection
--------------------------------
All Forms
Current Form
blah
browse
cust_zm
order
stk_mnu
          (8 items)
```

You can choose to compile all forms, just the current form, or any individual form. Use the [ESC] key to select the form you want to compile.

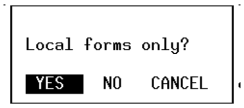
Generating 4GL Code

The generation of source code from .per form specification files is the job of the Fitrix *Screen* Code Generator. The Form Painter provides an option that allows you to generate source code without having to exit the program. The Generate 4GL option on the Run pull-down menu is used to produce source code for a ready-to-use data-entry front-end.

If you have changed the current screen form since your last save, a save is conducted once you select Generate 4GL. If the screen form is not yet compiled, it is compiled in the process of generating the 4GL source code.

Once you select the Generate 4GL option, a window containing a list of forms in your current directory appears. As with the Compile Form option, the picker list offers selections for generating 4GL for the current form, for all forms listed, or for any individual listed form.

When you select All Forms, a dialog box appears asking if you want to compile only the forms in the current directory.



Local forms only?

If you answer NO, the Code Generator uses Version Control to determine where to look to find .per forms not in the current directory. The variable `$cust_path` determines which directories to look in. The default `$cust_path` is 4gc:4gs. For more information on Version Control refer to "Version Control" on page 16-1.

As code is generated, it is displayed on the screen. When code generation is complete, the cursor returns to the Run pull-down menu.

A window appears at the conclusion of code generation to display the results of the operation.

Note

You can change the speed at which the code is generated by first selecting the Generate 4GL option, then pressing the [DEL] key. A prompt appears asking if you wish to cancel the generation. At this prompt you can change the speed of generation by typing the number of the generation level, from 0-5, 0 being the fastest. The default generation level is 4.

Compiling Generated Source Code

The next step following the generation of 4GL code is the compilation of that source code. Generation and compilation of 4GL code are performed separately, allowing you to modify the generated code.

A more detailed description of source code compilation is found in "Compiling Generated Source Code" on page 8-5. The discussion here is limited to running the Compile 4GL option, found on the Run pull-down menu.

Once selected, the Compile 4GL option uses the information from the local Makefile to compile the .4gl files found in the local directory. No picker list appears after this option is selected—compilation begins immediately on the source code in the local directory.

Once the source code successfully compiles, the cursor is returned to the Run pull-down menu on the command line.

Running a Compiled Program

The compiled data-entry interface can be executed from within the Form Painter by executing the Run 4GL Program option found on the Run pull-down menu. Once you select this option, the compiled executable (named after the directory in which it resides) is executed.

Upon quitting from the generated program, the cursor returns control to the Run pull-down menu.

Navigation in the Form Painter

The Navigate option, found on the Run pull-down menu, is a powerful enhancement to the Form Painter. Navigation is the ability to move around the system and carry out other jobs without sacrificing your current process. By selecting the navigate menu from within an application, you can "jump" to an event such as reading mail, printing a report, or loading another application.

Events can be classified as internal (internal to the current program) or external. Internal events include accept, backtab, cancel, help, and hot keys. External events include operating system events, or any event handled outside of the current program. When finished with the event, you are returned to the point of departure.

The Form Painter allows you to create navigation events for your own use while running the Form Painter. Also, if the site that will be running the application you are creating has the User Control Library, you can create events for use with your application.

For detailed information on the Navigation feature refer to the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

For detailed information on the concept of Navigation and events "Event Handling Logic" on page 15-2.

Hooking a Navigation Event to Your Application

If the site runs your finished application has the User Control Library installed, you can hook navigation events to your programs.

Once the event is specified, you need to create and hook the event into your program. This involves creating a trigger that adds a few lines of code to the 4GL program.

To hook your navigation event to your application:

1. **Specify the event by selecting the Add a Navigation Action option from the Navigate menu.**

   The navigate menu can be displayed by selecting Navigate from the Run pull-down menu or by pressing [CTRL]-[g].

2. **Add an `on_event` trigger to the trigger file for the specific form from which you want the action to occur.**

   Make sure that you place the `on_event` trigger in the appropriate section of the trigger file. If you put your trigger in the input 1 section, the event will only be executed from the input 1 area of the program.

3. **The `on_event` trigger requires an argument. Enter the value that you put in the Action Code field for your event.**

   For example, if you created an event and put `my_event` in the Action Code field, your trigger would look like the following:

   ```
   input 1

   on_event my_event
        display "You have just executed my_event."
        sleep 3;
   ```

   This example displays the words "You have just executed my_event" on the screen whenever this event is executed from the header section of the form.

4. **Generate 4GL code.**

5. **Transfer navigate table information to destination database.**

After setting up an event in this manner, it can then be executed while running the application by selecting the event from the Navigation Menu, or it can be assigned to a hot key.

Hot Keys

The Hot Keys option on the Run pull-down menu allows you to associate keys with Navigation events for convenient, instant execution of events. Within the Form Painter, you can set up hot keys for your own use while using the Form Painter. Also, if the site that runs the application you are creating has the User Control Library, you can create hot keys for use with your application.

The Hot Keys feature is part of the User Control Library of features. For more information refer to the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

9

Database Administration

This section discusses the Database Administration system. The Database Administration system gives you the flexibility of modifying your database through the front end. You can create columns and tables as you create your application.

- Using the Database Administration system

Using the Database Administration System

The Form Painter includes a Database Administration system. The Database option on the Form pull-down menu leads to the Table Information form. This form allows you to use the desktop environment of the Form Painter to modify the current database. You work with the database without exiting from the Form Painter.

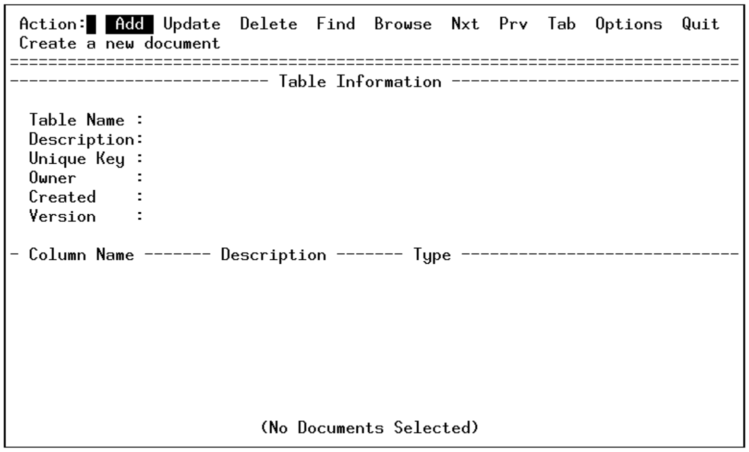
This option allows Form Painter users to modify the database; tables can be added, modified, and removed. With this option, permissions are an issue to consider. You cannot set or modify database privileges through the Database option; however, database privileges can be set elsewhere (e.g., ISQL).

You can also invoke the Database Administration feature directly from a UNIX prompt by typing `fg.dbadmin`.

Note

The Database Administration program does not communicate directly with the dbmerge program. Changes made to a database with the database administration program must also be made to dbmerge (if you want your changes to affect the sample database.)

The Table Information form appears as follows:



```
 Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
 Create a new document
============================================================================
--------------------------- Table Information ------------------------------

  Table Name :
  Description:
  Unique Key :
  Owner      :
  Created    :
  Version    :

- Column Name ------- Description ------- Type -----------------------------







                           (No Documents Selected)
```

The fields appearing in the Main section of this form are as follows:

Table Name: This field accepts up to 18 alphanumeric characters. Table names must be unique within a given database, and must begin with a letter. Do not use an INFORMIX-reserved word as a table name. Once stored in the database, the Table Name field becomes no entry; it cannot be modified on this form.

Description: This field stores a table description of up to 30 alphanumeric characters. The table description can be changed at any time.

Unique Key: Enter column names that make a unique row within the table. Column names must be separated with a comma. The entry in this field can be updated at any time. Once the unique key for a table is defined on this form, it serves as the unique key definition for input areas that use the table as the main table.

Owner: This field is system-maintained; it cannot be manually modified. Once the document containing the table information is stored, the table is created within the current database. The name of the user entering the document is then listed as the owner.

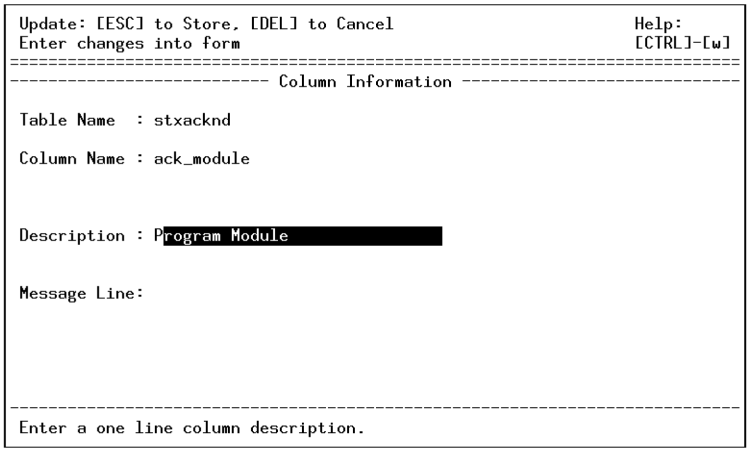
Created: This field is system-maintained; it cannot be manually modified. The field displays the creation date of the table.

Version: This field is system-maintained; it cannot be manually modified. When a table is first defined through this form, it is considered version one. Each future modification to the table causes the version number to be incremented. For instance, a document with the value of three in the Version field has been modified twice—the initial value of one plus two increments yields a current value of three.

The columns appearing in the detail section of this form are as follows:

Column Name: This column stores the column name for a given table field. The field accepts an entry of up to 18 alphanumeric characters. Once stored, the entry in this field cannot be modified.

Description: This optional column stores a description for the field name. Descriptions may be up to 30 alphanumeric characters long, but only 18 characters are displayed. The Zoom feature displays a form which allows you to enter the full 30 characters for the description as well as the message you wish to display when the cursor enters the field on the working application. When selected, the zoom form appears as follows:



```
 Update: [ESC] to Store, [DEL] to Cancel                       Help:
 Enter changes into form                                       [CTRL]-[w]
===========================================================================
-------------------------- Column Information ----------------------------

 Table Name  : stxacknd

 Column Name : ack_module


 Description : Program Module

 Message Line:




---------------------------------------------------------------------------
 Enter a one line column description.
```

Type: This field stores the field type. Field types are valid INFORMIX data types. Field types include:

- Byte—a data object that contains any kind of binary data. It has no maximum size. Bytes are one type of BLOB (binary large object). You can use bytes only in 4GL version 4.00 or higher and only with the OnLine engine.

- Char—a character string up to 32,511 characters long.
- Date—a date entered as a character string.
- Datetime—a moment in time that can include year, month, day, hour, minute, second, and fraction of a second. This data type is only available with 4GL version 4.00 or higher.
- Decimal—a decimal floating-point number.
- Float—a floating-point number corresponding to the double C data type.
- Integer—a whole number from -2,147,483,647 to +2,147,483,647.
- Interval—a span of time that can include year, month, day, hour, minute, second, and fraction of a second. This data type is only available with 4GL version 4.00 or higher.
- Money—a decimal data type displayed with a leading dollar sign.
- Serial—a sequential integer assigned automatically by the database engine.
- Smallint—a whole number from -32,767 to +32,767.
- Smallfloat—a floating-point number corresponding to the floating C data type.
- Text—a data object that contains text data. It has no maximum size. Text is the other type of BLOB (binary large object). You can use text only in 4GL version 4.00 or higher and only with the OnLine engine.
- Varchar—a variable-length text string of up to 255 characters. It can include ASCII characters, tabs, and newlines. You can use varchars only in 4GL version 4.00 or higher. You also need to run the OnLine engine if you specify varchars from a database (table and column) in your form.

Please consult "Engine/4GL Compatibility" on page E-4 and your INFORMIX-4GL documentation for further information.

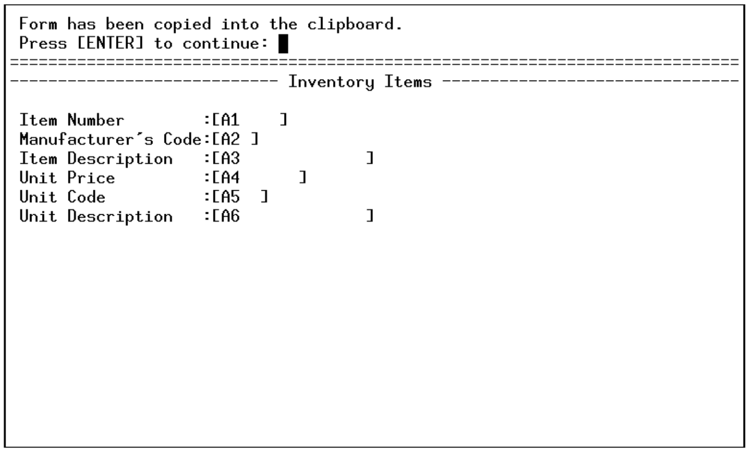
Note

Once a column is stored as `not null`, the `not null` definition cannot be removed.

Using the AutoForm Feature

The Database Administration system offers a feature that copies the columns of the current table into a Clipboard page. Later the page can be used as a template for building new data-entry forms. You can access the AutoForm feature through the Options command on the main ring menu of the Table Information form. The Table Information form is accessed through the Form Painter with the Database option on the Form pull-down menu.

The Form Painter displays the data created from columns in the selected database table. The following example is based on the orders table from the `stores` sample database (installed with INFORMIX-4GL).



```
 Form has been copied into the clipboard.
 Press [ENTER] to continue: █
============================================================================
---------------------------- Inventory Items -------------------------------

 Item Number        :[A1    ]
 Manufacturer´s Code:[A2 ]
 Item Description   :[A3             ]
 Unit Price         :[A4      ]
 Unit Code          :[A5  ]
 Unit Description   :[A6             ]
```

The text and definitions for the selected table comprise a new page on the Clipboard, titled according to the contents of the Description field on the Table Information form. The data on the Clipboard page can then be used as a template for other projects involving all or part of the particular database table.

For further information on using the Clipboard, see "Working with the Clipboard" on page 6-5.

To generate a data-entry form based columns in a table:

1. **From the Form Painter menu line, select New from the File pull-down menu.**

2. **Select what type of form you want to create.**

3. **Name the form.**

4. **Select the Database option on the Form pull-down menu.**

5. **Make a particular database table current on the Table Information form.**

   Use the Find command to locate a table.

6. **Select the Options command on the main ring menu.**

7. **Select the AutoForm command on the Options ring menu.**

   A generic form appears along with a message explaining that the form was copied to the clipboard.

8. **Quit out of the Database Administration form.**

9. **Move down into the Form Editor.**

10. **Press [CTRL]-[p] to paste the automatically generated screen image.**

11. **Continue to modify the form as you like.**

Using the Database Administration Recorder

The Database Administration feature also contains a "recorder" that allows you to apply changes made to one database to other databases. It also provides a record of all changes made to your database through the Database Administration program.

Whenever you modify a database with the Database Administration feature, a log of all of your database modifications is created with ISQL statement formats. Then with ISQL, you can use the recorded log to alter other databases. This means that you can easily modify multiple databases with little effort.

Each time you perform an Add, Update, or Deletion of a table while in the Database Administration program, the ISQL statement required to perform that task is written to a log file named `dbadmin.sql` in the current directory. Each action that is performed on a particular table is appended to the log file.

Example: You add a table called "freddy" with a column "name char(10)" to the `stores` database.

The following log file would be appended to the `dbadmin.sql` log in your current directory.

```
 { Fri Mar 27 10:57:29 PST 1992 }
database stores;
create table freddy (name char(10));
```

Notice that a time stamp is placed before each ISQL command in ISQL comment format.

Remember that changes are appended to the existing `dbadmin.sql` file. This means that maintenance of this file is left up to you.

Defining Column Level Help Text

With the Database Administration program you can define column level help for your application. Information about defining column level help with the Database Administration program can be found in "Defining Application Help Through the Form Painter" on page 15-27.

Part Three

The Code Generator

10

Creating Screen Forms

This chapter discusses in detail the different types of data-entry screens that can be generated with the Fitrix *Screen* Code Generator. This chapter covers:

- Designing screen forms
- Header screens
- Header/detail screens
- Extension screens
- Add-on header screens
- Add-on detail screens
- Query screens
- View-header screens
- View-detail screens
- Browse screens
- Zoom screens

Steps to an Application

This section is a simple step-by-step introduction to the creation of a data-entry application using the Fitrix *Screen* Form Painter and Code Generator. Use this list as an introduction to the flow of activities in Fitrix *Screen* as well as a checklist for development projects.

Preparatory Steps

1. If database tables for the application do not yet exist, plan the tables you need. Decide how the data is organized in the database.
2. Sketch the data-entry form on paper.
3. Decide on the main table for the form. The main table is the table that the screen is referencing. If the form is a header/detail form, also decide on the main detail table for the scrolling (detail) section.
4. Chart the columns in the database tables that the data-entry form maintains.
5. Plan for any fields on the form that requires a Zoom into a reference file to select a valid entry.
6. Decide whether any fields perform lookups. Plan out which fields automatically have values returned to them based on a lookup.
7. Select the Database option on the Form pull-down menu. If tables and columns are not already established, add them to the database that the data-entry form uses.

Form Painter Steps

1. If the form is not a header, make sure you are operating in Expert mode.
2. Select New Form, choose the form type you want, and name the form.
3. Define the form defaults, including the main table.
4. Define the input area for the main and scrolling sections.
5. Define the fields in the main (header) section of the form.

6. Enter the fields in the scrolling (detail) section. Enter one row, then copy it onto the clipboard. Use the clipboard to paste in copies for the rest of the screen array.
7. Specify the `join` statement to link the detail section to the main table of the main section of the form.
8. Carry out additional (optional) steps:
   - Define the lookups taking place in the data-entry form.
   - Define the formula for any form-only math field in the data-entry form.
   - Define Zooms for any fields that have the capability of allowing the user to select from a group of established entries. Be sure to create and name all referenced zoom forms.
   - Define a browse form for working with multiple documents. Make sure the browse form uses the same table that the main data-entry form uses.
9. Save the data-entry form. This step actually generates the .per file used by the Code Generator to create the 4GL source code.
10. Compile the data-entry form. To do this, select the Compile Form option on the Run pull-down menu.

Code Generator Steps

1. If you want to make any modifications to the generated code, create triggers or block commands. The code you write for triggers and blocks is incorporated into the base code with the Featurizer. You can also create a post processor script and identify it with the `$local_scr` variable.
2. Select the Generate 4GL option on the Run pull-down menu. This invokes the Code Generator on the data-entry form. The 4GL source code is created by this step.
3. Compile the generated 4GL code with the Compile 4GL option.
4. Run the 4GL application.

Form Types

You can define ten common types of forms with Fitrix *Screen*: header, header/detail, browse, zoom, extension, add-on header, add-on detail, query, view-header, and view-detail.

When creating a new form, you can choose your form type by selecting your choice from the Screen Type list. The form type can also be specified in the Form Type field on the Form Definition form in the Form Painter.

The following screen types are used as main input screens.

header: This is a flat type. Header forms contain one input area and one main table.

An example of a header screen can be found in `$fg/codegen/demo.4gm/screen1.bak/custform.per`.

header/detail: this is a flat type (header) with another scrolling (detail) section joined to the header. Header/detail forms are suited for order forms where there is one occurrence for customer information and multiple line items for merchandise.

An example of a header/detail form can be found in `$fg/codegen/demo.4gm/screen3.bak/order.per`.

The following auxiliary screen types are not used as stand-alone data-entry screens. They are generally called from the "main" input program.

add-on header: this is a header screen used in conjunction with another header or header/detail screen to provide an extra window of fields. This screen type generates disk read and write functions.

An example of an add-on header form can be found in `$fg/codegen/demo.4gm/screen5.bak/cust.per`.

add-on detail: this is a scrolling detail-only form. This form can be called from any other form to display any detail information. This screen type generates disk read and write functions.

An example of an add-on-detail screen can be found in `$fg/codegen/demo.4gm/screen8.bak/adddtl.per`.

extension: this is a special type of screen that enables you to include an extension of the main header table or detail table.   This screen type shares data with the main screen.

An example of an extension screen can be found in `$fg/code-gen/demo.4gm/screen7.bak/company.per`

query: this form is used only for building an SQL query. This form can replace the `mlh_construct` function.

An example of a query screen can be found in `$fg/code-gen/demo.4gm/screen9.bak/custqry.per`

view-detail: this is a detail-only form that allows you to view data but not alter it.

An example of a view-detail screen can be found in `$fg/code-gen/demo.4gm/screen9.bak/ordview.per`

view-header: this is a flat form used to view header information.

An example of a view-header screen can be found in `$fg/code-gen/demo.4gm/screen9.bak/custvw.per`

The following special screen types are unlike any other screen type. These types are used in conjunction with the main input program and are basically used to locate and select information.

browse: this is a scrolling type screen whose main table is the same as the header section main table. It enables you to view one row of the header table per line rather than one row per screen. Only one browse screen may be used per program. An example of a browse form can be found in `$fg/code-gen/demo.4gm/screen3.bak/browse.per`.

zoom: this is a special type of screen that enables you to view and/or retrieve data from another table (or set of tables which are "joined"). An example of a zoom form can be found in `$fg/code-gen/demo.4gm/screen3.bak/custzm.per`.

Form Design

This section discusses basic concepts in effective form design. The Appendix contains a form style guide providing information on established design conventions.

The proper structure of a form is defined in your INFORMIX-4GL documentation. The structure established by INFORMIX-4GL is followed here, though the Form Painter appends an additional section, `FGSS`, to the perform to direct the operation of the Code Generator.

This discussion points out the issues that you should bear in mind prior to invoking the Fitrix *Screen* Form Painter. The following list of questions is by no means a complete checklist of what you should decide prior to painting a form. Nevertheless, it does cover some basic points that expedite your use of the Form Painter.

- **What is the purpose of the form?** This is perhaps the most basic question regarding the painting of forms. Decide how the screen form affects the database. Determine how the form complements other forms in the same application. Try to condense the purpose of the form into a brief statement.
- **What type of form are you creating?** Decide whether the data-entry form is header(flat file) or header/detail. Is the form used as a Browse or Zoom?
- **How are fields grouped on the form?** Pay attention to the grouping of related data. This concept leads to better organization in form design. Well-organized forms are easier to understand and use. Although the cut and paste capabilities in the Form Painter make reorganization of fields relatively simple, a drawing done ahead of time can only expedite the process.
- **Which table(s) will the form use?** Know ahead of time which table(s) the form will use.
- **What database engine and 4GL version are you using?** Make sure that your engine and 4GL are compatible with each other as well as with this software.
- **What type of validation is required for fields on the form?** Determine which fields require data checks and data-entry validation. This planning ensures that the new form preserves the database's integrity.

- **Do any fields on this form use browse and zoom screens?** If this is a primary data-entry form (header or header/detail), do any of the fields use a reference (zoom screen)? Is a browse form used to display current documents on individual rows?
- **Do any fields require mathematical computation?** Plan for any required equations.

Form Limitations

Forms cannot be designed with a completely free hand—some limitations on structure do exist. Observe the following limitations as you plan the structure of your form:

- The maximum number of lines in the input section of a form must not exceed 18. Additionally:

  Two lines are reserved for the border.

  One line is reserved for messages at the bottom of the form.

  Three lines are reserved at the top for the ring menu, message line, and the double dashed line.

  Which brings the total number of lines to 24.
- The maximum number of columns in a form must not exceed 76.
- The maximum number of lines in the completed .per form specification file may not exceed 200.
- The maximum number of .per forms in any one directory must not exceed 50.

Header Screens

The simplest type of .per file is the header screen. Header screens are also known as flat file applications. Such applications operate on one row of a table at a time. An example of a header application can be found in `scr_demo 1`.

The `standard` database contains the table `customer`. The records for this table can be maintained through the demo `scr_demo 1` data-entry interface based on the code generated from the `custfrm.per` file.

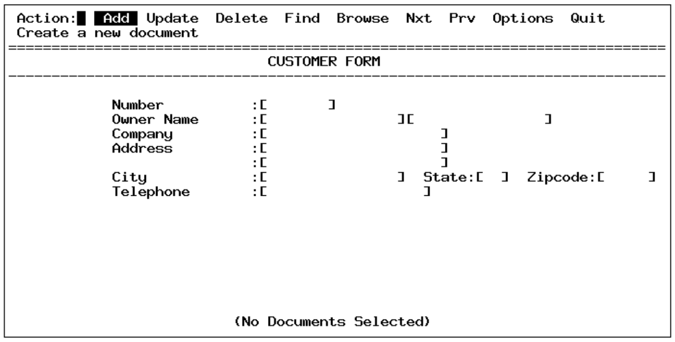
The following is a sample .per specification file that is used to create a header application.

```
DATABASE standard

SCREEN
{
CUSTOMER FORM
------------------------------------------------------------------------
Number          :[f000   ]
Owner Name      :[f001           ][f002           ]
Company         :[f003                ]
Address         :[f004                ]
                :[f005                ]
City            :[f006           ]  State:[a0]  Zipcode:[f007 ]
Telephone       :[f008              ]
}
TABLES
 customer

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0   = customer.state, UPSHIFT;
f007 = customer.zipcode;
f008 = customer.phone, PICTURE = "###-###-#### XXXXX";

INSTRUCTIONS
SCREEN RECORD cust (customer.customer_num, customer.fname, customer.lname,
        customer.company, customer.address1, customer.address2,
customer.city,
        customer.state, customer.zipcode, customer.phone)
```

Notice that the `INSTRUCTIONS` section includes only one `SCREEN RECORD cust`. This implies that, based on this .per form specification file, the Code Generator generates code for a flat file application.

During code generation, the information in the .per form produces a number of source code files that combine to produce a fully-operational data-entry front end. The source code produced by the Code Generator is the focus of "Source Code" on page 11-1.

Below is a sample header only application that was generated with the Code Generator:



```
 Action:   Add  Update  Delete  Find  Browse  Nxt  Prv  Options  Quit
 Create a new document
=============================================================================
                              CUSTOMER FORM
-----------------------------------------------------------------------------

            Number          :[       ]
            Owner Name      :[               ][               ]
            Company         :[                    ]
            Address         :[                    ]
                            :[                    ]
            City            :[               ]  State:[  ]  Zipcode:[     ]
            Telephone       :[                  ]







                          (No Documents Selected)
```

Header/Detail Screens

Often it is the case that one row of a certain table is related to a set of rows from another table. In order to manage such one-to-many relationships you can create a header/detail (also known as master/detail) application.

On a data-entry level, header/detail screens contain data from one row of the header table and several detail line rows from the detail table.

HEADER	DETAIL
customer data	invoices for that customer
categories of inventory items	items within that category
employee data	list of jobs completed by the employee
cash receipts	invoices paid with that receipt
company profit centers	account activity for that profit center
rental items	customers renting the item
hotel rooms	reservations for each room
inventory vendors	parts available from each vendor
employee payroll data	individual payments made to that employee
any item database	notes concerning each item

The Code Generator allows you to easily create header/detail screens with a section of the window for header information and another section to display lines of detail data. You may toggle between header and detail lines.

The .per form specification file used to create a header/detail application is slightly more complicated than the .per file specified for a flat file application. In short, the following differences apply:

1. The TABLES section must contain more than one table name, with the table used for the header information being listed first.
2. The field tags in the ATTRIBUTES section must be defined with the appropriate table name in addition to the column name (`orders.customer_num`, `customer.fname`, etc.). You must specify the correct table name for each particular column.
3. The INSTRUCTIONS section must include more than one screen record. The screen record for the detail section is in the form of an array, with the number of rows to be displayed at one time enclosed in square brackets (for instance, `s_items[4]`).
4. The FGSS section must include the `join` criteria for the detail table to the header.

An example of a .per form specification file used to generate the header/detail application is shown on the following page. The example shown is the .per file used to generate the `scr_demo 3` demo application.

The detail information appears under the columns `Item Description, Manufacturer, Qty., Price`, and `Extension`. Notice that the field tags are the same from row to row.

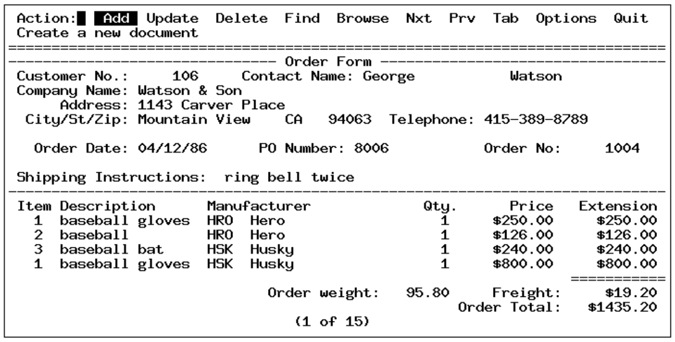
Example Header/Detail Form

The following is an example of a header/detail form:

```
DATABASE standard

SCREEN
{
------------------------------- Order Form ---------------------------------
-
 Customer No.:[f000   ]    Contact Name:[f001           ][f002            ]
 Company Name:[f003                ]
      Address:[f004                ][f005                ]
  City/St/Zip:[f006           ][a0] [f007 ] Telephone:[f008               ]

   Order Date:[f010    ]     PO Number:[f011      ]    Order No:[f009     ]

 Shipping Instructions: [f012                                     ]
----------------------------------------------------------------------------
-
 Item Description      Manufacturer            Qty.      Price   Extension
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
                                                                 ===========
                              Order weight:[f30    ]    Freight:[f31       ]
                                                    Order Total:[f32       ]
}

TABLES
  orders
  items
  customer
  stock
  manufact

ATTRIBUTES
f000 = orders.customer_num, comments =
  " Enter the customer code.";
f001 = customer.fname, noentry;
f002 = customer.lname, noentry;
f003 = customer.company, noentry;
f004 = customer.address1, noentry;
f005 = customer.address2, noentry;
f006 = customer.city, noentry;
a0 = customer.state, noentry;
f007 = customer.zipcode, noentry;
f008 = customer.phone, noentry;

f009 = orders.order_num, noentry;
f010 = orders.order_date, format = "mm/dd/yy", default = today, comments =
  " Enter the order date.";
f011 = orders.po_num, comments =
  " Enter the customer’s purchase order number.";
f012 = orders.ship_instruct, comments =
  " Enter any special shipping instructions to show on the invoice.";

f14 = items.stock_num, comments =
  " Enter the stock number for this line item.";
f15 = stock.description, noentry;
f16 = items.manu_code, comments =
  " Enter the manufacturers code for this stock number.";
f17 = manufact.manu_name, noentry;
f18 = items.quantity, comments =
  " Enter the number of units sold for this item.";
f19 = stock.unit_price, noentry;
f20 = items.total_price, noentry;

f30 = orders.ship_weight, comments =
  " Enter the total shipping weight for this order.";
f31 = orders.ship_charge, comments =
  " Enter the total shipping charge for this order.";
f32 = formonly.t_price type money, noentry;
```

```
instructions
screen record s_order (orders.customer_num, customer.fname, customer.lname,
    customer.company, customer.address1, customer.address2, customer.city,
    customer.state, customer.zipcode, customer.phone, orders.order_date,
    orders.po_num, orders.order_num, orders.ship_instruct,
orders.ship_weight,
    orders.ship_charge, formonly.t_price)

screen record s_items[4](items.stock_num, stock.description,
items.manu_code,
    manufact.manu_name, items.quantity, stock.unit_price, items.total_price)

delimiters "  "

{
#####################################################################
FGSS
#####################################################################

defaults
    type    = header/detail
    init    = order_num > 100

input 1
    table   = orders   (default = 1st table in the "tables" section)
    key     = order_num
    filter  = order_date > "12/31/80"
    order   = order_num
    math    = t_price = sum(total_price) + ship_charge
    lookup  = key=customer_num, table=customer,
              filter=customer_num = $customer_num
    zoom    = key=customer_num, screen=cust_zm, table=customer

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

}
```

After running the Code Generator on the preceding .per form, the 4gl source code can be compiled to produce a ready-to-use data-entry front end. The data-entry screen appears as follows:



```
Action:█ Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Create a new document
=============================================================================
------------------------------ Order Form -----------------------------------
Customer No.:     106     Contact Name: George           Watson
Company Name: Watson & Son
     Address: 1143 Carver Place
 City/St/Zip: Mountain View    CA   94063  Telephone: 415-389-8789

  Order Date: 04/12/86      PO Number: 8006           Order No:     1004

Shipping Instructions:  ring bell twice
-----------------------------------------------------------------------------
Item Description      Manufacturer             Qty.       Price   Extension
  1  baseball gloves  HRO  Hero                  1     $250.00     $250.00
  2  baseball         HRO  Hero                  1     $126.00     $126.00
  3  baseball bat     HSK  Husky                 1     $240.00     $240.00
  1  baseball gloves  HSK  Husky                 1     $800.00     $800.00
                                                                ===========
                              Order weight:   95.80    Freight:      $19.20
                                                   Order Total:    $1435.20
                                 (1 of 15)
```

The following function is how other screen types are hooked into your main program.

The `socketManager()` Function

The `socketManager()` function is the main flow control manager for all screen types other than header or header/detail screens. This function controls what functions get generated for a particular type of screen. The Code Generator automatically creates a flow manager for the main screen, which can be a header or header/detail. You must add the call to the `socketManager()` function for all additional screens, except for zoom screens in most cases.

`socketManager("`*`screen_name`*`", "`*`screen_type`*`",`
`"`*`flow_manager`*`")`

`screen_name` The name of the screen without the .per extension.

`screen_type` The type of screen. The screen type can be one of the following: extension, add-on header, add-on detail, zoom, query, view header, view detail, custom, and single_function. The single_function screen type is discussed next.

`flow_manager` The flow manager is usually "default." If you want to use your own custom flow manager you can enter "custom." If you pass custom to the `socketManager()` be sure to provide a local `F_`*`scr_id`*`()` function.

For example, extension screens have four types of flow control managers: flat_ext, deep_ext, view, custom. Refer to "Extension Screens" on page 10-17.

The single_function Screen Type

The single_function screen type is used to call a single function in a screen's flow without having to pass control to the screen's flow control manager.

The flow manager for the single_function screen type is the function in the screen's switchbox you want to execute ("init", "read", and so on), For example, you could create a header/detail screen that displays a field from a third table. This requires you to create a .per form for the third table, generate code for it, then make the main screen a non_source_form and add the same field name and record from the additional screen. Then you need to add the call to the s`ocketManager("your_screen", "single_function", "display")`. Most of the time you will have two calls to the `socketManager()` function: one call to read the data and another call to display it.

Extension Screens

Extension screens are extensions of the main screen. They allow you to display information related to your main form on separate screens that are called from the main screen. Extension screens use the same table as the main screen.

If you have a table with many columns, you can organize the columns by subset and display each subset on a separate form. For example, you can break up a customer table into subsets, each containing specific types of columns (O/E info, ship-to info, billing, etc.) You can then create extension screens off of the main screen using the subset columns.

Extension screens are "generic" and can be stored in a library and used by other programs. You can call an extension screen from either a header or detail section.

Extension screens support lookups, math, Zooms, free-form notes, help text, and if used as an extension of the main header table, required field logic (nonull). Duplicate check logic is handled by the main screen. The .4gl file that is generated is similar in style to an add-on header .4gl file.

When using extension screens you must set the environment variable called `non_scr_q_elems` equal to "include" and export it. This variable is associated with the Featurizer and can be set in a `screen.opt` file. For more information refer to "The Code Generator Options File (screen.opt)" on page 2-21.

Example Extension Screen Form

The following is an example extension screen .per form:

```
{######################################################################
# Copyright (C) 1992 Fitrix, Atlanta, Georgia.
# All rights reserved.
# Use, modification, duplication, and/or distribution of this
# software is limited by the software license agreement.
# Sccsid:  @(#)  .../demo.4gm/screen7.bak/company.per  1.3  Delta: 4/23/92
#######################################################################
# Screen Generator version: 4.11.UC1 }

DATABASE standard

SCREEN
{

------------------------- Company Information -----------------------


  Company Name : [A1                  ]
       Address : [A2                  ] [A3                  ]
          City : [A4             ]
         State : [A5]
      Zip Code : [A6   ]
  Phone Number : [A7                ]

---------------------------------------------------------------------
}

TABLES
    customer

ATTRIBUTES
A1 = customer.company, comments = "Enter the Company Name";
A2 = customer.address1, comments = "Enter the Address";
A3 = customer.address2, comments = "";
A4 = customer.city, comments = "Enter the City";
A5 = customer.state, upshift, comments = "Enter the State Code";
A6 = customer.zipcode, comments = "Enter the Zip Code";
A7 = customer.phone, picture = "###-###-#### XXXXX",
     comments = "Enter the Phone Number";

INSTRUCTIONS
screen record s_company (customer.company, customer.address1,
    customer.address2, customer.city, customer.state, customer.zipcode,
    customer.phone)

delimiters "  "
```

```
{
######################################################################
FGSS
######################################################################

defaults
    module     = demo
    type       = extension
    init       = 1=0
    attributes = border, blue
    location   = 2, 3

input 1
    table   = customer
    key     = customer_num
    filter  = 1=1
}
```

Creating Extension Screens

The following steps create an extension screen:

1. **Create the extension screen with the Form Painter.**

   The extension screen is created in the same way as a header or add-on header type screen. The extension .per file contains a single screen record, and the `FGSS` section contains a `defaults` and `input 1` section.

2. **Create a `switchbox_items` trigger.**

   In order for your extension screens to get hooked into your program, you need to create a `switchbox_items` trigger and put it in the trigger file for the main screen. The `switchbox_items` trigger is needed for the main screen to recognize the extension screen.

   All triggers used are either of type `defaults` or `input 1`.

   Example:

   ```
   defaults

   switchbox_items
       morord S_morord
       company S_company;
   ```

3. **Create a hook to call the `socketManager()` function.**

   Next, you need to create some kind of hook to call the `socketManager()` function. This can be an `after_field` trigger or some other method such as a navigation event.

   Example:

   ```
   after_field company
        call socketManager("company", "extension", "flat_ext");
   ```

   For an explanation of the `socketManager()` function, refer to "The `socketManager()` Function" on page 10-15.

Creating Zooms from Extension Screens

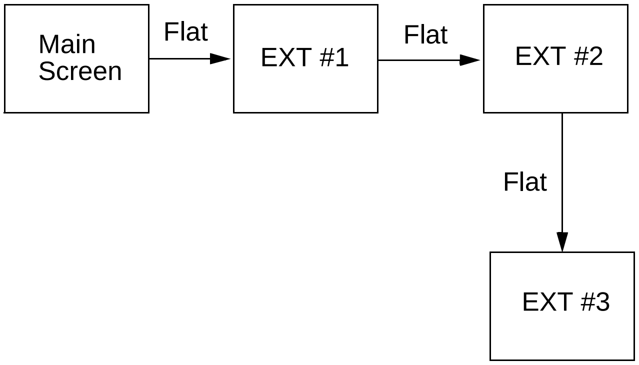
You can create Zooms on extension screens. However, if the zoom field on the extension screen does not appear on the main screen, then you need to create a `switchbox_items` trigger for that Zoom.   If the Zoom field on the extension screen does appear on the main screen, then the Code Generator automatically creates a `switchbox_items` trigger for that Zoom.

Types of Extension Screen Flow Control Managers

You can specify four types of flow control managers for extension screens: flat_ext, deep_ext, view, custom. These types are explained next.

flat_ext: This is the flow control manager for flat type screens. The data in each flat type screen is independent of any other extension screen, except if the flat screen is called from a deep type screen. The `t_clear` function only clears the data for that flat screen upon an interrupt.

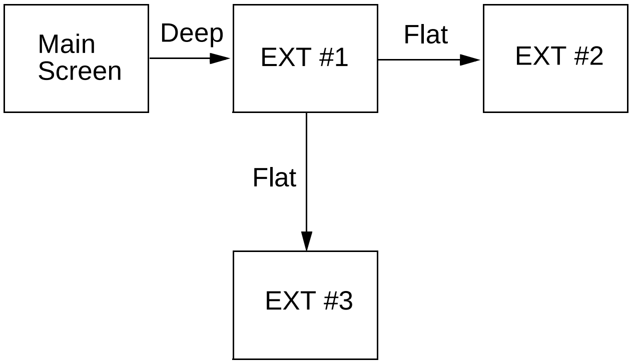
Any number of flat extension screens can be called from other flat extension screens. You can also string together calls to flat extension screens like this:



You can call a flat extension screen from a deep extension screen, but if you do, the data in the flat extension screen depends on the deep screen. If the data in the deep screen is stored by pressing [ESC], the data in the flat screen is also saved. If [DEL] is pressed from the deep screen, then the data is also lost from the flat screen.

deep_ext: This is the flow control manager for deep type screens. Screens called from a deep type screen are dependent upon the deep screen. If [DEL] is pressed while in the deep screen, all data entered into any screen called from the deep screen is not saved.

Anytime the user presses [DEL] from the EXT #1 screen (deep), all changes entered into EXT #2 and #3 screens (both flat), are lost. Each individual flat extension screen is independent from the other.



view: This is the flow control manager for view type screens. This type of flow model allows the user to view the data in an extension screen. An example of it's use is to create a menu command in the main screen Option’s ring menu that invokes the screens with a flow type of view.

custom: Calls the local `F_()` function.

Extension Screen Upper-Level Library Functions

Several Upper-Level Library functions facilitate the use of extension screens. The calls to these functions are automatically generated when you use extension screens.

`t_init():` Initializes the temporary data table used to communicate between screens and sets up the read, write, and delete cursors for this temporary table.

`t_read():` Reads from the temporary data table. A function called `PR_header` or `PR_detail` is also generated in the local code for reading the data from the temporary table into the main screen.

t_write(*row_num, name, data*): Writes to the temporary data table. A function called PW_header or PW_detail is also generated in the local code for writing the data to the temporary table from the main screen.

t_clear(*scr_id*): This function deletes rows from the temporary table based on the screen id passed to it. If the screen id is blank it deletes all rows.

dec_let(d): This function is used to preserve the precision of values contained in decimal & money fields since data transfer is by character fields.

Extension Screen Functions

There are five functions generated that are not called automatically. They are the C_ (clear), T_ (touch), K_ (key), F_ (flow), and OP_ (options) functions.

C_*scr_id*(): This is the clear function. This function initializes both the p_ and q_ records to null.

T_*scr_id*(): This is the touch function. It's purpose is to identify this screen to the object manager.

K_*scr_id*(): This function puts into scratch, via the put_vararg() function, the name of the main input table, followed by pairs of put_vararg() calls for each key field which makes up the key. The first half of the pair is a call to put_vararg() sending the column name of the key field. The second half of the pair, which is a call to put_vararg() sends the q_ record element which holds the actual value of the key field. In the case of a key specified as "customer_num. lname," and with the main input named "customer," the calls to put_vararg() would be as follows:

```
call put_vararg("customer")
call put_vararg("customer_num")
call put_vararg(q_stomer.customer_num)
call put_vararg("fname")
call put_vararg(q_stomer.fname)
```

For more information on varargs see "The Vararg Family of Functions" on page 11-35.

OP_*scr_id*(): The options function can be used when you create a "view" type flow model. A view type screen allows you to view data without adding or updating the form.

***F_scr_id*()**: The flow function can be used to create a custom flow control manager. For certain applications, you may want to create a custom screen that behaves differently from the standard extension screen flow model. To do this you need to build a custom data flow manager, then create a call to `socketManager("screen_name", "extension", "custom")`. This instructs the program to use the custom `F_ ()` function as its flow manager.

For example you may want to call custom library functions that are not provided in the standard flow model. The following F_ function provides the basic flow control for a header extension screen.

Example (cuscr is the name of the screen):

```
function F_cuscr()
     call PW_header()
     call lib_message(menu_item)
     call PR_cuscr()
     call I_cuscr()
     call PW_cuscr()
     call PR_header()
end function
# F_cuscr()
```

Here are the block commands necessary:

```
start file "cuscr.4gl"

    after block F_cuscr flow
      call PW_header()
      call lib_message(menu_item)
      call menu_line()
      call PR_cuscr()
      call I_cuscr()
      call PW_cuscr()
      call PR_header();
```

You also need the following block commands to finish the application:

```
start file "header.4gl"

    after_field field3
         call socketManager("cuscr", "extension", "custom");

start file "main.4gl"

    switchbox_items
        cuscr S_cuscr
```

Extension Screen Limitations

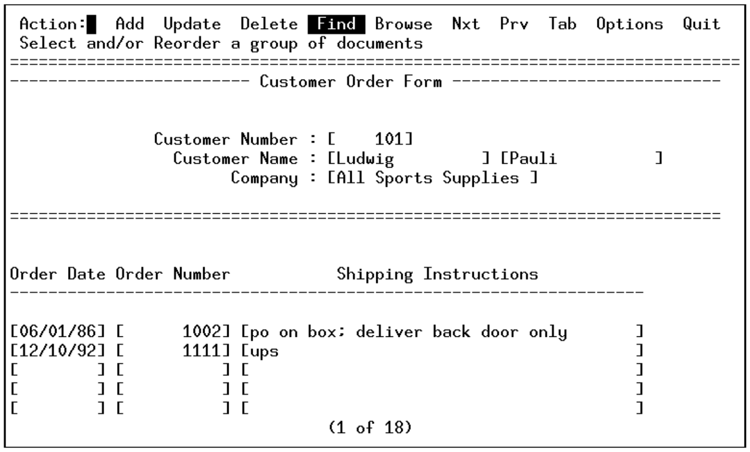
- Currently, the extension screen types do not support the BLOB datatypes.
- There are problems with formonly table fields, especially if the datatypes are money or decimal. What you see is that if the fields are part of a math statement, the updated values are not displayed until after the user presses [ESC] and re-enters the screen.

Extension Screen Demonstration

For an example of what an input program looks like using this screen type, a screen demo has been provided. This demo contains an extension screen from both the header and detail portions of the main screen. It is invoked by typing `scr_demo 7`. This places you into a new shell in `$fg/codegen/demo.4gm/screen7.4gs`.

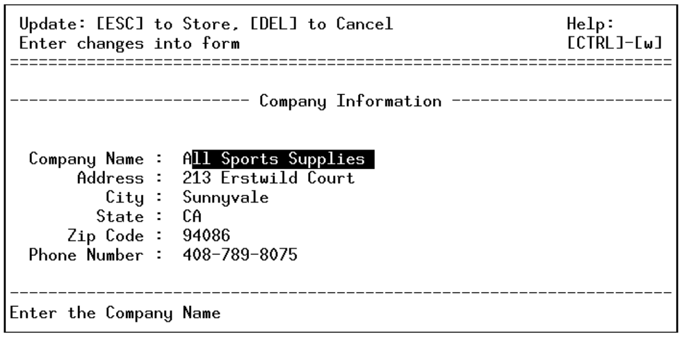
Screen demo 7 is a header/detail form that uses two extension screens to display detailed customer information on one, and shipping information on the other.

The main header/detail form (`screen.per`):



```
 Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
 Select and/or Reorder a group of documents
============================================================================
------------------------- Customer Order Form ----------------------------


               Customer Number : [    101]
                 Customer Name : [Ludwig          ] [Pauli          ]
                       Company : [All Sports Supplies ]

==========================================================================


Order Date Order Number           Shipping Instructions
------------------------------------------------------------------

[06/01/86] [      1002] [po on box; deliver back door only       ]
[12/10/92] [      1111] [ups                                     ]
[        ] [          ] [                                        ]
[        ] [          ] [                                        ]
[        ] [          ] [                                        ]
                                 (1 of 18)
```

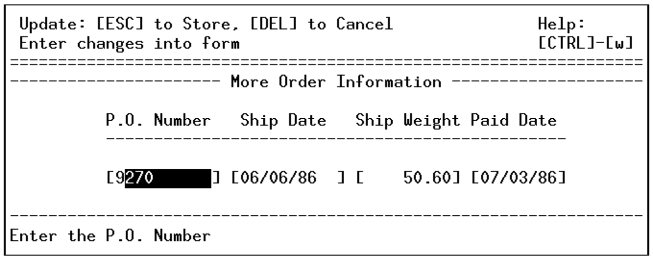
The customer information extension screen (`company.per`):



```
 Update: [ESC] to Store, [DEL] to Cancel                 Help:
 Enter changes into form                                 [CTRL]-[w]
=====================================================================

------------------------- Company Information -----------------------


  Company Name :  All Sports Supplies
       Address :  213 Erstwild Court
          City :  Sunnyvale
         State :  CA
      Zip Code :  94086
  Phone Number :  408-789-8075


---------------------------------------------------------------------
Enter the Company Name
```

The company screen is accessed with after field logic placed in the company field on the main screen. Pressing [ENTER] while in the Company field automatically displays the company extension screen.

The orders extension screen (`morord.per`):



```
 Update: [ESC] to Store, [DEL] to Cancel              Help:
 Enter changes into form                              [CTRL]-[w]
=================================================================
--------------------- More Order Information --------------------

         P.O. Number   Ship Date   Ship Weight Paid Date
         -----------------------------------------------

         [9270      ] [06/06/86  ] [    50.60] [07/03/86]

-----------------------------------------------------------------
Enter the P.O. Number
```

Like the company screen, the orders screen is called up with after field logic. Pressing [ENTER] while in the Shipping Instructions field calls up the orders extension screen.

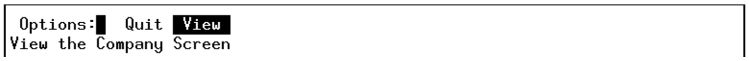
A few simple blocks are needed to connect these extension screens to the main form. The blocks used are in the `screen.ext` file. You can see how these extension screens are hooked in to the main form by looking at this file. The `screen.ext` file:

```
start file "main.4gl"

    switchbox_items
        morord S_morord
        company S_company;

start file "header.4gl"

    after_field company
         call socketManager("company", "extension", "flat_ext");

start file "detail.4gl"

    after_field ship_instruct
        call socketManager("morord", "extension", "flat_ext");
```

Screen demo 7 also provides an example of how to access an extension screen through the Options command on the main ring menu. Look in `options.ext` to see the block command that accomplishes this.

```
start file "options.4gl"

before block ring_options command_key

      command key (v) "View" "View the Company Screen"
        call socketManager("company", "extension", "view")
```

The block displayed above changes the Options command on the ring menu to display this:



```
 Options:  Quit  View
View the Company Screen
```

Add-On Header Screens

Add-on header screens are additional data entry screens to the main screen which can be incorporated into your input programs. These screens access tables other than the header or header/detail tables used by the main screen. Add-on header screens generate disk read and write functions.

Add-on header screens, unlike zoom screens, can be called up from anywhere in a program. For example, you can have an add-on screen appear after the user enters a certain value into a field, after the user is finished inputting a record, before the user starts inputting a record, when an additional ring menu option is selected under "Options," or as an event.

Add-on header screens can be used to add or update information. To update information on an add-on header you need to create a navigation event to call the add-on screen from the main screen. Navigation events allow your user to switch back and forth from the main screen to the add-on header screen.

For an example of how to call an add-on header, see the `order.trg` file in `scr_demo 5`.

Note

The name of the add-on header .per file (not including the .per extension) can be *no more* than 7 characters. The same requirement applies for all .per files.

Sample Add-On Header Form

The following is an example of a simple add-on header .per file:

```
DATABASE standard
```

```
  SCREEN {
                             CUSTOMER FORM
  -----------------------------------------------------------------------
              Number  :[f000   ]
              Owner Name:[f001          ][f002           ]
              Company:[f003                ]
              Address:[f004                ]
              :[f005                ]
              City:[f006           ]  State:[a0]  Zipcode:[f007 ]
              Telephone:[f008              ]
  }
  TABLES
    customer

  ATTRIBUTES

  f000 = customer.customer_num;
  f001 = customer.fname;
  f002 = customer.lname;
  f003 = customer.company;
  f004 = customer.address1;
  f005 = customer.address2;
  f006 = customer.city;
  a0   = customer.state, UPSHIFT;
  f007 = customer.zipcode;
  f008 = customer.phone, PICTURE = "###-###-#### XXXXX";

  INSTRUCTIONS
  SCREEN RECORD cust (customer.customer_num, customer.fname,
         customer.lname, customer.company, customer.address1,
         customer.address2, customer.city, customer.state,
         customer.zipcode, customer.phone)
  {
  #####################################################################
  FGSS
  #####################################################################
  defaults
      type    = add-on
  input 1
      table   = customer
      key     = customer_num
      math    = t_price = sum(total_price) + ship_charge
}
```

Assigning a Key Field

The .per file for an add-on header resembles a header form. When you create an add-on header, you *must* specify a key in the .per form specification file. The key is needed in order to attach certain User Control Library functions such as User Defined Notes to the add-on header screen. For example, an add-on header screen for adding/updating customers might have the following key in the .per file:

```
input 1
   key = customer_num
```

Specify this key in the input 1 section of the add-on header .per file.

Calling the Add-on Header

You must pass three arguments on to the add-on header functions: entry mode, entry filter, and an order by. These arguments are sent via a function called `fgStack_push().`

The arguments to send the `fgStack_push()` function follow:

entry mode Either an "A" to add a record, or a "U" to update a record.

```
call fgStack_push("U")
```

entry filter In Add mode for the add-on header, the filter is null (""). In the case of Update mode for the add-on header, the SQL filter needs to be able to select the row you are going to update.

You should put this filter together into a string, and pass the string name as the argument:

```
let sqlstr = "orders.customer_num = ", get_num
call fgStack_push(sqlstr)
```

If the filter returns *more than one row*, the *first* row retrieved is the one put into an update session.

order by If more than one row is returned by the select statement, this order by is used to order the rows. With the current release, you can only work on one row within the add-on header.

```
call fgStack_push("")
```

However, if there is a scenario where you want to update the first row of a particular group of rows, specifying an order by allows you to do that.

The following example shows how to send the arguments above before calling an add-on header screen:

```
                on_event update_cust
                      let scratch = "orders.customer_num = ",
  entry mode                          orders.customer_num
                      call fgStack_push("U")
  entry filter        call fgStack_push(scratch)
                      call fgStack_push("")
  order by            call socketManager("cust", "add-on header", "default");
```

Creating an Add-On Header Screen

To create an add-on header screen do the following steps:

1. **Create the add-on header screen with the Form Painter.**

   Add-on headers are created much like regular header screens.

2. **Define a key field.**

   Add-on headers must have a key field in the input 1 section.

3. **In the trigger file for the add-on header, create a `switchbox_items` trigger.**

   In the base program trigger file, you must create a `switchbox_items` trigger for the add-on header in order for the main screen to recognize the add-on header screen.

   The first part of the trigger is the name of the add-on header and the second part being the name of the add-on header prefixed by an "S_." In the case of an add-on header named `cust` (`cust.per`), put the following `switchbox_items` trigger in your base program trigger file:

   ```
   default
      switchbox_items
          cust S_cust;
   ```

4. **Create the logic used to call the add-on header.**

   This can be either a trigger or a block command. The most common ways to call an add-on header is to use either an `after_field` trigger or a navigation event.

   One way to use an add-on header in `after_field` logic is based on a certain condition. In the case of `scr_demo 5`, one of the hooks is in `after_field customer_num`, where if `p_orders.customer_num` has the value of zero, then call `socketManager()` to add a customer.

   Another way to use an add-on header is to define a navigation event called `update_cust`. Then map the navigation event to an available hot key.

   When in an Update session of an order, pressing [CTRL]-[u] would put you in Update mode in an add-on header for that particular customer record.

   After setting up a hot key, go into your base program trigger file and put in a trigger like this in the `input 1` section, and re-generate the base program:

   ```
   on_event update_cust
   ```

```
let scratch = "orders.customer_num = ",
               orders.customer_num clipped
call fgStack_push("U")
call fgStack_push(scratch)
call fgStack_push("")
call socketManager("cust", "add-on header", "default");
```

You may also call the `socketManager()` function from within `options.4gl`.

Creating Zooms from Add-On Header Screens

You can create Zooms on add-on screens. However, if the zoom field on the add-on screen does not appear on the main screen, then you need to create a `switchbox_items` trigger for that Zoom.   If the zoom field on the add-on screen does appear on the main screen, then the Code Generator automatically creates a `switchbox_items` trigger for that Zoom.

Add-On Header Triggers

Not all triggers can be used when working with an add-on header. You do not, however, get an error when you use a trigger not supported by a code generation of an add-on header. What follows is a list of triggers, and where they are inserted into the add-on header code:

defaults section

`before_init`: This trigger inserts code in the function `A_`*`scr_id`*`()`, right after the call to `put_scrlib()`, and before the call to `window_pos()`.

`after_init`: This trigger inserts code in the function `A_`*`scr_id`*`()`, right after opening the add-on header window.

input 1 section

`define`: This trigger puts all its elements into the `q_` record. For more information on the `q_` record, see the section on the `q_` record.

`static_define`: This trigger inserts variables static to the source at the top of the .4gl file under define.

`on_event`: This trigger code is inserted into the function `EV_`*`scr_id`*`()`.

`before_field` ***`fieldname`*****:** This trigger is used to insert `before field` logic for the fieldname specified. The logic is put into function `BF_`*`scr_id`*`()`.

`after_field` ***`fieldname`*****:** This trigger is used to insert after field logic for the fieldname specified. The logic is put into function `AF_`*`scr_id`*`()`.

`after_change_in` ***`fieldname`*****:** This trigger is used to insert `after field` logic for the field name specified. The logic is put into the `AF_`*`scr_id`*`()`. This `after change in field` logic executes after leaving the field specified only if the data for this field has changed.

`on_disk_read`: This trigger is used to insert code in the function `R_`*`scr_id`*`()` after the fetch and `rowid` assignment.

`on_disk_update`: This trigger is used to insert code in the function `W_`*`scr_id`*`()` before the update statement.

`on_disk_add`: This trigger is used to insert code in the function `W_`*`scr_id`*`()` before the insert statement.

`before_input`: This trigger is used to insert code in the function `I_`*`scr_id`*`()` before the input command.

`on_exit`: This trigger is used to insert code in the function `Z_`*`scr_id`*`()` after the `"let scratch = null"` statement. It is here that you can use `put_vararg()` to return information back to your base program.

`at_eof`: This trigger is used to add code to the end of the add-on header .4gl source file.

The Add-On Header Demonstration

Screen demo 5 has an example of an add-on header. The base program is used to add customer orders, and the add-on header is used to add or update customers. The demo can be executed by running `scr_demo 5`. This puts you into a new shell, and into the directory `$fg/codegen/demo.4gm/screen5.4gs`. This directory contains several .per files. The `order.per` is the base program perform file, and the cust.per is the add-on header perform file. Once in this directory, you need to run the Code Generator on all the perform files. The program is designed to work in the following manner:

1. If you type in a zero in the customer number field, it calls the add-on header, putting you into Add mode. Saving this newly added customer puts you back into the base program screen, bringing the new customer number with it.

2. If you press [CTRL]-[u] from within an Update session, you are put into the add-on header Update session of the particular customer. You need to set up a navigational event named update_cust specifying nothing for the O/S command, then map this navigational event to the [CTRL]-[u] key.

Transaction Processing Using Add-On Header Screens

When running transaction processing on a Fitrix *Screen* data entry program, a begin work is executed when entering a document via an Add or Update on the main screen. When you press [ESC] a commit work is issued. When you press [DEL] (for cancel) a rollback work is issued. The main tables are added to or updated based when [ESC] is pressed. If [DEL] is pressed, the tables are not added or updated and a rollback work is issued. All work done while in Add or Update mode is rolled back if [DEL] is pressed. The default is to roll back.

If you want to handle transaction processing so a commit work is issued when [DEL] is pressed instead of the default rollback work, you need to put the following lines into the trigger file of the main entry screen for the appropriate programs:

```
after_init
   call put_scrlib("scrn_trx", "commit");
```

Transaction processing notes:

1. No transaction processing takes place when entering or leaving an add-on header, *except* if the add-on header is called from the Options menu.

2. If you use rollback as your default, all work is rolled back including all add-on work and User Control work such as Notes or Hot Keys when you press [DEL] from the main entry screen.

Add-On Header Functions

What follows is the list of functions created in the add-on header source code.

The *scr_id* would be substituted for the basename of the add-on header .per file. If this file is named cust.per, then the *scr_id* would be substituted for the string "cust," like in the function name S_cust().

S_*scr_id*(): This function contains this particular add-on header switchbox mechanism. It works basically the same way as the switchbox mechanism you are used to seeing in Zoom source code. It calls the appropriate add-on header function based on the value of the variable scr_funct. This function must be specified in a switchbox_items trigger in the base program trigger file.

T_*scr_id*(): This is the touch function. It's purpose is to identify this screen to the object manager.

A_*scr_id*(): This function is called to open and initialize the screen. This is one of the first functions called when an add-on header is invoked.

C_*scr_id*(): This is the clear function. This function initializes both the p_ and q_ records to null.

R_*scr_id*(): This function reads the data from the disk into the program variables. It is called *only* if *all* the key fields contain data when the add-on header is invoked. The first thing this function does is retrieve the rowid of the record that is to be updated. If the key is not complete when the add-on header is invoked, this function is never called, and the rowid referenced in the add-on header source code is null. This rowid variable is used in a conditional within the W_*scr_id*() to determine whether or not to insert or update the row in the table.

W_*scr_id*(): This function writes the program variables to disk. The function decides whether to perform an INSERT or an UPDATE based on the contents of the rowid variable. Both the INSERT and the UPDATE statements use column lists. The column lists are created in the following order: first, all the p_ record variables that exist in the main input table, and second, all the q_ record variables that exist in the main table. The same rule applies in the creation of the values list. The creation of the INSERT/UPDATE statements in any add-on header follow the same rule mentioned above.

When a serial field for the main table exists in p_ and/or q_ record(s), the insert statement generated specifies this table column last in the column list, and it puts the value "0" (zero) as the last value in the values list. In the case of an update statement, the serial field is not specified if it exists in the p_ and/or q_ record(s).

Based on the add-on header .per example above, and an input 1 define trigger of:

```
define
    state like customer.state,
    jkl like customer.customer_num;
```

the following SQL INSERT and UPDATE statements are generated. In this particular example, the column customer_num is a serial field.

```
# Insert the new row
insert into customer (fname, lname, company, address1, address2, city,
zipcode, phone, state, fname, customer_num) values (p_custfor.fname,
p_custfor.lname, p_custfor.company, p_custfor.address1, p_custfor.address2,
p_custfor.city, p_custfor.zipcode, p_custfor.phone, q_stomer.state,
q_stomer.fname, 0)
```

Note

The serial field customer_num is referenced last in both the column list, and the values list.

Also in the column list is the duplicate references of the column "fname." This is because it is a field actually on the screen (in the p_ record), as well as in the input 1 define trigger (which is then eventually put in the q_ record). When this particular INSERT is executed, the fname column contains the value from the variable q_stomer.fname in the values list, since it is hooked up to the *last* reference of the column "fname."

```
# Update the existing row
update customer set (fname, lname, company, address1, address2, city,
zipcode, phone, state, fname) = (p_custfor.fname, p_custfor.lname,
p_custfor.company, p_custfor.address1, p_custfor.address2, p_custfor.city,
p_custfor.zipcode, p_custfor.phone, q_stomer.state, q_stomer.fname) where
rowid = q_stomer.row_id
```

Note

The serial field `customer_num` is not referenced.

K_*scr_id*(): This function uses the `put_vararg()` function to pass the name of the main input table, followed by pairs of `put_vararg()` calls for each key field which makes up the key. The first half of the pair is a call to `put_vararg()` sending the column name of the key field, and is followed by the second half of the pair, which is a call to `put_vararg()` sending it the q_ record element which holds the actual value of the key field. In the case of a key specified as "`customer_num. lname`," and with the main input named "`customer`," the calls to `put_vararg()` are as follows:

```
call put_vararg("customer")
call put_vararg("customer_num")
call put_vararg(q_stomer.customer_num)
call put_vararg("fname")
call put_vararg(q_stomer.fname)
```

For more information on varargs see "The Vararg Family of Functions" on page 11-35.

I_*scr_id*(): This function contains the input statement for the add-on header. It is identical to the `llh_input()` function found in `header.4gl` on any header program.

BF_*scr_id*(): This function is the `before field` function for an add-on header. It is similar to the `llh_b_field()` function found in `header.4gl` on any header program.

AF_*scr_id*(): This function is the `after field` function for an add-on header. It is similar to the `llh_a_field()` function found in `header.4gl` on any header program.

AI_*scr_id*(): This function is the after input function for an add-on header. It is similar to the `llh_a_input()` function found in `header.4gl` on any header program.

EV_*scr_id*(): This function is the `on event` function for an add-on header. It is similar to the `llh_event() function` found in `header.4gl` on any header program.

SD_*scr_id*(): This function is the `set data` function for an add-on header. It is similar to the `llh_setdata()` function found in `header.4gl` on any header program.

HI_*scr_id*(): This function is the `highlight` function for an add-on header. It is similar to the `llh_high()` function found in `header.4gl` on any header program.

SH_*scr_id*(): This function is the `show data` function for an add-on header. It is similar to the `llh_display()` function found in `header.4gl` on any header program.

Z_*scr_id*(): This function is called upon the end of an add-on header session. It closes the add-on header window, and initializes `scratch` back to null.

PL_*scr_id*(): This function contains the lookup logic for the add-on header. This function is *not* present if there are no lookups.

MA_*scr_id*(): This function contains the math logic for the add-on header. This function is *not* present if there is no math specified.

PR_*scr_id*(): This function loads the `p_` and `q_` records from the temporary table.

PW_*scr_id*(): This function writes the `p_` and `q_` records to the temporary table.

Add-On Detail Screens

Add-on detail screens are designed to provide auxiliary detail input to your entry programs. The tables used by these screens are other than the tables used in your header/detail program.

Add-on detail screens can be hooked to the following type of screens: header, header/detail (either the header or detail portion), add-on header, extension, and other add-on detail screens.

An add-on detail screen can utilize the following logic: Zooms, lookups, order, filter, autonum, or math.

Add-on detail screens aren’t restricted to *only* being called by a particular entry screen, and can be *plugged in* to several different calling screens, as long as each calling screen sends it the same number of values.

All Zooms used in an add-on detail screen must have `switchbox_items` triggers defined for them in the entry screen’s trigger file.   For more explanation, refer to "The Add-On Detail Demonstration" on page 10-46.

Fields involved in a math equation all must be referenced within the add-on detail screen.

Example Add-On Detail Form

The following is the `adddetl.per` form from scr_demo 8.

```
DATABASE standard

 SCREEN
 {
 ----------------------------------------------------------------------------
  Item Description      Manufacturer             Qty.      Price   Extension
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 [f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
 ----------------------------------------------------------------------------
 }

 TABLES
   items
   stock
   manufact

 ATTRIBUTES
 f14 = items.stock_num, comments =
   " Enter the stock number for this line item.";
 f15 = stock.description, noentry;
 f16 = items.manu_code, comments =
   " Enter the manufacturers code for this stock number.";
 f17 = manufact.manu_name, noentry;
 f18 = items.quantity, comments =
   " Enter the number of units sold for this item.";
 f19 = stock.unit_price, noentry;
 f20 = items.total_price, noentry;

 instructions

 screen record s_items[11](items.stock_num, stock.description, items.manu_code,
     manufact.manu_name, items.quantity, stock.unit_price, items.total_price)

 delimiters "  "

 {
 #####################################################################
 FGSS
 #####################################################################

 defaults
     module     = demo
     type    = add-on
     attributes = white, border
```

```
        location   = 9, 2

    input 1
        table   = items
        join    = items.order_num = orders.order_num
        order   = item_num
        arr_max = 100
        autonum = item_num
        math    = total_price = quantity * unit_price
        lookup  = name=stock_num, key=stock_num, table=stock,
                  filter=stock_num = $stock_num, into=description
        lookup  = name=stock_manu, key=manu_code, table=stock,
                  filter=stock_num = $stock_num and manu_code = $manu_code,
                  into=unit_price
        lookup  = key=manu_code, table=manufact, filter=manu_code = $manu_code
        zoom    = key=stock_num, screen=stockzm, table=stock, noautozoom
        zoom    = key=manu_code, screen=stk_mnu, table=stock,
                  filter=stock.stock_num = $stock_num

    }
```

Note

Also notice that form type in the add-on detail is add-on. The Code Generator determines if the form is an add-on header or add-on detail type form by the screen record. If the screen record is an array, then the form is a detail form. If it is not, then the form is a header form.

Characteristics of an Add-On Detail .per Form

The following are the characteristics of an add-on detail screen:

- The SCREEN section contains only a detail layout.
- The INSTRUCTIONS section of the perform file *only* has the screen record array defined.
- The FGSS section *only* has default and input 1 sections, just like an add-on header screen, but the input 1 section in an add-on detail screen contains info regarding the detail. In other words, in the traditional header/detail screen, the information found under the input 2 section would be the same type of information found under an add-on detail screen's input 1 section.
- The `join` clause contains elements that get passed to the screen being called.

Creating an Add-On Detail Screen

The following are the steps creating an add-on detail screen:

1. **Create the add-on detail .per form.**

   The add-on detail looks much like the detail section of a normal header/detail screen.

   - Specify "add-on" for the form type.
   - Create the detail array.

2. **Create a join clause in the .per to join the add-on detail form to the main form.**

   The `join` clause in your add-on detail .per form is used to tell the add-on detail select statement how to join itself with the data from the screen that invokes this add-on detail. This `join` is defined in the same way for an add-on detail screen as it is defined in any other header/detail screen. The syntax for the `join` clause is:

   ```
   join = detail_tbl.detail_column = other_tbl.other_column
   ```

   In this example, `detail_tbl` is the name of the detail table that this add-on detail maintains, and `detail_column` is a column of the `detail_tbl` table used in the `join`. The `other_tbl` table is used by the screen that calls the add-on detail screen, and `other_column` is a column from the `other_tbl` table used in the `join`. Multiple `joins` can be specified if required, such as:

   ```
   join = detail_tbl.detail_column1 = other_tbl.other_column1 and
   detail_tbl.detail_column2 = other_tbl.other_column2
   ```

   The `other_tbl.other_column` references are translated into "?"'s in the add-on detail screen's select statement. These join elements are referenced only as a way to provide the Screen Code Generator the number of host variables to be supplied by the screen calling this add-on detail screen, as well as letting the add-on detail screen know how many passed data values it should expect to receive.

   In other words, just because the table named `other_tbl` was used in the join does *not* mean that the calling screen has to be tied to the `other_tbl` table.

To clarify this, let’s look at the join from the screen demo 8 demonstration for add-on detail screens:

```
join    = items.order_num = orders.order_num
```

The add-on detail screen is joined to the table named items. The calling screen is a header which is hooked to the table named orders. Hence the join above. The following shows what is generated in the add-on detail select clause:

```
select stuff from items
where items.order_num = ?
```

When the add-on detail screen is called in this example, the calling screen sends an order number value, which the add-on detail screen retrieves and uses in it’s open cursor statement. The add-on detail screen doesn't care whether the data value sent was from orders.order_num or inventory.order_num or stock.order_num. All it knows is that it expects a value, one value (as specified), and that the value represents an order number.

3. Create a call to the add-on detail screen.

Add-on detail screens are displayed by placing a call in the code for the screen from which the add-on is called from. This call is made up of three possible elements:

1. **an additional filter** - used in conjunction with the hard filter specified in the add-on detail .per file. Sending an additional filter is optional, and uses a keyword (filter) to specify that it is used as a filter.
2. **an order by** - to replace the order by specified in the add-on detail .per file. Sending an order by is optional, and uses a keyword (order by) to specify that it is to be used as a filter. If no order by is sent, then the one specified in the add-on detail perform file is used if it exists.
3. **the data values** - these are the join elements. The join_elems keyword is used to specify that these data values are used in the join criteria of the add-on detail select statement. The join_elems keyword is required, followed by the proper number of elements.

The put_vararg() function in the calling screen sends these pieces of data to the add-on detail screen. The add-on detail screen uses get_vararg() to retrieve the sent data. The following is an example of what a calling screen does to send a filter, order by, and three data values:

```
a)     call put_vararg("filter")
b)     call put_vararg("items.order_num > 1000")

c)     call put_vararg("order")
d)     call put_vararg("order_num")

e)     call put_vararg("join_elems")
f)     call put_vararg(p_record.data_value1)
g)     call put_vararg(p_record.data_value2)
h)     call put_vararg(p_record.data_value3)

i)     call socketManager("adddetl", "add-on detail", "default")
```

a) passes the keyword "filter." When the add-on detail screen starts retrieving this data, it first gets the keyword "filter." Because of this keyword, it knows that the next piece of data is the actual filter that it is supposed to use.

b) passes the actual filter to be used.

c) sends the keyword "order."

d) the actual order by elements you wish to use. When the add-on detail reads in the keyword "order," it knows its next read returns the order by that it is supposed to use.

Again, passing additional filters or a substitute order by is optional. Passing data values is required.

e) sends the keyword "join_elems."

f) sends the first actual data value.

g) sends the second actual data value

h) sends the third actual data value.

Since the add-on detail screen in this example expects three data values sent to it, the calling screen must send three data values. If it doesn't, then the add-on detail screen does not display. When the add-on detail reads in the keyword "join_elems," it knows that the rest of the passed pieces of data are in fact actual data values it needs to use in satisfying its `join`. It keeps count of how many it reads, and it must match what it expects.

i) is the last line of code for the call for add-on detail screens. It is the function that brings up the add-on detail screen. It is only after the socketManager() function is called that data passed to it is read. You send the name of your add-on detail screen as an argument to this function. This function call of course is required.

That's it for the hook code required in the calling screens logic. The screen demo 8 demonstration program is explained below, and it is highly recommended you run through this demo and study it before launching into the creation of your own add-on detail screens.

The Add-On Detail Demonstration

Screen demo 8 illustrates the use of an add-on detail screen. The add-on detail screen is invoked in this demo with after field logic from the main header screen, but any event can be used to trigger an add-on detail screen. In this example, the add-on detail screen is named adddetl.per. There are Zooms, lookups, and math defined in the .per file.

Similar to add-on headers, a switchbox_items trigger exists for this add-on detail screen, as well as switchbox_items trigger entries for any other auxiliary screens called from within this add-on detail screen, such as zooms, add-on headers, or add-on details. For example, the entry header is named order.per, and the Zooms called from the add-on detail are named stockzm and stk_mnu. In order for the add-on detail Zooms to work, you need to create the file order.trg, with the following entry:

```
default
  switchbox_items
      adddetl S_adddetl
      stockzm stockzm
      stk_mnu stk_mnu;
```

The first switchbox_items entry is the actual add-on detail screen itself. The S_ preceding the screen name represents the name of the add-on detail's switchbox() function. The second and third switchbox_items entries are for the two Zooms, which can be invoked from within the adddetl add-on detail screen.

The rest of the order.trg trigger file consists of this:

```
input 1
  after_field ship_instruct
```

```
    call put_vararg("join_elems")
    call put_vararg(p_orders.order_num)
    call socketManager("adddetl", "add-on detail", "default");
```

Once again, this add-on detail is invoked with `after_field` logic, namely `after_field` logic for the entry header field named `ship_instruct`. After this field add the following:

```
call put_vararg("join_elems")
```

This statement puts the keyword "join_elems" into the variable arguments, which the add-on detail screen logic reads once it gets control. Once the add-on detail screen finds this keyword "join_elems," it knows that subsequent calls to `get_vararg()` return the data values it needs to use in the `open cursor` statement. These retrieved values are used to substitute the host variables of the `join` clause within the `select` statement.

```
call put_vararg(p_orders.order_num)
```

Here you need to put into the variable argument pool the actual data to be used by the `open cursor` statement in the add-on detail logic. This particular add-on detail only needs one data element to fill its `join` clause, so you only send one. If the `join` required three external data values, then you would have three separate calls to `put_vararg()` here, one for each of the three data values expected.

```
call socketManager("adddetl", "add-on detail", "default")
```

This is the controlling function that activates the add-on detail screen. Notice that this function is sent the name of the add-on detail screen; in this case `adddetl`.

Generic Detail Write

The write functionality implemented within add-on detail screens works differently from the way disk update is handled in the traditional header/detail screens.

In header/detail screens, pressing [ESC] to save the record first deletes all of the initially selected detail lines and then inserts the current detail lines back into the detail table. There has never been an "update" function for detail lines. This way of handling disk updates for detail rows has never been a real problem until you have an instance where a serial column is a part of your detail row, and you depend on its value being consistent once established. Deletion and re-insertion in this scenario results in a desired static serial number becoming very dynamic. This problem has been resolved for add-on detail disk updates in the new add-on detail logic.

Add-on detail screens handle updates differently from other screen types. When it comes time to update the disk, each detail row is determined to be either updated, inserted, or deleted. Furthermore, *no* action is taken on those rows that have *not* had any data changed within them, resulting in performance benefits proportionate to the amount of detail activity a particular interface has.

Note

As stated, this new detail disk update functionality is present only in your add-on detail code and not in your traditional `detail.4gl` code.

Query By Example Screens

The query screen type provides a form into which a user can enter queries to find information. The Code Generator automatically uses the main input screen as a query screen when the Find command is used. You may create your own query screen if you don’t want to query using the main form.

Creating a Query Screen

The following is required to create the query screen.

1. **Create the query screen with the Form Painter**

   Be sure to define the screen type as "query."

2. **Create the call to the query screen and place them in an .ext file. Example:**

```
start file "midlevel.4gl"

replace block mlh_construct define_var
  define
    m smallint,
    n smallint;

delete block mlh_construct end_construct

replace block mlh_construct construct
   call socketManager("custqry", "query", "default")
   let scratch = null
   let n = fgStack_pop()
   if n = 0
   then
       let int_flag = true
   else
       for m = 1 to n
           let scratch = scratch clipped, fgStack_pop()
       end for
   end if;
```

3. **Create a `switchbox_items` trigger.**

   This trigger can be placed either in the trigger file for the main form or, as in this example, in an .ext file with a `start file` command.

```
start file "main.4gl"

switchbox_items
     custqry S_custqry;
```

View-Header Screens

This screen type is used to display header information only. This screen type only generates a read, showdata, and view function.

To use a view-header screen you must do the following:

1. **Call the view-header screen.**

```
# first pass the the rowid to put_vararg
let p_cur = arr_curr()
select rowid into scratch from customer
    where customer_num = q_ordview[p_cur].customer_num
call put_vararg(scratch)
call socketManager("custvw", "view header", "default")
```

2. **Create `switchbox_items` trigger for screen.**

```
switchbox_items
    custvw S_custvw;
```

View-Detail Screens

This screen type is used only to display detail information. View-detail screens only generate a read, showdata, and view function.

To use a view-detail screen you must do the following:

1. **Call the view-detail screen.**

```
on_event tab
   call put_vararg("order")
   call put_vararg("order_date desc")
   call put_vararg("join_elems")
   call put_vararg(p_stomer.customer_num)
   call socketManager("ordview", "view detail", "default");
```

2. **Create `switchbox_items` trigger for screen.**

```
   switchbox_items
       ordview S_ordview;
```

Browse Screens

Ring menu code produced automatically by the Code Generator provides a Browse command. The Browse command facilitates your work with batches of documents. Often, a user needs to update a number of documents. The Browse command is used to page through a number of current documents.

In programs without a `browse.per` form specification file, the Browse option displays the following commands at the top of the screen:

```
Browse:  First    Last    Next    Prev    Goto    Exit
Move to first selected document
```

These Browse commands allow for movement among the selected or current set of documents. Only one document can be viewed at a time.

The Browse option can be further enhanced by condensing current documents to one line of information for array-style display on a browse screen. The browse screen provides summary information of the selected set of documents. The user points to the desired row (representing a unique current document) and selects it. This feature can be added to your applications by creating a `browse.per` file in the application directory.

Example Browse Form

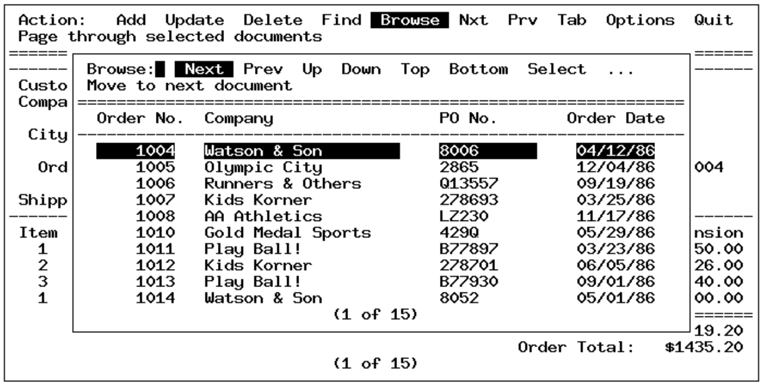
The .per form specification file for the Browse resembles the detail array portion of a header/detail form specification file. That is, the structure of the field tags from row to row underscore the fact that data displayed in a browse form appears as an array. The example of a `browse.per` file shown below is taken from the `scr_demo 3` application.

The `browse.per` file:

```
database standard

screen
{
  Order No.  Company                 PO No.       Order Date
--------------------------------------------------------------
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
 [fp_1    ] [fp_2                ]  [fp_3      ]  [fp_4    ]
}
tables
 orders
 customer

attributes
fp_1 = orders.order_num;
fp_2 = customer.company;
fp_3 = orders.po_num;
fp_4 = orders.order_date;

instructions
screen record b_ordr[10] (orders.order_num, customer.company,
    orders.po_num, orders.order_date)

delimiters "  "

{
#####################################################################
FGSS
#####################################################################
defaults
    type       = browse
    location   = 5,10
}
```

The columns in the form access data in the `customer` and `orders` tables in the `standard` database. The number of rows specified in the screen section as well as the number indicated in the screen record (`b_ordr[10]`), indicate that the browse screen displays up to ten rows at a time.

The following graphic shows how a browse screen appears within a data-entry application:



```
Action:   Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Page through selected documents
======                                                                ======
------  Browse:  Next  Prev  Up  Down  Top  Bottom  Select  ...       ------
Custo   Move to next document
Compa  ==============================================================
          Order No.  Company                PO No.       Order Date
 City  --------------------------------------------------------------
               1004  Watson & Son           8006          04/12/86
  Ord          1005  Olympic City           2865          12/04/86    004
               1006  Runners & Others       Q13557        09/19/86
Shipp          1007  Kids Korner            278693        03/25/86
------         1008  AA Athletics           LZ230         11/17/86    ------
Item           1010  Gold Medal Sports      429Q          05/29/86    nsion
  1            1011  Play Ball!             B77897        03/23/86    50.00
  2            1012  Kids Korner            278701        06/05/86    26.00
  3            1013  Play Ball!             B77930        09/01/86    40.00
  1            1014  Watson & Son           8052          05/01/86    00.00
                                 (1 of 15)                            ======
                                                                      19.20
                                                   Order Total:   $1435.20
                                 (1 of 15)
```

The browse screen displays one-line summaries of current documents. The user selects the desired document by scrolling the cursor to the desired row and selecting the document.

By convention, browse and zoom forms should begin on row 5.

Zoom Screens

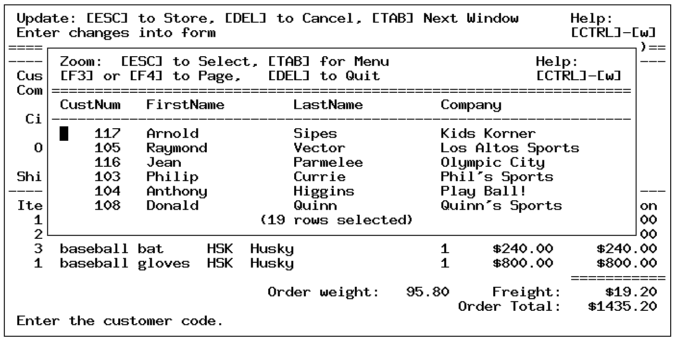
Zoom is a feature that permits users to view a screen of valid column entries, select a value from a list, and paste the selected value (and other dependent values) into the current data-entry form. Zooms can help reduce the likelihood of data-entry errors and free data-entry personnel from having to memorize codes.

In code generated by the Code Generator, the Zoom feature is handled as an event. The default access to the Zoom feature is through [CTRL]-[z], although Zoom can be assigned to other keys as well.

In order to have Zoom logic generated automatically by the Code Generator, you must create a .per form for each Zoom. The zoom .per file needs to be in the style of a detail form, with records displayed one to a row.

For example, the screen demo 3 has been generated with a Zoom in the header section of the screen. The Zoom is setup to be used while the cursor is in the `Customer No.` field. When the user activates the Zoom feature by pressing [CTRL]-[z], a new screen opens up, displaying an array of rows listing codes currently defined in the `customer` table of the `standard` database. When the user selects a row (displaying the customer number desired), the selected entry automatically appears in the `customer_num` field.

Sample Zoom form:



```
 Update: [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
 Enter changes into form                                         [CTRL]-[w]
====                                                                     )==
---- Zoom:  [ESC] to Select, [TAB] for Menu                 Help:        ---
 Cus [F3] or [F4] to Page,   [DEL] to Quit                  [CTRL]-[w]
 Com ================================================================
     CustNum   FirstName         LastName         Company
  Ci ----------------------------------------------------------------
     █    117   Arnold            Sipes            Kids Korner
   O      105   Raymond           Vector           Los Altos Sports
          116   Jean              Parmelee         Olympic City
 Shi      103   Philip            Currie           Phil's Sports
----      104   Anthony           Higgins          Play Ball!            ---
 Ite      108   Donald            Quinn            Quinn's Sports        on
   1                          (19 rows selected)                         00
   2                                                                     00
   3 baseball bat     HSK  Husky                   1      $240.00     $240.00
   1 baseball gloves  HSK  Husky                   1      $800.00     $800.00
                                                                  ===========
                               Order weight:   95.80     Freight:      $19.20
                                                     Order Total:    $1435.20
 Enter the customer code.
```

Calling a Zoom Screen

If your zoom screens are created and attached using the Form Painter then the 4gl code talked about in this section is generated automatically. This section discusses the 4gl code that calls the zoom screens.

A zoom screen is called with the `socketManager()` function just like all other additional screens. You must send an entry filter to the zoom functions with the `fgStack_push()` function before the call to the `socketManager()` function. The entry filter is the initial filter that is used when the Zoom is activated. If this is null (""), then the user goes into the query by example form. The entry filter is sent to the zoom functions via the `fgStack_push()` function. The following is an example zoom screen call:

```
#_zoom_stock_num
when scr_funct = "zoom" and infield(stock_num)
  call fgStack_push("")
  call socketManager("stockzm","zoom", "default")
```

When control returns back to the main program from the zoom screen several lines of code are needed to assign the value returned from the zoom to the current p_ variable.

```
let tmp_str = fgStack_pop()
if tmp_str is not null
then
    let p_items[p_cur].stock_num = tmp_str
    let nxt_fld = "stock_num"
end if
```

The `fgStack_pop()` function returns the value selected by the user from the zoom functions. This value is assigned to the current p_ variable and `nxt_fld` is set.

Creating a Zoom Screen

The first step to adding a zoom screen to an application is to create the zoom .per form. The zoom .per form contains the basic instructions for the zoom window, and is set up like your regular .per file. When naming zoom forms it is a good idea to follow a naming convention so you can recognize zoom .per forms at a glance. For example, the name `cust_zm.per`, uses zm to identify the Zooms. Zoom .per files, like all .per files, must be named with a maximum of seven characters not including the .per extension.

By convention, zoom screens should be located on row five. Zooms should also be centered. Refer to "Centering a Window" on page 18-4 for an easy way to determine the coordinates for a centered window.

The next step is to add the required Zoom information to the main input screen .per file that the Zoom keys from.

Special steps to adding a Zoom to your program:

1. **Create the main input form your Zoom is attached to.**

2. **Specify Zoom logic on the field you want Zoom functionality.**

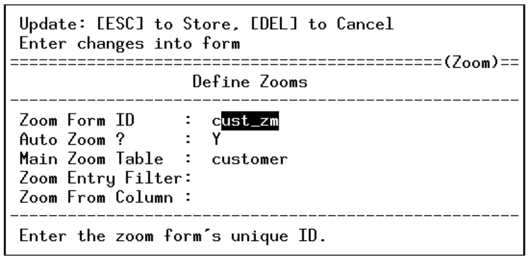
   From the field you want to initiate the Zoom from display the Define Zooms form. Enter the name of the zoom form along with the main table used by the zoom form. If you want to be able to AutoZoom on the field then enter a Y in the Auto Zoom field. If you want to specify a filter for the Zoom, do so in the Zoom Entry Filter field. The Zoom From Column field is required when the following are true:

   1. The table.column name being Zoomed in is different from the name of the column on the screen that the Zoom is attached to.

   2. The screen field you are Zooming from is a character field.

   3. AutoZoom is enabled. Zooms with AutoZoom set to N or a filter, do not require the "from" keyword.

For example, if the field on your screen is named `customer_num` and the column being Zoomed into is called `cus_num`, you would enter `cus_num` in the Zoom From Column field. For more information on the Define Zooms form refer to "Defining a Zoom" on page 7-30.



```
Update: [ESC] to Store, [DEL] to Cancel
Enter changes into form
============================================(Zoom)==
                   Define Zooms
----------------------------------------------------
Zoom Form ID     :  cust_zm
Auto Zoom ?      :  Y
Main Zoom Table  :  customer
Zoom Entry Filter:
Zoom From Column :
----------------------------------------------------
Enter the zoom form's unique ID.
```

3. **Create the zoom form.**

4. **On the Define the Form form, specify the main table for the Zoom.**

5. **On the Define the Form form, specify a returning field if you wish to return data from the zoom form to the main screen.**

   If you want the Zoom to return data to your main form, specify the name of the field on the main form in the Returning (zoom) field. The `returning` keyword identifies the name of the field on the zoom that is to return information to the calling function. Only one field per zoom screen can be specified to return information. You can, however, return information into more than one field with a little bit of code manipulation. Refer to the discussion of vararg functions for a real example of returning more than one value with a Zoom. See page "Examples of put_vararg() and get_vararg()" on page 11-39. This `returning` keyword is required for zoom screens and has no default.

Example Zoom Form

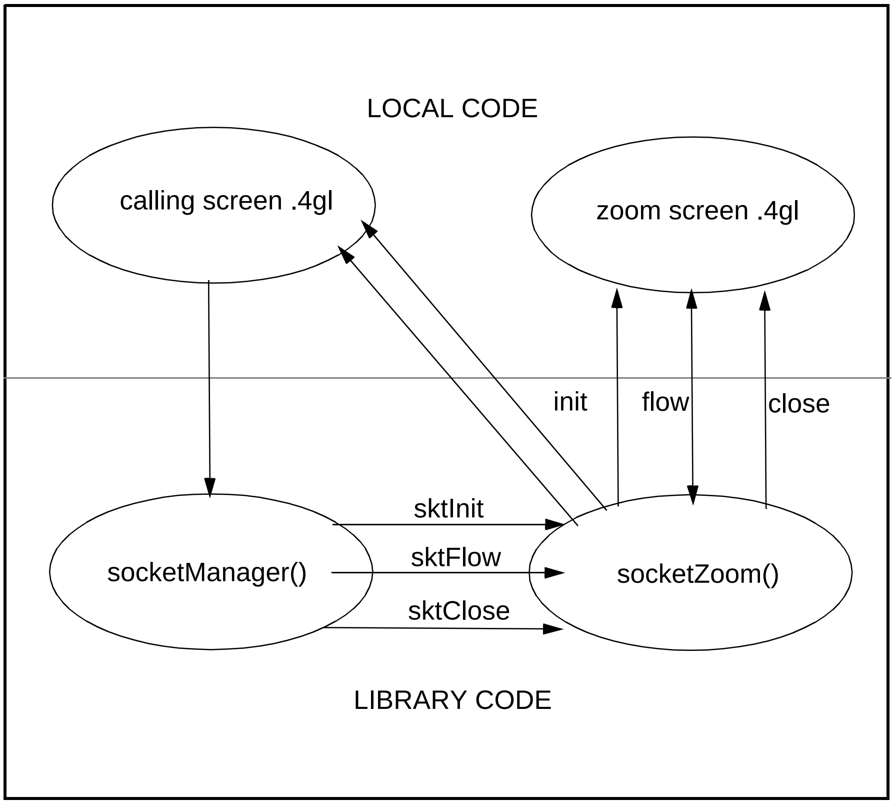
Three zoom .per files have been included as part of the screen demo 3 application. The `cust_zm` file is shown here as an example of the format shared by all three Zoom form specification files.

The `cust_zm.per` file:

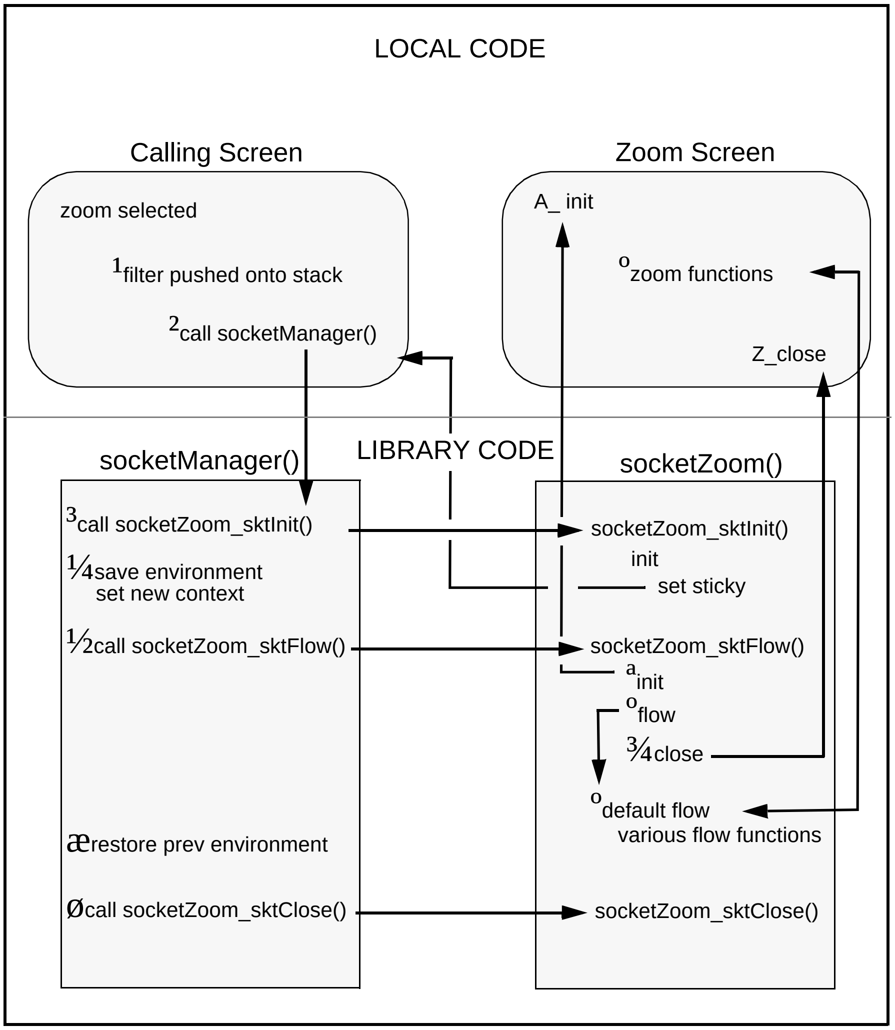
```
DATABASE standard

SCREEN
{
 CustNum   FirstName        LastName         Company
-------------------------------------------------------------------
[f01    ] [f02            ][f03            ][f04                 ]
[f01    ] [f02            ][f03            ][f04                 ]
[f01    ] [f02            ][f03            ][f04                 ]
[f01    ] [f02            ][f03            ][f04                 ]
[f01    ] [f02            ][f03            ][f04                 ]
[f01    ] [f02            ][f03            ][f04                 ]
}

TABLES
  customer

ATTRIBUTES
f01   = customer.customer_num;
f02   = customer.fname;
f03   = customer.lname;
f04   = customer.company;

INSTRUCTIONS
screen record s_custz[6](customer.customer_num, customer.fname,
    customer.lname, customer.company)

DELIMITERS "  "

{
#####################################################################
FGSS
#####################################################################

defaults
    module     = ar
    scr_id     = cust_zm
    type       = zoom
    location   = 10,4
    attributes = border,red
    returning  = customer_num

input 1
    arr_max    = 100
    order      = company
    key        = customer_num
}
```

Zoom Logic

The following diagram represents the basic flow describing what happens when a Zoom is called. The arrows indicate function calls.



The following diagram displays a bit more detail about the order that the zoom events occur.



The following are the local zoom screen functions:

`Ascr_id():` (Init) Initializes variables and pulls up the zoom screen.

`Kscr_id():` (Key) Returns name of main table and fields that build a unique key.

Q*scr_id*(): (Query) Obtains selection criteria using the INFORMIX-4GL CONSTRUCT statement.

R*scr_id*(): (Read) Builds and executes the SQL statement and fills the p_record array for the Zoom.

D*scr_id*(): (Display) Displays p_record array for user and waits for user to select. Handles logic for pressing [ESC], [DEL], [TAB], or event processing.

Z*scr_id*(): (Close) Closes screen, assigns resulting field value to respective p_record field. It does not display this value.

Refer to "Diagram of the switchbox() function:" on page 11-30 to see how each of these single-task functions interacts with the sole tailored function that calls them. Notice that just like switchbox and lib_screen, switchbox and the tailored function act merely as "routers" of requests from libraries to perform each individual task that makes up a Zoom. The Switchbox diagram also shows the scr_functs that are passed via switchbox to initiate these single-task functions.

Each Zoom has its own scr_id and tailored function. When a Zoom is initiated, switchbox routes the requests not to the "default" data entry screen but to the zoom screen (based on the value of scr_id). The tailored function is then called and again routes the requests. Requests are routed by calling the single task function based on the value of scr_funct.

A tailored function is very similar to lib_screen; a chief difference is that lib_screen is a library function and a tailored function is generated in local code.

Zooms are usually initiated as part of a case statement in llh_event/lld_event (within an input statement). Several library functions are called from the function Zoom and control works its way down to lowlevels via switchbox.

Note

If you want to be able to use the Freeform Notes feature ([CTRL]-[n] for notes) on each line of your Zoom, then you must display all key fields for that table on the zoom screen. The key fields must be defined in the following sections of your zoom.per form: SCREEN, ATTRIBUTES, and INSTRUCTIONS. If all key fields are not displayed on the screen, then you must manually change the `K`*`screen_id`*`()` function to return the correct key field for that table based on the line that the user is on.

Creating a Permanent Zoom Filter

You can attach an initial filter to a zoom statement to define the complete set of rows selected when the zoom is first executed. If no zoom filter is provided, the user first sees a "Find" (query-by-example) version of the form, which allows the user to enter search criteria to limit the rows selected. However, if the zoom has an initial zoom filter, that filter is disregarded if the user pressed [TAB] to re-sort and re-select zoom rows.

You can define a permanent zoom filter, often refered to as a *sticky* filter, which defines a subset of data that can be accessed through the zoom, even when the user enters search criteria. That is, when the user enters patterns with the Find command of a zoom form, the zoom selects data that matches the entered criteria AND the criteria of the permanent zoom filter.

To demonstrate the permanent zoom, you can use the Screen Demo applications as an example. Type `scr_demo 3` to place yourself in the `screen3.4gs` program directory with sample *.per files which contain zooms. Run the Code Generator with the command `fg.screen -o1 -yes`. When the code has been generated, locate the function `llh_zoom_filter()` in the `header.4gl` file (e.g. with the vi editor). The code should look like the following:

```
######################################################################
function llh_zoom_filter()
######################################################################
# This function puts the persistent filter on fgStack
#
    #_define_var - define local variables
    define
        tmp_filter char(512) # Temporary string for building the filter

    #_filter - Set the filter based on the current field
    case
      #_when_false - Just to satisfy 4GL syntax
      when false
        #_false - Code placed here will never execute
      #_otherwise - No condition satisfied so execute this logic
      otherwise
```

```
            #_no_filter - No persistent filter for zoom being called
            let tmp_filter = null
        end case

        #_push_filter - Push the sticky filter onto fgStack
        call fgStack_push(tmp_filter)

    end function
    # llh_zoom_filter()
```

Note that there is already a local variable that can be used for building the filter that you will send. Note also that the filter will be sent through the `fgStack_push()` function. In this example, the selection for the customer zoom is limited to customer numbers less than 110. In order to do this you need to build an .ext file with block commands for inserting your code.

Create and edit a `header.ext` file (e.g. `vi header.ext`). Enter the code below, then quit and save this file.

```
    start file "header.4gl"

    before block llh_zoom_filter otherwise
      #_when_in_customer_num - Are we in the customer_num field?
      when infield(customer_num)
        #_customer_num_filter - Set the sticky filter
        let tmp_filter = "customer.customer_num < 110"
        ;
```

After creating the `header.ext`, create a `base.set` file that contains the name of the .ext file.

Next, merge and compile the new code with the `fg.make` command, then run it (e.g. with `fglgo *4gi`). Select the Update or Add option and press [CTRL]-[z] in the Customer No. field. When the zoom Find screen appears, press [ESC]. Note that the zoom form contains only those customers for which `customer_num` is less than 110. (To verify that more rows are in the customer table, you may use SQL query utilities or `scr_demo 1`.)

The AutoZoom Feature

Zoom logic generated by the Code Generator for character fields automatically includes a useful short-cut for selecting values. The AutoZoom feature is an accelerated version of the regular Zoom feature. It allows users to quickly narrow the selection of records that appear on the standard zoom form.

The AutoZoom takes effect when an asterisk (*) is entered into a character field that supports Zoom functionality. The Zoom feature is automatically invoked to display all rows that match the criteria specified in the field. For example, if you enter "sta*" into a field with AutoZoom, all rows that begin with sta are instantly displayed on the standard zoom form.

Note

The field you are auto-zooming from must have the same column name as the one you are Zooming into.

In the case where the column that you are Zooming from isn't the name of the column in the table you are Zooming to, you must put `noautozoom` in the zoom definition of the .per file. If you want an AutoZoom on a field that has a different column name than the column being Zoomed into, you must put it in the "`after field <myfield>`" logic.

`autoZoom` is considered after field logic, thus an `autoZoom` is initiated in `llh_a_field/lld_a_field` (within an input statement). The function `autoZoom` is called to start the AutoZoom.

From here the same single task functions are called by the Zoom tailored function. The only single task function that is never called is `Qcust_zm`, because there is no need to query the user since the user provides filtering criteria at data entry time. Even if the user enters nothing or the value is not in the list of valid data, the AutoZoom is still performed and pulls up all valid values.

The `autoZoom` function builds the SQL statement by putting the user's input into SQL format and adding a `matches` clause. A default SQL statement is always appended to this SQL statement in case the user enters nothing or invalid data so at least the SQL statement is executable; *all* valid values are returned if this happens. Additional filtering criteria can be added by assigning the variable `scratch` to an SQL statement before calling `autoZoom` in the input statement. When the `autoZoom` function has assembled the entire SQL statement and placed it all in `scratch`, `autoZoom` calls the `zoom` library function to perform the rest of the Zoom, based on the SQL statement held in `scratch`.

11

Source Code

This section of the Fitrix *Screen* Technical Reference covers the source code created by running the Code Generator on .per form specification files. Topics covered in this section include:

- Basic code design
- Variables used by the Code Generator
- Data flow
- Screen generated program flow
- Ring detail processing
- Switchboxes
- Varargs

Source Code Logic Overview

Application source code has both a physical and logical structure. The physical design describes where and what the source code is. The logical design describes what it does. The following elements of source code are discussed in this chapter:

Code Design: This section covers the basic design of code generated with the Code Generator. You learn how Fitrix *Screen* generated programs utilize libraries to reuse generic functions.

Code Structure: Examination of the source code for the created application begins by looking at the physical nature of the source code: where it is stored and in what files. This part of the source code is simple and straight forward involving directory structure and file naming conventions.

Data Flow: More complicated is the logical nature of the code. The first part of this section looks at how information moves through the application. This discussion first goes through the variables that are used to hold the application's data as that data moves through the various stages of processing. This section looks at how those variables are named and how their values change as information comes from the data entry forms. The main functions in which these data variables are stored and the trigger points at which you can manipulate the data are identified.

Function Flow: After looking at data flow, the functional program flow is discussed. This section steps through the various functions that go together to make up the application. This process analyzes how library and generated functions work together to create the application the user sees on the screen. In moving through these functions, you learn what the application is doing where and how the triggers fit into this process.

Switchbox Function: The local `switchbox()` joins local screens together by transferring calls from the global libraries to other screen forms that are connected to the main screen. The global code only knows about the main data entry screen and all of the ring menu commands work directly through it (global code also knows about the browse screen, which is really a different version of the main screen). All other screen forms are tied to the main screen in one way or another. While the global libraries make calls that affect these secondary screen forms, these calls are generic requests to read, display and so on that are routed to the appropriate screen form and the data on it through the `switchbox()`.

Vararg Functions: The vararg functions allow you to pass a variable number of arguments between functions. The last section in this chapter covers the vararg family of functions and how they are used throughout the generated code.

The Basic Code Design

The unmodified generated application handles information input and output in a standardized way, offering a consistent user interface. Screen forms developed using the Fitrix *Screen* Form Painter are used to take input from the user and to store and view the data in the database. When using generated applications, you can access a ring menu of commands that allows you to add, update, find, delete, or move through the various documents displayed on data entry screens. The ring menu commands available on the data entry screen are designed to directly access the information on the main header/detail type screen forms. Other types of screens: browse, zoom, extension, and add-on screens created by the Form Painter, *plus* any other kind of custom screen you might add, are all tied to the information displayed on that main screen in one way or another.

The source code that makes up the final application consists of two parts:

Local Code: The most visible part is the code generated from the forms created with the Form Painter. These forms are stored in your local directory. This local code is built automatically after defining the application characteristics with the Form Painter, and it contains all the specific references to the data entry forms, database tables, and the program.

The local, generated functions themselves are designed as a "white box" so that you can understand how they work so that you can modify them. However, the programs are designed so that most (and hopefully all) modifications need not be made directly in the generated source code. Since this generated source code contains all types of references to the screen forms and data tables you are using, that generated code must be able to change when your data tables and screen forms are changed. Since applications evolve over time, we have made generated applications regenerable so that, as the screen forms and database tables change, the tool can make the changes to the code for you.

To accommodate your need to make enhancements to local code and to simultaneously preserve regenerability, we have built *triggers* into both the Form Painter and the Code Generator. Triggers are predefined areas in the code to which you can make additions. The Code Generator then merges your custom triggers into the code at the appropriate points as it is regenerated. This allows you to create, for example, after field processing for a specific data entry field on your screen and, if your screen changes, preserve that processing in the regenerated code. For more information on triggers refer to the next chapter "Customizing Your Base Program With Triggers" on page 12-1.

Since you can make almost any type of modification through the use of triggers, you don't actually have to modify *any* source code, either generated or library code, to modify your applications. However, you will find a more complete understanding of the code design almost invaluable in modifying the application. While you can add code at triggers without understanding the overall structure of the source code that drives the application, it is a little bit like trying to make a meal blindfolded: it is unnecessarily difficult. A good understanding of the design and flow of the source code makes writing successful triggers much easier and more immediately rewarding.

Library Code: The other less visible part of the source code resides in various libraries that are linked to the local code to create the application the user sees at runtime. The functions in the libraries are designed to be used unaltered by the generated application. Since the focus is on reusing as much code as possible, Fitrix *Screen* design moves as much work as possible from the generated files in the local directory to the generic functions in the libraries.

The library code itself is also divided into two parts:

Screen: The code in the screen library (`$fg/lib/scr.4gs`) contains the basic templates for document maintenance commands (add, update, delete and so on) that the users access from a ring menu. This code controls much of the program flow as the users access these various commands, calling the local, generated, I/O type commands as needed.

Standard: There is also code in the standard library (`$fg/lib/standard.4gs`), which contains utility commands that do a variety of tasks, such as passing variables. These functions are called both by the screen library functions and the locally generated application specific functions.

When understanding the basic design of Fitrix *Screen* code, it is unnecessary, for the most part, to know exactly how the library functions work. It is best to think about them as "black boxes" that are called upon to produce a certain result. Later, this chapter describes the program flow through these black boxes, but only because you should think of these various functions as resources you can draw upon in making enhancements or additions to the generated application.

For example, if you want the screen to redraw itself at a certain point, all you need to do is call the `ring_refresh()` function passing it the current rowid. You need not know how this redraw is accomplished.

At no time should you consider modifying the library functions themselves to produce a specific result. The code is designed so that everything a programmer might normally want to control in the source code can be controlled from the application specific code in the local library. Except for certain well-defined situations, you should not need to alter any global functions to get the applications to behave as you desire. When you need a global function to behave differently within a specific application, the code is designed so that you can make a copy of the library function in your local directory and, if you use the same function name, the local version of that function is linked in instead of the global one.

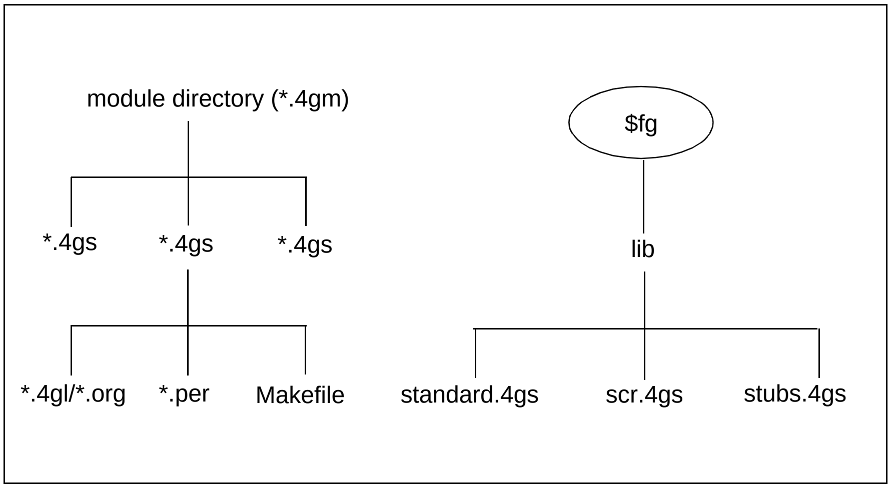
Though this is rarely done with most library functions, there is a class of library functions that developers are expected to commonly replace with local functions of the same name. For these types of functions, there are *stub* functions (functions that are called and exist at the library level, but which do very little if anything) at the library level, but they exist just to allow you a level of control over the way the library behaves.

For example, the `ok` functions such as `ok_add()` and `ok_delete()` are used to allow or disallow the addition or deletion of documents from the ring menu under specific conditions. In this case, these functions are called before a document is added or deleted. Because they exist on the library level always allowing document additions or deletions, these functions are not generated. If you wish to control this upper level file maintenance command, you simply add these functions to your local code, putting whatever tests you desire to take place before the Add or Delete command.

Code Structure

Apart from the Makefile (see the section "The Makefile" on page 14-12), all files created by the Code Generator contain uncompiled INFORMIX-4GL code.

Data-entry applications must use the following directory structure.



Fitrix *Screen* creates only the code appearing within the .4gs program directories, although the application utilizes executables and pre-compiled libraries installed with Fitrix *Screen*.

The library source code is covered after the discussion on program directory source code.

Program Directory Source Code

As mentioned previously, the Code Generator creates source code files within the module directory that contains the .per form specification files. An example of the type and purpose of source code files created can be found after running the Code Generator in the `screen3.4gs` demo application directory.

The following .per files provide all of the instructions necessary to create a basic data entry program for customer orders. The `screen3` application directory contains the following .per files:

- **`order.per:`** contains instructions passed to the Code Generator for the main header/detail form.
- **`browse.per:`** adds a document browse screen to the application.

- **custzm.per**, **stk_mnu.per**, and **stockzm.per:** add Zoom features to particular fields on the main screen.

Refer to "Creating Screen Forms" on page 10-1 for information on screen forms.

After invoking the Code Generator on the .per files listed above, the following source code files are created in the screen3.4gs program directory:

- **Makefile**: This file is created to facilitate the compilation of source code. It determines variables for the real make, stored in $fg/Make. This local Makefile is used to compile source code that has been modified since the last compilation. Source code compilation is discussed in "After Code is Generated."
- **globals.4gl**: This file defines all global variables. A change to this file causes make to compile all .4gl source files.
- **main.4gl**: This code initializes variables and windows before calling one of the following library functions. The ringMenu_start(1) function is called if the data-entry screen is a flat file, or ringMenu_start(2) if the screen is header/detail.
- **midlevel.4gl**: This file contains all middle-level source code. Cursor control logic and some data validation occurs at this level.
- **header.4gl**: This source file stores low-level code used to handle the data in the header portion of the data-entry form. This code is present in both flat file and header-detail applications.
- **detail.4gl**: This file contains low-level code for handling detail lines in header/detail application screens. When no detail section is specified in the .per form specification file (flat file—header only applications), no detail.4gl file is created by the Code Generator.
- **options.4gl**: This source code is used to add further functions to the ring menu of commands that appear at the top of the data-entry form.
- **browse.4gl**: This source code file is only created when the browse.per file is specified prior to code generation. Code in browse.4gl controls the opening of the browse window and the display of data within the window.
- **cust_zm.4gl**: This file is created as a result of specifying the cust_zm.per file prior to code generation. It contains logic used to control the Zoom window, and the data appearing within the window.

- **`stk_mnu.4gl`**: This file is created as a result of specifying the `stk_mnu.per` file prior to code generation. It contains source code required for the manufacturer code Zoom function in the orders example.
- **`stockzm.4gl`**: This file is created as a result of specifying the `stockzm.per` file prior to code generation. It contains logic used to control the item code Zoom function in the orders example.

Library Source Code

Fitrix *Screen* differs from other application creating tools in its use of 4GL libraries. Libraries, as used by Fitrix *Screen* generated code, consist of functions that are pre-compiled and grouped together under a directory with other similar functions. A library function should be created when a generic data-independent task can be used by more than one program module.

The Code Generator libraries produce two desirable conditions: they allow many programs to share common code, and they serve as the foundation for any CASE-generated system. Libraries save time for both initial development and maintenance.

The Code Generator uses the following libraries located in `$fg/lib`:

- **`scr.4gs:`** This directory contains the uncompiled source for library functions for Fitrix *Screen* only. The source code in this directory is used by programs generated with the Code Generator.
- **`scr.a:`** This is the pre-compiled collection of screen library functions for systems that run INFORMIX-4GL and not RDS.
- **`scr.RDS:`** This is the pre-compiled collection of screen library functions for systems that run RDS rather than INFORMIX-4GL
- **`standard.4gs:`** This directory contains the uncompiled source for library functions. The source code in this directory is used by Fitrix *Screen* and Fitrix *Report*, and can also be used by other programs.
- **`standard.a:`** This is the pre-compiled collection of standard library functions for systems that run INFORMIX-4GL and not RDS.

- **`standard.RDS:`** This is the pre-compiled collection of standard library functions for systems that run RDS rather than INFORMIX-4GL.
- **`stubs4gs:`** This library contains the source code for systems that do not have the Enhancement Toolkit installed. The stub functions are installed if the Enhancement Toolkit has not been purchased. The functions are then compiled into the `user_ctl` directory (discussed next). When the Enhancement Toolkit is purchased, it replaces the compiled null functions in the `user_ctl` directory.
- **`stubs.a:`** This is the pre-compiled collection of user control library functions for systems that run INFORMIX-4GL and not RDS.
- **`stubs.RDS:`** This is the pre-compiled collection of user control library functions for systems that run RDS rather than INFORMIX-4GL.
- **`user_ctl.a:`** This directory contains the pre-compiled User Control Libraries if they have been purchased. Otherwise, it contains the pre-compiled null functions. This directory is found on systems running INFORMIX-4GL and not RDS.
- **`user.ctl.RDS:`** This directory contains the pre-compiled User Control Libraries if they have been purchased. Otherwise, it contains the pre-compiled null functions. This directory is found on systems running RDS rather than INFORMIX-4GL.
- **`forms:`** This directory stores the .per forms and compiled .frm files used by library functions.
- **`tags:`** This file stores the list of function call dependencies. This list is used by the hypertext feature made part of Fitrix *Screen*-generated code upon compilation. This file is read as you move through the source code function-by-function using the tags feature. For more information refer to "The Tag Utility" on page A-5.

Code Design Levels

Source code generated with the Code Generator can be categorized into three levels: upper level, midlevel, or low level. The classification of code depends on the function it performs. This method of organization helps make the code easier to understand and modify.

Upper Level Functions: Upper level code includes library functions, menuing logic, and flow control logic. Upper level code contains no local logic and does not reference specific databases, tables, or columns. It can be considered generic. Upper level code calls on midlevel code and low level code, leaving the specifics to these other two classes of source code. The following are some of the upper level functions.

Function	Purpose
`ring_options()`	controls the options ring menu option
`ring_add()`	controls the add ring menu option
`ring_update()`	controls the update ring menu option
`ring_delete()`	controls the delete ring menu option

Midlevel Functions: Midlevel code interacts with the upper level code to control the function of command line or ring menu commands. This class of source code includes functions that make reference to the database (but are not usually modified), and local code not found in the library functions. Generally speaking, midlevel functions give program specific information to control database access for the program. Midlevel code is identified by the first part of the function name: `mlh_` for header functions, `mld_` for detail functions, and `ok_` for functions that control upper level code.

The `ok_` functions allow for control over pre-compiled functions of the data-entry ring menu options. These functions can be defined within the `midlevel.4gl` file in the local source directory for any of the ring menu commands, such as add, update, and delete. When defined in `midlevel.4gl`, the function returns a value of true or false, which determines whether the ring menu command is executed.

Function	Purpose
`mld_clear()`	clears the program variables
`mlh_cursor()`	cursor handling for header screen
`ok_add()`	controls upper level add
`ok_delete()`	controls upper level delete
`ok_update()`	controls upper level update

Low Level Functions: Lowlevel code passes data between screen record and database. Data validation occurs in low level logic as well. Functions in this class act as a pipeline between input to the screen (by way of the screen array(s) specified on the .per form specification file) and information stored in database tables.

Lowlevel code is found in the local source directory, and is commonly modified. Low level code can be identified by name, since functions containing this code begin with the characters `llh_` (for header) or `lld_` (for detail).

Function	Purpose
`llh_input()`	Input logic for header part of screen.
`lld_showline()`	Displays screen record variables onto screen line.
`lld_p_prep()`	Creates screen record array.
`lld_read()`	Reads in array elements from disk.
`lld_add()`	Inserts data into detail table.

Code Generator Variables

Certain variables are created in the Code Generator libraries and the `globals.4gl` source code file by the Code Generator. These variables are relied upon by a number of functions generated as part of the application.

This section of the documentation provides a list of such variables as well as a brief explanation of the purpose they serve.

There are several categories of variables that are used extensively throughout the Code Generator: global variables used by the program only, global variables used at the library-level, static variables, and local variables.

Some of the most useful global and static variables are discussed next.

Global Variables Used by the Program

Global variables can be used anywhere within a program. The following generated variables are put at the very end of the `globals.4gl` file in the Library communication area. They are used for communication within libraries and between local code and libraries. These variables must be located at the very end of the globals section.

Variable	Type	Purpose
`progid`	`char(17)`	Stores identification of the current program.
`scr_id`	`char(7)`	Stores the current screen identification.
`scr_funct`	`char(20`	Stores name of current screen function being run.
`sql_filter`	`char(512)`	Stores filter portion of an SQL statement.
`sql_order`	`char(100)`	Stores order portion of an SQL statement.
`menu_item`	`char(10)`	Current ring menu option being performed.
`input_num`	`smallint`	Stores current input section within screen.
`p_cur`	`smallint`	Stores current input array element.
`s_cur`	`smallint`	Stores current screen array element.
`scr_fld`	`char(40)`	Stores name of current screen field ("table.column" format).
`nxt_fld`	`char(40)`	Stores name of programmatic next field.
`prev_data`	`char(80)`	Stores the data in the field prior to entry.
`this_data`	`char(80)`	The data currently entered into field.
`data_changed`	`smallint`	Indicates whether the field data changed.
`hotkey`	`smallint`	Identifies hot key pressed.
`scratch`	`char(2047)`	Provides "scratch pad" for communication between functions.

Global Variables Used at the Library-Level

Another category of variables is library-level variables. These are status variables that hold data about the program and activity happening at the moment. These variables answer questions such as:

- What version is running?
- Can the user update user definable fields?
- What is the current rowid?
- How many element were retrieved in a Find, etc.?

Library-level variables are located in the file `$fg/lib/standard.4gs/scr_lib.4gl`. These static variables are local to the screen library functions, yet they can be accessed from anywhere in the application. The `scr_lib` family of functions, located in the same file, maintains these variables. The library function `put_scrlib` loads these variables and `get_scrlib` returns the values of these variables. All `scr_lib` variables are defined as `char(80)`. Here are the library-level variables used with the scrlib functions:

Variable	Purpose
`dbname`	Stores the name of the current database.
`version`	Stores current version of the generated code.
`module`	Stores the name of the module.
`language`	Stores current language being used.
`scr_type`	Stores the current type of screen.
`curs_pos`	Tracks the current "Find group" position.
`curs_count`	Stores number of elements in "Find group."
`curs_rowid`	Stores rowid of the current document.
`num_rows`	Stores number of rows in the current window (1-24).
`num_cols`	Stores number of columns in the current window(1-80).
`scr_tab`	Stores the name of the main screen table.
`fld_tab`	Stores the name of the current screen field’s table.
`scr_key`	Stores the unique key for the screen.
`auto_udf`	Determines whether user automatically updates user-defined fields (y/n). Part of user control package.
`auto_note`	Determines whether user automatically updates freeform notes form (Y/N). Part of user control packages.
`auto_answr`	Automatically answer Y/N to all prompts.
`scrn_tier`	Level of screen tier.
`scrn_trx`	Determines whether to commit or rollback when [DEL] is pressed.

The functions `put_scrlib()` and `get_scrlib()` are used to maintain static variables that are used by the library functions. They are intended for use by the library functions, but they can be accessed from anywhere in the application to check on the status of the library functions.

The syntax of the `put_scrlib()` and `get_scrlib()` functions:

```
put_scrlib("variable","value") —— inserts "value" into the library variable
passed as "variable."

get_scrlib("variable") — returns the current value stored in that
"variable."
```

Example:

```
call put_scrlib("version","4.1")
let version_num = get_scrlib()
```

In this example, a call to `put_scrlib("version", "4.1")` is made from within the `main()` function. This identifies the version of the Code Generator that generated the local code.

It is the responsibility of each library function to maintain backward compatibility with older versions of generated code.

Static Variables for Header/Detail Forms

The following static (module) variables are used by `header.4gl` and `detail.4gl`. These variables maintain their values only within the .4gl file they are defined in.

Variable	Type	Purpose	header.4gl or detail.4gl
`lookup_prep`	`char(1)`	Have the lookups been prepared?	both
`select_prep`	`char(1)`	Has the select statement been prepared?	both
`dup_prep`	`char(1)`	Has the duplicate check been prepared?	`header.4gl`
`defaulted`	`char(1)`	Has the defaulting been done?	`header.4gl`
`exit_level`	`smallint`	0=input, 1=field	both
`tab_pressed`	`smallint`	(boolean) was the tab key pressed?	both
`insert_prep`	`char(1)`	Has the insert statement been prepared?	`detail.4gl`
`del_flag`	`char(1)`	Is this an insert after a delete?	`detail.4gl`
`in_insert`	`smallint`	(boolean) true if we’re in insert row	`detail.4gl`

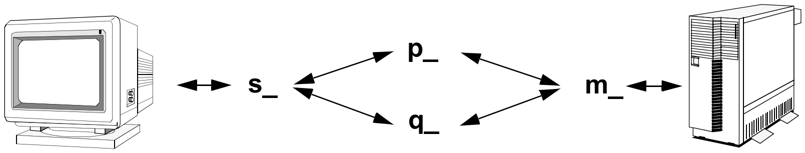
Data Flow

The key to understanding a Fitrix *Screen* generated program is to understand how information flows from disk to screen and from screen to disk. This flow was designed to simplify modification of generated program.

The Data Variables

To understand how information moves within the data entry applications, you first have to know a little about the data variable structure within which the generated code stores data variables. There are different sets of these variable records for both the header and detail sections, which are initially defined in `globals.4gl`. These records create a pipeline for moving records back and forth between tables and screen. These records are manipulated by midlevel and lowlevel code.

These variable records (variables defined to parallel screen and database records) are called the `m_` (map), `p_` (picture), `q_`, and `s_` (screen) records because of the naming convention they follow. If the name of the database table is `yyyxxxxzz` and its fields are `yyyxxxxzz.field1` and `yyyxxxxzz.field2`, then the name of the `m_` record would be `m_xxxxzz` and the name of the `p_` record would be `p_xxxxzz`. The last six characters of the table name are used.



The most important variables are the screen and disk variable records.

p_ record: The `p_` (picture) record parallels the data elements defined on the screen. The `p_` record only contains those fields displayed on your data entry screen.

m_ record: The m_ (map of data table) record parallels the information in the data tables. The m_ record contains all of the same columns as the database table. The m_ record is sometimes referred to as the disk record.

Note

INFORMIX-4GL has a table naming convention that requires the first eight characters to be unique. The Code Generator requires that the last six characters of those first eight characters also be unique.

Any input data from the screen is first validated and stored in a corresponding p_ record. From the p_ records the data is formatted to fit the tables by moving the data into m_ records. The m_ records look like the tables.

These p_ and m_ records contain data from either screen to database or from database to screen and are found in only two .4gl files, header.4gl or detail.4gl.

The m_prep() function transfers the data from the p_ records into records that look like the tables. These records are named with m_ and the last six characters of the table. Since the rows of the detail line table are transferred one-by-one, there is only one m_ record for detail lines instead of an array.

Similarly, the p_prep() function transfers data in the opposite direction, filling the screen-like p_ records with the contents of the table-like m_ records whenever data is read from storage and displayed on the screen.

For detail rows, the m_ record is a single record, mapped to the row in the database, but the p_ record is an array with as many elements as you have defined to allow in the rows of your documents. This is usually a hundred or more. It *isn't* limited to the number of lines on the screen itself.

q_ record: The q_ record contains all columns for the table not displayed on the screen. The q_ record follows the same naming convention (q_xxxxzz).

A special file exists in $fg/codegen/options/screen.opt that allows you to control how the q_ record gets generated in the header.

If the non_scr_q_elems variable is set to "exclude," then you must add the q_ records you want with triggers.

If the `non_scr_q_elems` variable is set to "include," then `q_` records automatically get created for all columns not defined on the screen.

For more information on the `non_scr_q_elems` variable see "The Code Generator Options File (screen.opt)" on page 2-21.

The `q_` record for the detail rows is also an array containing all the rows you have defined as part of the detail section of the document.

Other data variables: A number of variables are used to track the detail rows on the screen. For displaying the detail section of the screen, the system also keeps the variable `scr1_max`, which tells the program how many lines are displayed in the detail section of the screen. This variable is used, for example, in function `lld_display()`, to display all the lines of detail to the screen. The variable `rec1_max` is used to keep the total number of records in the detail section stored on the disk, so that when a read is done, it checks this variable to make sure that there is a record there. The variable `rec1_cnt` is used to keep track of the last of those disk records stored in the display array. The variable `p_cur`, is the array number of the "current" detail line.

s_ record: There is another set of screen variable arrays that are not defined anywhere in the program but in the .per file itself. This screen record is named for the screen itself. If the screen's name was "screen," the name of the screen record for the header would be `s_screen` and the name for the screen record for the detail section would be `s_dscreen`. The only time these variables are mentioned in the program is when information is being displayed or input from the screen. These `s_` records always interact directly with the `p_` records.

MEMORY TRICK: If you find yourself getting confused about the meaning of the `m_ p_ q_` and `s_` records, try this:

`m_` means "Map of data table."

`p_` means "Picture" - as in what is on the screen.

`q_` means "data not seen on screen." This is the complement of p_.

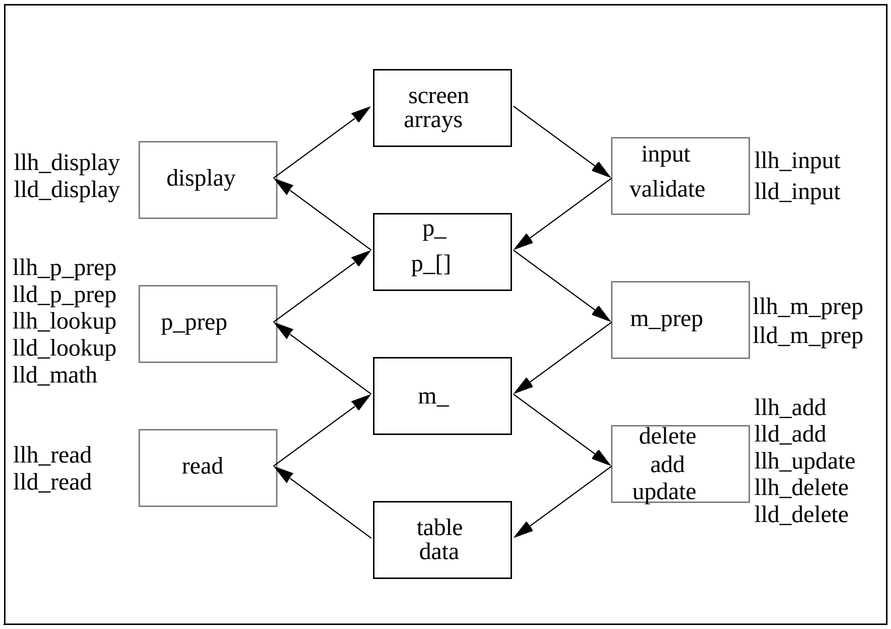
`s_` means "Screen."

Data Flow Through Variables

Most of the flow between these various variables that carry the data takes place in very few places. It works like this:

	Low Level Header	Detail Functions
From Disk to Screen		
`From disk to m_ variables`	`llh_read()`	`lld_read()`
`From m_ variables to p_ variables`	`llh_p_prep()`	`lld_p_prep()`
`From p_ variables to s_ variables`	`llh_display()`	`lld_display()`
From Data Entry to Disk		
`From s_ variables to p_ variables`	`llh_input()`	`lld_input()`
`From p_ variables to m_ variables`	`llh_m_prep()`	`lld_m_prep ()`
From `m_` variables to Disk		
`To create a new row`	`llh_add() (new)`	`lld_add()`
`To update a row`	`llh_update ()`	`(none: deleted & added)`
`To delete a row`	`llh_delete()`	`lld_delete()`

The flow of data input and display and associated lowlevel functions may be represented by the diagram below:



In the following section concerning the program flow, you see how these various functions fit into the entire flow of the program, but here they are always "low-level" functions: called by other functions simply to move data. This is done at various times in different ways depending upon what the user and the programmer are trying to accomplish. Using the standard Add, Find, Update, Delete and other commands, the flow follows the basic to disk and from disk patterns. The Add command documents flow is from screen to disk. The Find, Tab, Next, Prev, Browse and Tab (view detail lines) commands all have information flow from disk to screen. The Update command uses both from disk to screen and from screen to disk. You have to first Find a document before you update it. The Delete command is a special case since it removes information, but information here basically flows from the user (if not the data entry screen) to the disk.

The point of having all these different variables is so that, at any time in the application, you can use both disk data and screen data independent of each other and so that you can work in between the various data transformation processes. Triggers allow you to deal with the data flow without changing the generated programs. They give you specific points in the program where you can manipulate data flow between variables.

Triggers in Data Flow

To change the information that comes in off of the screen to affect its display, you have pass through these triggers. There are actually two sets of each of these triggers: one for the "header" information and one for the "detail" information, but since they all basically work the same, except that detail information has to be put into a larger array, each one is listed only once:

From Disk to Screen Triggers

Header section:

Trigger	In Function	Happens After	Happens Before
on_disk_read	llh_read()	from "s" to "m" movement	llh_p_prep()
on_screen_record_prep	llh_m_prep()	from "m" to "p" movement	llh_display()

Detail section:

Trigger	In Function	Happens After	Happens Before
on_disk_read	lld_read()	from "s" to "m" movement	lld_p_prep()
on_screen_record_prep	lld_m_prep()	from "m" to "p" movement	lld_display()

The flow here is more complicated, largely because during user input, you have lots of different points at which things can be controlled. In many ways this is the most complicated part of the program, but this discussion should help to simplify it. The asterisk indicates flows where the before or after situation is *sure* to happen, but in which the situation may not have happened immediately before or after. For example, `after_field` will always eventually be followed by `after_input`, but an `after_change_in` or another `before_field`, may intervene.

From Screen to Disk Triggers

Header Section:

Trigger	In Function	Happens After	Happens Before
on_event	llh_event()	user presses key	anything
before_input	llh_input()	lld_default()	from "s" to "m"
before_field	llh_b_field()	before_input *	after_field
after_field	llh_a_field()	before_field	after_input *
after_change_in	llh_a_field()	after_field	after_input *
after_input	llh_input()	after_field *	llh_m_prep
on_disk_record_prep	llh_m_prep()		on_disk_add
	after_input()		
llh_add	llh_m_prep()	on_disk_update	llh_update()
llh_m_prep	on_disk_delete()	llh_delete()	delete verification

Detail section:

Trigger	In Function	Happens After	Happens Before
on_event	lld_event()	user presses key	anything
before_input	lld_input()	lld_default()	from "s" to "m"
before_row	lld_b_row()	before_input *	before_field
before_insert	lld_b_insert()	before_input *	before_field
before_delete	lld_b_delete()	before_row	after_delete
before_field	lld_b_field()	before_row *	after_field
after_field	lld_a_field()	before_field	after_row *
after_change_in	lld_a_field()	after_field	
after_row	lld_a_row()	after_field *	after_input *
after_insert	lld_a_insert()	after_change_in	after_input
after_delete	lld_a_delete()	before_delete	after_input
after_input	lld_input()	after_field *	lld_m_prep()
on_disk_record_prep	lld_m_prep()	after_input	
on_disk_add	lld_add()	lld_m_prep()	
on_disk_update	lld_update()	lld_m_prep()	
on_disk_delete	lld_delete()	user delete verification	

These triggers are involved every place at which you can test for conditions and alter field values. When you use these regenerable triggers, you do not need to replace entire sections of the code or use the `do_not_generate` trigger to preserve those changes during regeneration.

Program Flow

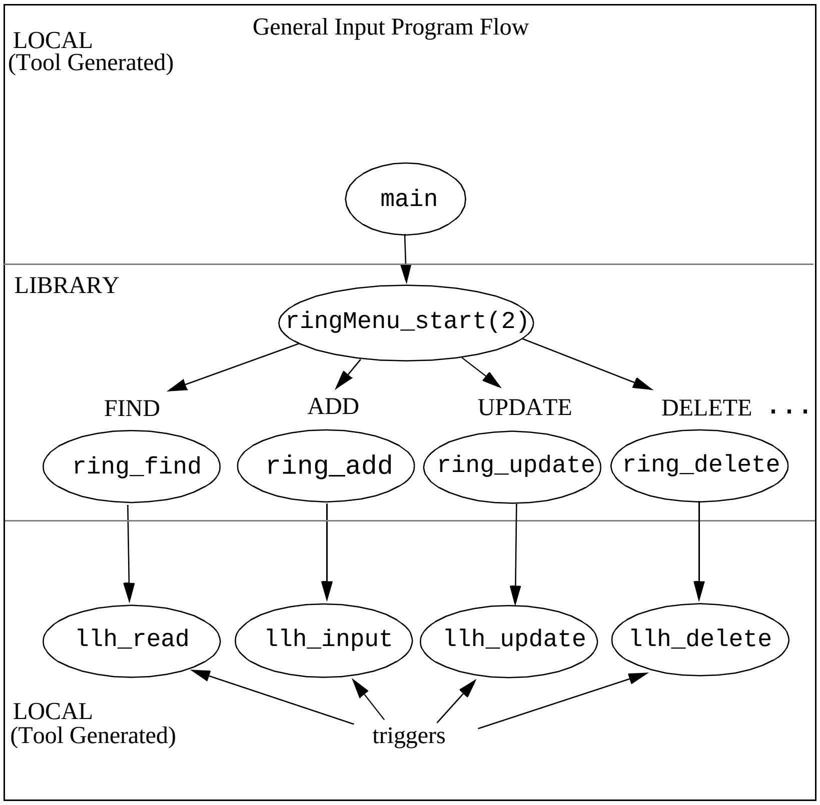
As a developer or programmer, it is important to understand the flow control of source code. This section focuses on the flow of logic in code generated by the Code Generator. The diagrams found in this section are designed to provide additional perspective on how the logic proceeds within an application.

Note

A couple of utilities are provided which help you locate source code. These utilities are especially useful when learning Fitrix *Screen* programs. The tags feature allows you to quickly access and display functions simply by typing their name. Another utility allows you to print the comments for specified functions. For more information on these useful utilities see "The Tag Utility" on page A-5

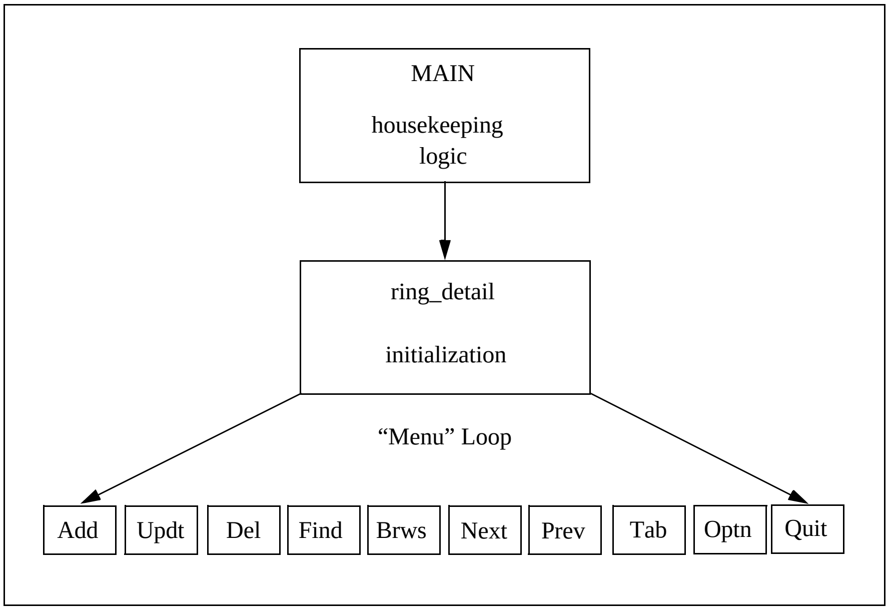
The Code Generator offers a vast improvement in efficiency over manual coding of data-entry applications. It also provides the groundwork for consistency across vertical applications that might (in the absence of the Code Generator) otherwise not be found. This consistency in data-entry applications is clearly identified through the use of generic upper level (ring menu) functions. The prior section on "Code Design Levels" mentioned the fact that these upper level ring menu functions provide a control loop, which in turn controls access to midlevel and low level functions.

The following diagram illustrates the basic flow of an input program.



The following diagram depicts the initial flow upon invocation. The local `main.4gl` file passes control to the `ringMenu_start(2)` function. The `ringMenu_start(2)` function performs initialization tasks and enters an action menu loop. The loop contains logic for the ring commands shown at the bottom of the diagram.

Upper Level Flow:



The Main Program and the `init()` Function

The main program starts on the local level with the file `main.4gl`. This is local, generated code. In general`, main()` first opens the database and the form, and then calls the upper level ring menu functions.

More specifically, `main()` clears the screen, calls the Informix function startlog to start an error log, then stores variables telling the system what version of the program is running and what database the user is accessing using the function `put_scrlib()`. It then opens a window and calls the `init()` function. After calling `init()`, `main()` opens the main screen form and calls the `ringMenu_start(2)` function. At this point, the main program flow is transferred to the `ringMenu_start(2)` function in the screen library.

The switchbox() Function

Object oriented programs are written to execute objects, not functions. Objects are tangible things such as a screen, a menu, or a dialog box. When programming with objects, you cannot always know the specific name for the function that the object represents. The switchbox() determines where to go when you want to execute a function for an object such as a read. More specifically, the switchbox() joins local screens together by transferring calls from the global libraries to the other screen forms that are connected to the main screen. Since the ring menu global code works directly through the main data-entry screen, all local screens must be tied to the main screen. The switchbox() routes generic library requests to the appropriate local or secondary screen forms.

There are two levels of switching: screen(object) level and function level. These are discussed next.

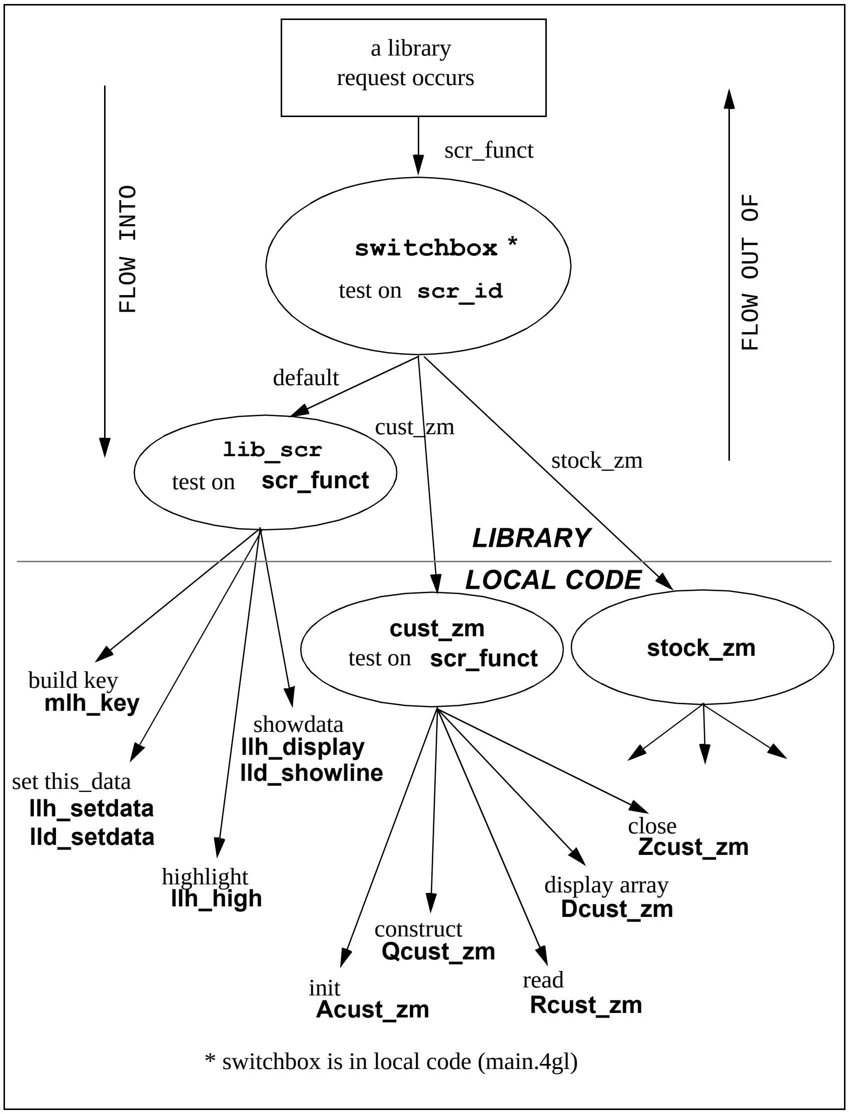
Screen Level Switchbox

The screen level switchbox() resides in main.4gl in a function called switchbox() and accepts requests from data independent library functions. This function's main responsibility is to direct program flow to the appropriate screen. Switchbox() directs control to the appropriate screen handling function based on the value of scr_id. If the current screen being worked on is the header/detail screen (the main data entry screen), scr_id is set to default, and the library function lib_screen() is called. The lib_screen function includes a case statement for matching the value of the scr_id variable and calling subsequent functions based on the particular match. Since Zooms have screens, a call to a Zoom passes through the function switchbox(). The switchbox()function directs control down to the zoom handling functions based on the zoom's scr_id. The global variable scr_id is examined to see what screen is to be handled.

Following is an example of the function `switchbox()`.

```
######################################################################
function switchbox(funct)
######################################################################
# This is the switchbox function for version 4.11.UB1 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
      when scr_id = "cust_zm" call cust_zm()
      when scr_id = "stockzm" call stockzm()
      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()
```

Diagram of the `switchbox()` function:



* switchbox is in local code (main.4gl)

Function Level Switchbox

Once the program flow has been directed to the appropriate screen, the second level of `switchbox()` is run. This second level is the function level `switchbox()`. Its main job is to direct program flow to a particular local code function. It does this based on the value of the global variable `scr_funct`, which is set above by the library function that called the function `switchbox()`. If the current screen being worked on is the header/detail screen (i.e., the `scr_id` is "default") the library function `lib_screen` acts as the function level `switchbox()` and passes control as follows:

```
######################################################################
function lib_screen()
######################################################################
# This function is the hardcoded switchbox for the default header or
# header/detail screen.
#
    define
        input_type integer,
        ring_rowid integer,
        ring_cursor integer,
        ring_total integer

    # Trap fatal errors
    whenever error call error_handler

    #_get_cursor_info - Get information about the cursor if needed
    if scr_funct = "add" or scr_funct = "update" or
       scr_funct = "delete" or scr_funct = "construct" or
       scr_funct = "browse" or scr_funct = "view"
    then
        let ring_rowid = get_scrlib("curs_rowid")
        let ring_cursor = get_scrlib("curs_pos")
        let ring_total = get_scrlib("curs_count")
    end if

    case
      # New functionality
      when scr_funct = "add"
        #_get_scr_type - Determine if this is header or header/detail
        if get_scrlib("scr_type") = "header/detail"
          then let input_type = 2
          else let input_type = 1
        end if
        call ring_add(input_type, ring_rowid, ring_cursor, ring_total)
          returning ring_rowid, ring_cursor, ring_total
        call ring_border(ring_rowid, ring_cursor, ring_total)
      when scr_funct = "update"
        let ring_rowid = ring_refresh(ring_rowid)
```

```
  #_get_scr_type - Determine if this is header or header/detail
  if get_scrlib("scr_type") = "header/detail"
    then let input_type = 2
    else let input_type = 1
  end if
  call ring_update(input_type, ring_rowid)
  call ring_border(ring_rowid, ring_cursor, ring_total)
when scr_funct = "delete"
  let ring_rowid = ring_delete(ring_rowid)
  if ring_rowid = -1 and ring_total > 0
  then
      let ring_rowid = 0
      call put_scrlib("curs_rowid", 0)
  end if
  call ring_border(ring_rowid, ring_cursor, ring_total)
when scr_funct = "construct"
  call ring_find(ring_rowid, ring_cursor, ring_total)
    returning ring_rowid, ring_cursor, ring_total
  call ring_border(ring_rowid, ring_cursor, ring_total)
when scr_funct = "browse"
  call ring_browse(ring_rowid,ring_cursor,ring_total)
    returning ring_rowid, ring_cursor, ring_total
  call ring_border(ring_rowid, ring_cursor, ring_total)
when scr_funct = "view"
  call lib_message("scroll")
  call mld_scroll()
  let int_flag = 0
  call ring_border(ring_rowid, ring_cursor, ring_total)
when scr_funct = "set sticky"
  if input_num = 2
    then call lld_zoom_filter()
    else call llh_zoom_filter()
  end if
when scr_funct = "set this_data"
  if input_num = 2
    then call lld_setdata()
    else call llh_setdata()
  end if
when scr_funct = "touch"
  # Identify the screen type
  call put_vararg("type")
  call put_vararg("old header/detail")
  # Identify the cursor table, hard filter, and default order
  call put_vararg("cursor")
  call mlh_cursor()
when scr_funct = "highlight"
  if input_num = 2
    then call lld_high()
    else call llh_high()
  end if
when scr_funct = "pwrite"
  if input_num = 2
   then call PW_detail()
```

```
             else call PW_header()
            end if
          when scr_funct = "pread"
            if input_num = 2
              then call PR_detail()
              else call PR_header()
            end if
          when scr_funct = "after_query"
            call mlh_aquery()
          when scr_funct = "math"
            call llh_math()
          when scr_funct = "clear"
            call mlh_clear()
            call mld_clear()
          when scr_funct = "showdata"
            if input_num = 2
              then call lld_showline()
              else call llh_display()
            end if
          when scr_funct = "showarray"
            call lld_display()
          when scr_funct = "build key"
            call mlh_key()
        end case

    end function
    # lib_screen()
```

Note that control is directed to the appropriate function in local code based on the value of the variable `scr_funct` (for example, if `scr_funct` is set to `set this_data` then the local code functions `llh_setdata()` or `lld_setdata()` are called). The global variable `input_num` indicates whether the header portion of the screen is being worked on or the detail portion of the screen is being worked on.

Like `lib_screen()`, each non-default screen such as zoom screens, has its own tailored function that acts as a function level `switchbox()`. This tailored function has the same name as the .per file. One such tailored function may look like this:

```
    ######################################################################
    function cust_zm()
    ######################################################################
    # This is a screen function switching mechanism.
    # It’s job is to route requests from the screen manager
    # to the appropriate local function.
    #
        #_define_var - define local variables
        define
            no_function smallint  # true if scr_funct not in case statement
        #_err - Trap fatal errors
        whenever error call error_handler

        #_flow_init - initialize flags
        let no_function = false

        #_switchbox - Screen switchbox function
        case
        #_case - case statement
          #_init - init function
          when scr_funct = "init" call Acust_zm()
          #_read - disk read function
          when scr_funct = "read" call Rcust_zm()
          #_key - build unique key function
          when scr_funct = "build key" call Kcust_zm()
          #_close - close function
          when scr_funct = "close" call Zcust_zm()
          #_dsp_arr - display array function
          when scr_funct = "display array" call Dcust_zm()
          #_construct - construct function
          when scr_funct = "construct" call Qcust_zm()
          #_after_query - ’after construct’ function
          when scr_funct = "after_query" call AQcust_zm()
          #_get_filter - Get the persistent filter
          when scr_funct = "get sticky" call GFcust_zm()
          #_set_filter - Set the persistent filter
          when scr_funct = "set sticky" call SFcust_zm()
          #_otherwise - otherwise clause
          otherwise let no_function = true
        end case

        #_flow_close - check no_function status
        case
          #_no_function - no function found
          when no_function
            let scratch = "no function"
          #_reset - function was found, reset scratch
          when scratch = "no function"
            let scratch = null
          #_flow_close_otherwise - otherwise clause
        end case

    end function
    # cust_zm()
```

Notice that just like `lib_screen(),` control is directed to the appropriate function in local code based on the value of the variable `scr_funct` (for example, if `scr_funct` is set to "read" then the local code function `Rcust_zm()` is called).

The chief difference between `lib_screen()` and these tailored functions is that `lib_screen()` is a library function and the tailored functions are generated in local code.

In summary, the switchboxes act as a conduit between libraries and functions in the lowlevel code. Data is passed back and forth between libraries and mid and low-level functions. The function `switchbox()` acts very much like a generic library function, but because specific `scr_id`'s and their respective functions must be hardcoded into it, it is generated in local code as part of `main.4gl`.

The Vararg Family of Functions

Sometimes a variable number of arguments have to be passed to the same function. Since functions normally can be passed only a predefined number of arguments, a set of functions have been created that allow you to deal with situations where you don't know the exact number of data elements being passed between functions. This family is called the *vararg* family of functions.

The vararg family of functions includes the following:

```
put_vararg()
get_vararg()
num_vararg()
max_vararg()
getx_vararg()
peekx_vararg()
```

The vararg functions are needed in a variety of situations. A good example of this is a request for `switchbox()` to build a key. A library function requests `switchbox()` to record the key(s) of the current table (by passing `switchbox()` the `scr_funct` "build key"). The library function is data independent and knows nothing about a table and what its key(s) is. In many instances, a table uniquely defines a row by more than one column, thus concatenated keys are used. If a key(s) is to be passed to a function as an argument, the capability of passing a variable number of arguments must be allowed. For instance, you can call a function and pass one key, two keys, three keys, etc. The vararg family of library functions takes care of this for you.

`put_vararg(`*`argument`*`)`

This function initiates the use of `vararg` and must be used immediately before any of the other functions. Its purpose is to temporarily store a string of characters as a single argument.   This function is normally called several times in a row, each time passing a new argument. These arguments are stored in an array from which they can be again retrieved by the `get_vararg()` function. Once a `get_vararg()` or similar function is called, no new arguments can be added. The next calling of `put_vararg()` starts a new variable list and erases the old.

Here are some examples of calls to `put_vararg()`, what the variable list and the array values would look like, and the number of arguments passed (key names and their values are being passed):

```
calls:      call put_vararg("customer_num")
            call put_vararg(p_orders.customer_num)

variable list: list = "customer_num", 104

number of arguments:  2

calls:      call put_vararg("stock_num")
            call put_vararg(p_items.stock_num)
            call put_vararg("manu_code")
            call put_vararg(p_items.manu_code)

variable list: list = "stock_num", 6, "manu_code", SMT

number of arguments:  4

calls:      call put_vararg("order_num")
            call put_vararg(p_items.order_num)
            call put_vararg("stock_num")
            call put_vararg(p_items.stock_num)
            call put_vararg("manu_code")
            call put_vararg(p_items.manu_code)

variable list: list = "order_num", 1005, "stock_num", 1, "manu_code",
HRO

number of arguments:  6
```

Notice that there is a single call to `put_vararg()` (passing a single argument each time) for every argument strung together in `arg_list`. The `arg_list` variable is the variable list that gets passed around to the vararg functions.

get_vararg()

This function retrieves the arguments from the `arg_list` in the order that they were stored. Each time it is called, the next variable in the string is returned. When `get_vararg()` or the related `getx_vararg()` is called, the next call of `put_vararg()` starts a new variable list.

Here is an example of the `get_vararg()` and how it retrieves values from the variable list:

```
variable list:  list = "customer_num", 104
calls:          let string = get_vararg()
                let number = get_vararg()
values:         string = "customer_num"
                number = 104
```

num_vararg()

This function returns a count of the total number of arguments in the `arg_list`.

max_vararg()

This function returns the length of the longest string being held in `arg_list`.

Example:

```
let biggest_string = max_vararg()
```

getx_vararg(*argument_number*)

This function is just like `get_vararg()` except that it returns a specific argument from `arg_list`, not just the next one in sequence. `getx_vararg()` receives as its argument the number of the argument in `arg_list()`. Example:

If there were the following calls to `put_vararg`:

```
call put_vararg("order_num")
call put_vararg(p_items.order_num)
call put_vararg("stock_num")
call put_vararg(p_items.stock_num)
call put_vararg("manu_code")
call put_vararg(p_items.manu_code)
```

Like `get_vararg()`, `getx_vararg()` pulls out both the values of variables along with the variable names.

```
let order_num =  getx_vararg(2)   #  returns the value "1005"
let stock_num =  getx_vararg(4)   #  returns the value "1"
let manu_code =  getx_vararg(6)   #  returns the value "HRO"
```

peekx_vararg(*argument_number*)

This function is just like `getx_vararg()` except that it does not remove the variables in the array. `peekx_vararg()` allows you to return an exact value stored by `put_vararg()`, without removing the string from the `arg_list`.

Note

Since `put_vararg()` and `get_vararg()` re-initialize each other, be careful not to use retrieving functions until you are finished storing arguments.

There are some limitations to the use of `put_vararg()` and `get_vararg()`:

- An argument cannot exceed 512 characters in length.
- You can use no more than 100 arguments.
- The total string size of all arguments cannot exceed 2048 characters.

Examples of `put_vararg()` and `get_vararg()`

The vararg family works nicely within a "while" loop. Often the key of a table is needed in *Screen* source code. A library function called `lib_getkey()` performs this task. However library functions are data independent and know nothing about the database. In many instances, a table uniquely defines a row by more than one column, thus concatenated keys are used. The function `lib_getkey()` uses the vararg family to handle getting one key, two keys, three keys, etc.

`lib_getkey()` first calls `switchbox()` passing it the `scr_funct` "build key." Control trickles down to `mlh_key()`, which is local code and calls `put_vararg()` to load `arg_list` with the keys, however many there are. When control passes back up, `lib_getkey ()`uses a while loop to evaluate what is in `scratch`. This time it uses `get_vararg()` and `num_vararg()`, which returns the number of arguments in `arg_list`. Here are the lines of code in `lib_getkey()` that accomplish this:

```
######################################################################
function lib_getkey()
######################################################################
# This function is called to define the key to the screen.
# It’s main purpose is to set the scr_tab and scr_key variables.
#
    define
        tabname char(18),     # main table for this screen
        tabkey char(30),      # the key to the table
        c char(1),            # temporary char variable
        n smallint            # generic number

    # Trap fatal errors
    whenever error call error_handler

    if (menu_item = "find" and scr_funct != "zoom") or
        menu_item = "browse"
    then
```

```
        call mlh_key()
    else
        call switchbox("build key")
    end if

    if scratch = "no function" then return end if
    let tabname = get_vararg()

    let tabkey = ""
    let n = num_vararg() - 1
    whenever error continue
    while n > 0
        let c = get_vararg()  # don’t need the column name
        let status = 0
        let tabkey = tabkey clipped, get_vararg()
        # check for too long of a key
        if status = -4401
        then
            call lib_error("standard","lib_key",1,"")
            let tabkey = ""
            exit while
        end if
        let n = n - 2
    end while
    whenever error call error_handler
    call put_scrlib("scr_tab",tabname)
    call put_scrlib("scr_key",tabkey)
end function
# lib_getkey()
```

12

Customizing Your Base Program With Triggers

This section explains how to customize your applications while maintaining regenerability. All modifications to the base code are stored in separate files. Your custom code automatically merges into the base code at specific points called triggers.

- How to modify your application with triggers
- Explanation of the triggers
- Custom .4gl/.org files

Using Triggers to Modify Your Application

There are two major concepts that you need to understand to fully utilize Fitrix *Screen*: triggers and blocks. The basic concept going on here is that all of your custom modifications are placed in separate files from the base code generated by the Code Generator. Then when you compile your program, your modifications automatically get placed into the base code.

By keeping your modifications separate from the base code, your programs can be completely regenerable, meaning that with each new upgrade of Fitrix *Screen*, all you need to do is to regenerate your application to take advantage of the newest features built in to Fitrix *Screen*. Also, triggers actually simplify the modification process because you don't even have to know where to place your modification in the source code.

Triggers: Triggers are specific locations, "trigger points," in the generated code where your modifications get inserted. Specifying the name of a trigger before a piece of custom code places your code into the generated code at the point where the trigger occurs.

Blocks: Blocks on the other hand are a bit more complicated. Block commands allow you to replace any piece of generated code or insert any code anywhere in any 4gl file. Blocks are the subject of the next chapter.

Featurizer: The Featurizer is the program that merges triggers and blocks into the source code. The Featurizer is discussed in the next chapter along with blocks.

Trigger File (.trg): Like the .per file, the trigger file also serves as a source of input to the Code Generator. Each trigger file contains modifications to a particular screen. One trigger file may exist for every .per file. While the .per file instructs the Code Generator what kind of code to generate, the .trg file contains additional code that you want to add to the base code.

Some of the main benefits of a trigger file:

- Simplifies the structure of Fitrix *Screen* code, making it easier to use and modify—instead of looking throughout source code to find modifications, specific modifications are kept in a single file.

- Optimizes placement of modifications—instead of having to learn the structure well enough to ideally place your modifications, triggers are automatically inserted into the 4GL code by the Featurizer.
- Reduces development time in creating complex input screens—a trigger file contains quick and easy rules for fast modification.
- Separates your modifications from the code generated by the Fitrix *Screen*—you can keep a custom trigger file with your own modifications so you can distinguish your code from the Code Generator’s.
- Allows regenerability—modifications can all be kept in a single file and code can be repeatedly generated based on the specifications in this file.
- Provides backwards compatibility—modifications made through triggers are assured proper placement in the .4gl code with future releases of the Code Generator.

Trigger files *only* work with source code. Modifications specified in a trigger file are placed in the appropriate place in the code by the Featurizer. Modifications specified in the trigger file are not guaranteed to compile. The Featurizer does not check the syntax of your custom modifications specified in a trigger file. This is the job of the compiler.

Thus when encountering errors during compiling, you should look first to resolve the errors in the trigger file and not in the source code itself.

Although much of the work required in customizing code can be done through a trigger file, there are circumstances when you need to make modifications outside of known trigger points. You can use a variety of special "block commands" to selectively modify virtually any piece of code in any .4gl file. For more information on block commands, refer to "Using Block Commands to Manipulate Code" on page 13-17.

The following items could be put into a trigger file to get added to the generated code:

- Global or static variables.
- Any libraries that are used with a particular program, especially custom libraries.
- Before initialization or after initialization logic.
- Additional I/O logic.
- Before field or after field logic.

- Before input or after input logic.
- Event handling logic.
- Custom functions can replace Fitrix *Screen* generated functions.

The Trigger File

A trigger file contains all of the triggers that modify one particular screen. Trigger files have the same name as the screen that they are modifying with a .trg extension instead of .per. For example, if the name of the .per is `order.per`, then the name of the trigger file modifying it is `order.trg`. Trigger files can accompany any .per file, including zoom and browse .per files.

A trigger file can contain up to three separate sections depending on the type of screen being modified: defaults, input 1 and input 2. These sections determine what .4gl source code files are effected by a particular trigger. For example, a `static_define` trigger can be placed in any one of the three sections with the following effects:

defaults: inserts code into the main.4gl file

input 1: inserts code into the header.4gl file

input 2: inserts code into the detail.4gl file

In each section of the trigger file, you list the name of the trigger and the custom code you want to insert into the source code. The following is the format for a trigger:

```
trigger_name
    custom code
    more custom code
    ;
```

Each trigger must be separated by a semicolon (;).

Note

If you need to use a semicolon as a formatting type characteristic in a trigger or a block command, you must put a backslash before it (`\;`). The backslash is removed during processing.

Comments ("#") placed before a trigger and its associated text causes that trigger and text to be ignored.

Do *not* place comments after the ending semicolon.

The following describes the parts of a trigger file.

`defaults`:

The `defaults` section manages custom entries in the `main.4gl` and `globals.4gl` files. Also, custom characteristics about the program are handled in this section (what libraries to use, what functions not to generate). Extra initialization and disk access logic can also be specified in the `defaults` section.

`input 1`:

The `input 1` section handles modifications in `header.4gl`. This section contains before/after field logic, before/after input logic, and event handling logic for the header portion of the screen.

This section is also used to place variables in `q_` parallel header records. Static variable definitions can also be created in `header.4gl`.

`input 2`:

The `input 2` section places modifications in `detail.4gl`. This section contains the same types of logic as an `input 1` section, except the detail portion of the screen arrays are often involved. Thus custom before/after row, before/after insert, and before/after delete logic is specified in the `input 2` section.

The `input 2` section also allows you to place variables in `q_` parallel detail records. Static variable definitions can also be created in `detail.4gl`.

The section of the trigger file you place your triggers into determines which .4gl file the code is merged into. The type of .per form also determines which section of the trigger file to place your triggers into.

The `input 1` section is used to place code into the `header.4gl` file of a header or header/detail form.

The `input 2` section is only used to place code into the `detail.4gl` file of a header/detail type form.

The `defaults` section is used with all screen types with a variety of effects.

Triggers placed in `input 2` on anything besides a header/detail form are ignored.

The following table shows what .4gl files are effected by placing triggers in certain areas of the trigger file for a particular form type.

Form Type	.trg Section	.4gl File Affected
header header/detail	defaults	main.4gl or globals.4gl
header header/detail	input 1	header.4gl
header/detail	input 2	detail.4gl
add-on zoom extension	defaults input1	scr_id.4gl
browse	defaults	browse.4gl

The following sample trigger (.trg) file gives you an idea of what these look like:

```
defaults
    define
        m_gcntrc record like stgcntrc.*,  # record holding GL control info
        m_xcntrc record like stxcntrc.*,  # record holding control info
        m_xperdr record like stxperdr.*,  # record holding period info
        l_prepared smallint,
        n   smallint,                     # temporary counter
        prev_fld char(40),                # global version of prv_fld
        str array[10] of char(80)         # language independent array;

    libraries
        ../../all.4gm/lib.a;

    do_not_generate
        lld_read;

    after_init
        call gcontrol() returning m_gcntrc.*
        call xcontrol() returning m_xcntrc.*
        call str_init();

input 1

    before_input
        let prev_data = null
        let prev_fld  = null ;

    before_field acct_no
        # set global prev_fld to prv_fld for after field skip logic
        let prev_fld = prv_fld ;

    before_field incr_with_crdt
        # set global prev_fld to prv_fld for after field skip logic
        let prev_fld = prv_fld ;

    after_change_in acct_no
        #_ck_dup - check for duplicate key if field is not null
        if this_data is not null
        then
            #llh_dupchk returns false if duplicate exists(err condition)
            if not llh_dupchk()
            then
                call scr_error("dupchk", "acct_no")
                let p_xchrtr.acct_no = prev_data
                let this_data = prev_data
                let data_changed = false
                let nxt_fld = "acct_no"
                return
            end if
        end if
```

For a more detailed example refer to "Sample Triggers File" on page 12-48.

Trigger File Limitations

A trigger file cannot exceed 699 lines.

If you create a trigger file that is bigger than 699 lines, you can remove large pieces of code, like custom functions, from the trigger file and move them into a separate .4gl/.org file. We recommend you name this file `custom.org` to ensure that your name does not conflict with file names in future upgrades of our applications. For more information on `custom.org` files refer to "Custom .4gl/.org Files" on page 12-47.

Using Triggers in .ext Files

Triggers can be used in .ext (extension) files. These .ext files are similar to .trg files but they contain block commands and triggers necessary to drive a particular feature. This allows you to selectively plug and unplug certain features for different versions of your program. For more information on using triggers in .ext files refer to "Pluggable Features (.ext Files)" on page 13-32.

The Triggers

Following is a discussion of each trigger. A complete sample trigger file can be found in Appendix D, "Sample Trigger File." Triggers associated with Add-On Headers are discussed under "Add-On Header Triggers."

define

This trigger lets you add global variables to the define statement in `globals.4gl`.

Trigger File Placement

- defaults
- input 1
- input 2

Defaults Section

A define trigger specified in the defaults section puts variables directly into the beginning of the globals (`globals.4gl`) file. A define trigger in the defaults section is used for individual variables not associated with any records. *A comma must be included after the last variable*.

Example

```
defaults
   define
          myvar4  smallint,
          myvar5  smallint,
          myvar6  smallint,
          ;
```

Input 1 and Input 2 Sections

A `q_` record(s) (record parallel to the header or detail record) is always created when generating with the Code Generator.

The `define` trigger specified in the `input 1` section or `input 2` section places variables into the `q_` record parallel to the header or the `q_` record parallel to the detail, respectively. *Leave comma out on last variable*.

Example

```
input 1
   define
          myvar7  smallint,
          myvar8  smallint,
          myvar9  smallint
          ;
```

Notes

The `define` trigger isn't allowed in .ext files. Use `function_define`, `static_define`, or `in_block` commands for these files.

When using multiple directory search paths, the Featurizer either *replaces* or *appends* `define` trigger definitions processed previously. The action that the Featurizer takes depends on the current setting in the `fglpp.opt` file. For more information, refer to "Maintaining Backwards Compatibility—The Options Files" on page 2-19.

static_define

The `static_define` trigger inserts static (local) variables into the define statement for particular .4gl files. The `static_define` trigger is the same as the `define` trigger except that variables are not placed in `globals.4gl`. Static variables are variables that are available only within a particular 4gl file.

Leave comma out on last variable.

Trigger File Placement

- defaults
- input 1
- input 2

Example

```
static_define
       myvar10  smallint,
       myvar11  smallint,
       myvar12  smallint
       ;
```

Notes

The `static_define` trigger, placed in different sections of the .trg file, yields different results. In the following table, a `static_define` placed in a section specified in the first column results in a placement of variables at the top of a .4gl file specified in the second column.

Form Type	.trg Section	.4gl File Affected
header(/detail)	defaults	main.4gl
header(/detail)	input 1	header.4gl
header/detail)	input 2	detail.4gl
add-on zoom extension	defaults	scr_id.4gl
browse	defaults	browse.4gl

For zoom and add-on screens, it makes the variable static to the `{screen_id}.4gl` file, and for browse screens, it is static to the `browse.4gl` file.

When using multiple directory search paths, the Featurizer either *replaces* or *appends* `static_define` trigger definitions processed previously. The action that the Featurizer takes depends on the current setting in the `fglpp.opt` file. Refer to "The Featurizer Options File (fglpp.opt)" on page 2-19 for more information.

function_define

This trigger defines the specified variables as local to the specified function.

Trigger File Placement

- defaults
- input 1
- input 2

Example

```
function_define my_function
       myvar23  smallint,
       myvar24  smallint
       ;
```

Notes

The file that this trigger tries to find the function in depends on where the trigger is located within the .trg or .ext file.

Form Type	.trg Section	.4gl File Affected
header header/detail	defaults	main.4gl
header header/detail	input 1	header.4gl
header/detail	input 2	detail.4gl
add-on zoom extension	defaults	scr_id.4gl
browse	defaults	browse.4gl

If the `function_define` trigger is in a "start file" section, the function must reside in that specified file.

on_event

The `on_event` trigger allows you to automatically add event handling logic.

Note

You must re-run the Code Generator after adding an `on_event` trigger. This is the only trigger that does not get merged into your code with the Featurizer.

Trigger File Placement

- defaults
- input 1
- input 2

Example

```
on_event noworry
        let scratch = "Don’t worry, be happy"
        call lib_message("scr_bottom")
        sleep 3
        ;
```

Notes

Additions to code required to handle global and local events can all be added automatically by using the `on_event` trigger. However, since an event must be invoked by a keystroke or by selecting it from the Navigation Menu, additional steps must be taken in order to add the event to the navigation and hot key tables. The event can be added to the navigation tables through the Navigation and Hot Key features available with the User Control Library.

The location of the `on_event` trigger in the trigger file determines where the event is placed in the code and where it can be used.

For a global event, specifying `on_event` in the `defaults` section results in `global_events()` being added to `main.4gl`. This function handles the logic for all global events in the program.

Specifying `on_event` in the `input 1` section causes the event to be executed only from input area 1 (header section).

Specifying `on_event` in the `input 2` section causes the event to be executed only from input area 2 (detail section).

The following table summarizes the results of the `on_event` trigger:

section	.4gl file:input function	.4gl:event handling function
defaults	none	main.4gl:global_events
input 1	header.4gl:llh_input	header.4gl:llh_event
input 2	detail.4gl:lld_input	detail.4gl:lld_event

For more information, refer to "Event Handling Logic" on page 15-2.

See also "The Navigate Feature" and "Hot Keys" in the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

More Examples

The following examples show where the `on_event` trigger inserts the event into the code.

Specifying a global event `noworry` in the `defaults` section of the trigger file:

```
on_event noworry
        let scratch = "Don’t worry, be happy"
        call lib_message("scr_bottom")
        sleep 3
        ;
```

results in `main.4gl`:

```
######################################################################
function global_events(act_key, p_funct)
# returning true if it runs the event, otherwise false
######################################################################
# This function’s job is to run all events that need to be run
# on a global (program wide) basis.  If you have defined an event
# that needs to be run at the menu level in addition to the local
# input level, the event must be listed here.
# If you wish to know the function name that called hot_key, it
# is passed as p_funct.
#
    define
        act_key char(15),       # Action to process
        p_funct char(15)        # Current function name

    # Process the events based on act_key
    case
      when act_key = "noworry"
        let scratch = "Don’t worry, be happy"
        call lib_message("scr_bottom")
        sleep 3

      otherwise return false
    end case

    return true
end function
# global_events()
```

This global event would not be accessible in a program until you add the global event via the Navigation feature.

Local events are specified in a similar manner. For local events, specifying `on_event` in the `input 1` or `input 2` sections will yield two entries. In the input statement, local event processing, a call to `hot_local` is placed in the code calling the event specified. Also, the event handling code specified is placed in the `llh_event()/lld_event()` function as an additional `when` clause to the `case` statement. Example:

In the trigger file, specifying the following in `input 1` section:

```
on_event zoom
      and infield(order_date)
        if zoom("date_zm")
        then
            let p_orders.order_date = scratch
            let nxt_fld = "order_date"
        end if
  ;
```

results in `header.4gl`:`llh_input`:

```
# Local event processing
label event:
  call hot_local("date_zm")
  . . .
  call llh_event()
```

and in `header.4gl:llh_event`:

```
case
      when . . .

      when scr_funct = "zoom"
        and infield(order_date)
          if zoom("date_zm","")
          then
              let p_orders.order_date = scratch
              let nxt_fld = "order_date"
          end if

      when . . .
end case
```

libraries

If you have built a library of custom functions and wish to use this custom library with the current application, you can specify the `libraries` trigger in a trigger file. Any libraries specified here are automatically placed into the `LIBFILES` section of the local `Makefile`.

Trigger File Placement

- defaults section

Example

```
libraries
        $(fg)/lib/mylib.a
        ;
```

Notes

Any library specified through a `libraries` trigger is placed after the `../lib.a` line but before the rest of the Fitrix *Screen* libraries. If there are any functions that have the same name across libraries, the functions found in the earlier `LIBFILES` entry is executed.

custom_libraries

If you have built a library of custom functions and wish to use them before the `../lib.a` library with the current application, you can do so by specifying the `custom_libraries` trigger in a trigger file. Any libraries specified here are automatically placed into the `LIBFILES` section of the local Makefile before `../lib.a`.

Trigger File Placement

- defaults section

Example

```
custom_libraries
       $(fg)/lib/newlib.a
       ;
```

switchbox_items

The switchbox_items trigger adds an additional when clause to the case statement in the function switchbox(). Use this trigger to add additional screens to the flow controller.

Trigger File Placement

- defaults section

Example

```
switchbox_items
       screen2 screen2
       ;
```

Notes

The switchbox() function resides in main.4gl.

You can also specify function arguments in your switchbox_items trigger code:

```
switchbox_items
       fredA fredA(arg1, arg2, arg3)
       ;
```

When using multiple directory search paths, the Featurizer either *replaces* or *appends* switchbox_items trigger definitions previously processed. The action that the Featurizer takes depends on the current setting in the fglpp.opt file. Refer to "The Featurizer Options File (fglpp.opt)" on page 2-19.

before_init

The `before_init` trigger let’s you insert logic before the program initialization occurs in the `init()` function.

Trigger File Placement

- defaults section

Example

```
before_init
       initialize myvar1 to null
       initialize myvar2 to null
       initialize myvar3 to null
       ;
```

Notes

The `main.4gl` file performs initialization. It does this by calling the function `init()`. Custom initialization logic can be placed before or after this call to `init()` by using the `before_init` or `after_init` trigger, respectively.

after_init

The after_init trigger is similar to before_init, but the after_init trigger places code after the open form statement in main.4gl (after the call to init()).

Trigger File Placement

- defaults section

Example

```
after_init
    open window w_cust_zm at 4,5 with form "cust_zm"
      attribute (white, border)
    ;
```

at_eof

The `at_eof` trigger is useful for placing custom functions at the end of a .4gl file. This trigger is often used in conjunction with the `do_not_generate` trigger.

Note

Block commands allow you to duplicate the function of the `do_not_generate` and `at_eof` triggers but in a much cleaner way. Using block commands is preferred. Refer to "Block Command Statements" on page 13-24.

Trigger File Placement

- defaults section
- input 1
- input 2

Example

```
defaults
at_eof           #placed at end of main.4gl
#######################################################
function please_wait()
#######################################################
# Trap fatal errors
     whenever error call error_handler

     instead of this
     message " Please wait..."

     let’s say this
     message " Have a happy day ..."

end function
# please_wait()
;
```

Notes

The placement of the at_eof trigger in the trigger file yields different results.

Form Type	.trg Section	.4gl File Affected
header header/detail	defaults	main.4gl
header header/detail	input 1	header.4gl
header/detail	input 2	detail.4gl
add-on zoom extension	defaults	scr_id.4gl
browse	defaults	browse.4gl

When using multiple directory search paths, the Featurizer either *replaces* or *appends* at_eof trigger definitions processed previously. The action that the Featurizer takes depends on the current setting in the fglpp.opt file. Refer to "The Featurizer Options File (fglpp.opt)" on page 2-19.

In the example above, the library function please_wait was copied into local code, modified, and placed after the at_eof trigger defaults section.

There are three ways that functions can be customized using at_eof:

1. A custom function can be coded from scratch and given a unique name. This function may also be called from the trigger file.
2. A library function can be copied into your local code and customized using the at_eof trigger. This function retains the same name as the library function and the call (wherever it is) is left the same. But when the function is called, the local function is used instead of the library function because during compilation the linker looks first to local code and then to the libraries when resolving function calls.

During regeneration, *Screen* placed this code at the end of `main.4gl`. When compiling, the linker looked locally first and found this function and linked it in. When running the program, this custom function was called instead of the library function.

3. A local function normally generated with *Screen* can be modified and used in place of the generated function. However a `do_not_generate` trigger must also be used to prevent the original function from being generated or else two local functions will exist in the local code resulting in a compile or run-time error. The `do_not_generate` trigger is discussed next.

do_not_generate

The `do_not_generate` trigger can prevent any function in local code from being generated. This is ideal for when you take a function generated with Fitrix *Screen* and modify it so that when the call is made, the modified function is used in place of the original function.

Note

Block commands allow you to duplicate the function of the `do_not_generate` and `at_eof` triggers but in a much cleaner way. Using block commands make this trigger unnecessary and is preferred. Refer to "Block Command Statements" on page 13-24.

Trigger File Placement

- defaults section
- input 1
- input 2

Example

```
defaults
  ...
  ...
    do_not_generate
            mlh_clear
            ;
```

Notes

The modified function must *replace* the function generated by Fitrix *Screen*. You can use the `at_eof` trigger to specify the modified function and `do_not_generate` to prevent the original function from generating.

As for calls to functions, `do_not_generate` has no effect on *calls* to functions that are specified under `do_not_generate`. Fitrix *Screen* generates calls to functions whether the function is there or not.

The do_not_generate trigger acts as a "delete block" block command. The trigger can be in any section of the .trg or .ext file. If there is a file context (it's not in a "defaults" section), then the context is determined by the current file context (within a "start file" or "input n" section).

If it is in a defaults section of a .trg file (there is no such section in an .ext file), then the following rules are applied to determine which .4gl file to assign the command to.

If the screen type is zoom or add-on, it uses the .4gl file associated with this .trg file.

If the screen type is browse, it uses browse.4gl.

If the screen type is header or header/detail, then it uses the following logic to determine the .4gl file:

If the first 2 characters of the function are "ml", then it uses midlevel.4gl.

If the first 3 characters of the function are "llh", then it uses header.4gl.

If the first 3 characters of the function are "lld", then it uses detail.4gl.

Otherwise, it uses main.4gl.

Form Type	.trg Section	Function Not Generated	.4gl File Affected
header header/detail	defaults	ml* llh* lld* other	midlevel.4gl header.4gl detail.4gl main.4gl
header header/detail	input 1		header.4gl
header/detail	input 2		detail.4gl
add-on zoom extension	defaults		scr_id.4gl
browse	defaults		browse.4gl

More Examples

This example explains how a local function can be modified using the `at_eof` and `do_not_generate` triggers.

You can take this local function generated with Fitrix *Screen*:

```
######################################################################
function mlh_clear()
######################################################################
#
    initialize p_stomer.* to null
    initialize q_stomer.* to null
    initialize m_stomer.* to null
end function
# mlh_clear
```

and modify it and place it within the `at_eof` trigger of the trigger file:

```
    defaults
      ...
      ...
```

```
        at_eof

######################################################################
function mlh_clear()
######################################################################
#
    initialize p_stomer.* to null
    initialize q_stomer.* to null
    initialize m_stomer.* to null
a-> initialize myvar1 to null
a-> initialize myvar2 to null
a-> initialize myvar3 to null
end function
# mlh_clear
```

If you were to regenerate at this point, the modified function would be placed at the end of `main.4gl`. However, Fitrix *Screen* would also regenerate the original `mlh_clear()`, yielding two functions of the same name in the local program directory. To avoid this from happening, you would add the `do_not_generate` trigger to tell the Code Generator to *not* generate the original `mlh_clear()`:

```
defaults
  ...
  ...
    do_not_generate
            mlh_clear
            ;
```

Thus the original `mlh_clear()` is not generated and the *one* modified `mlh_clear()` is placed into the source code. When running the program, the call to `mlh_clear()` uses the modified `mlh_clear()`.

on_screen_record_prep

Code specified in this trigger is placed at the bottom of the p_prep() function.

Trigger File Placement

- input 1
- input 2

Example

```
on_screen_record_prep
       display "p_prep successful"
       sleep 3
       ;
```

Notes

The p_ records are filled automatically from data found in m_ records and from lookups. If additional code is needed to fill p_ records or code is needed to load q_ records, this is the trigger to do it.

This trigger can be used in either input 1 or input 2 to affect llh or lld respectively.

on_disk_record_prep

Code specified in this trigger is placed at the bottom of the `m_prep()` function.

Trigger File Placement

- input 1
- input 2

Example

```
on_disk_record_prep
       display "p_prep successful"
       sleep 3
       ;
```

Notes

m_ records are filled automatically from data found in `p_` records (and sometimes `q_` records) before a disk write. If additional code is needed for writing to table columns not specified in the `p_` or `q_` records, this is the trigger to do it.

This trigger can be used in either `input 1` or `input 2` to affect llh or lld respectively.

on_disk_read

This trigger is placed in `llh_read()/lld_read ()`after the disk read is performed and before the call to `llh_p_prep()`/`lld_p_prep()`. This trigger is executed only if the read is successful.

Trigger File Placement

- input 1
- input 2

Example

```
on_disk_read
       display "read successful"
       sleep 3
       ;
```

Notes

This trigger can be used in either `input 1` or `input 2` to affect llh or lld respectively.

on_disk_delete

This trigger is added after a disk record has been successfully deleted. This is usually at the end of the delete function, `llh_delete()/lld_delete()`. Data is deleted based on rowid.

Trigger File Placement

- input 1
- input 2

Example

```
on_disk_delete
       display "delete successful"
       sleep 3
       ;
```

Notes

This trigger can be used in either `input 1` or `input 2` to affect llh or lld respectively.

on_disk_add

In `input 1`, this trigger is inserted after the row is successfully added to the table. This is placed in `llh_add`.

In `input 2`, because the insert cursor is used for speed, the `on_disk_add` is executed after each row of detail is put into the cursor. The cursor does not get written to the table until it is closed.

Trigger File Placement

- input 1
- input 2

Example

```
on_disk_add
       display "add successful"
       sleep 3
       ;
```

Notes

This trigger is used in either `input 1` or `input 2` to affect llh or lld respectively.

on_disk_update

This trigger is inserted after the update statement successfully occurs in `llh_update()`.

Trigger File Placement

- input 1 only

Example

```
on_disk_update
       display "update successful"
       sleep 3
       ;
```

before_input

This trigger inserts your code directly before the input statement in `llh_input()/lld_input()`. Data checks before an input can be done here.

Trigger File Placement

- input 1
- input 2

Example

```
      before_input
             display "start inputting"
             sleep 3
             ;
```

Notes

This trigger can be used in either `input 1` or `input 2` to affect llh or lld respectively.

before_field

This trigger inserts logic that occurs just before a field is entered.

Trigger File Placement

- input 1
- input 2

Example

```
before_field ship_instruct
       if menu_item = "add    "
       then
           let p_orders.ship_instruct = "will call"
           let nxt_fld = "ship_charge"
       end if
       ;
```

Notes

This field-specific before field logic is placed in `llh_b_field()/lld_b_field()` as an additional `when` clause of a `case` statement. This `case` statement handles any specific before field work before `lib_before` is called. Library function `lib_before()` handles the generic before field logic.

after_field

This trigger inserts logic that occurs just after leaving a field.

Trigger File Placement

- input 1
- input 2

Example

```
after_field ship_charge
        if menu_item = "add    "
        then
            let p_orders.ship_charge = "10.00"
        end if
        ;
```

Notes

This field-specific after field logic is placed in `llh_a_field()/lld_a_field()` as an additional `when` clause of a `case` statement. This `case` statement handles any specific after field work before `lib_after()` is called. The library function `lib_after()` handles the generic after field logic.

after_change_in

This trigger allows you to place logic to be executed when data changes in a particular field.

Trigger File Placement

- input 1

Example

```
after_change_in order_date
        if p_orders.order_date < "01/01/80"
        then
            error "Invalid date"
            let nxt_fld = "order_date"
        end if
        ;
```

Notes

This trigger is an ideal place to put field-level data validation logic. Logic is put in the `if data_changed` statement in `llh_a_field()`.

If you create validation on a field that restores the original value if the data is changed, you need to manually set the `data_changed` variable to false.

Here is an example: Say you have a field that the user can enter into but you don’t want the user to be able to change the value so you use a let next_field=current field statement and then you restore the original value.

When the user enters a field for the first time the data changed variable is set to false. If the user changes a value, the `data_changed` variable is set to true, which initiates any validation logic. This puts the user in a loop until a valid value is specified. However, if your validation logic restores the data to its original value, the `data_changed` variable is still set to true, even though the data hasn’t changed. The user would never be able to get out of the field. The solution is to manually set the `data_changed` variable to false.

after_input

This trigger is placed in `llh_a_input()` or `lld_a_input()` which is called just before the `exit input` statement in `llh_input()/lld_input()`. This is an ideal place to put record-level data validation.

Trigger File Placement

- input 1
- input 2

Example

```
after_input
       if p_orders.ship_charge < 5.00
       then
           error "ship charge too low"
           let nxt_fld = "ship_charge"
       end if
```

Notes

The `nxt_fld` variable can be set to re-enter the input statement.

before_insert

The `before_insert` trigger places code at the bottom of the `lld_b_insert()` function. This code is executed before a row is inserted.

Trigger File Placement

- input 2 only

Example

```
before_insert
       display "done before insert"
       sleep 3
       ;
```

Notes

The `lld_b_insert()` function is called within an input array statement.

after_insert

The `after_insert` trigger places code at the bottom of the `lld_a_insert()` function. This code is executed after a row is inserted.

Trigger File Placement

- input 2 only

Example

```
after_insert
       display "done after insert"
       sleep 3
       ;
```

Notes

The `lld_a_insert()` function is called within an input array statement.

before_row

The `before_row` trigger places code in the `lld_b_row()` function. This code is executed before a row is entered.

Trigger File Placement

- input 2 only

Example

```
before_row
       display "done before row"
       sleep 3
       ;
```

Notes

The `lld_b_row()` function is called within an input array statement.

after_row

The `after_row` trigger places code in the `lld_a_row()` function. This code is executed when leaving a row.

Trigger File Placement

- input 2 only

Example

```
after_row
       display "done after row"
       sleep 3
       ;
```

Notes

The `lld_a_row()` function is called within an input array statement.

before_delete

The `before_delete` trigger places code in the `lld_b_delete()` function. This code is executed before a row is deleted.

Trigger File Placement

- input 2 only

Example

```
before_delete
       display "done before delete"
       sleep 3
       ;
```

Notes

This code is performed when the user presses [F2] to delete the row and before the actual array elements have shifted. `lld_b_delete()` is called within an input array statement.

after_delete

The after_delete trigger places code in the lld_a_delete() function. This code is executed after a row is deleted.

Trigger File Placement

- input 2 only

Example

```
after_delete
       display "done after delete"
       sleep 3
       ;
```

Notes

This code is performed when the user presses [F2] to delete the row and after the actual array elements have shifted. lld_a_delete() is called within an input array statement.

Custom .4gl/.org Files

The custom.org file is the place where you can put any custom functions that are called from your trigger file. This file must have a name of eight characters or less and a .org extension. The Fitrix *Screen* Code Generator uses the information in your trigger files along with information in your .per files to create the 4GL code. Anything within a custom.org file is not touched by the Code Generator. Since syntax checking is performed on the custom.org file, it is a good place to start when debugging your applications.

Note

If you have any custom.org files and you are using version control, then you need to be sure to add a start file "custom.org" line to your .ext file in any directory that does not contain the .org file. This ensures that the Featurizer sees your .org file and compiles it properly in your local directory. For more information on version control and .org files refer to "Using Non-Generated .4gl files With Version Control (fg_funcs.4gl)" on page 16-19.

The "whenever error call error_handler" line of code should be placed in the first function of every .4gl file. Sections of code should be wrapped with the "whenever error continue" ... "whenever error call error_handler" only when specific errors are expected. If this wrap is used, then the expected errors should be tested for immediately following.

If your custom.org file grows too large, you may need to create more than one .org file. Note that the name before the .org extension must be a maximum of 8 characters.

Sample Triggers File

```
defaults
       define
              myvar1  smallint,    # placed in globals
              myvar2  smallint,    # include comma on last variable
              myvar3  smallint,
              ;
       static_define
              myvar10  smallint,   # placed at top of main.4gl
              myvar11  smallint,   # leave comma out on last variable
              myvar12  smallint
              ;
       on_event noworry             # placed in main.4gl:global_events
              let scratch = "Don’t worry, be happy"
              call lib_message("scr_bottom")
              sleep 3
              ;
       libraries                    # placed in local Makefile:LIBFILES
              $(fg)/lib/mylib.a
            ;
       do_not_generate              # does not generate this function
              mlh_clear
              ;
       switchbox_items
              new_zm new_zm        # placed in switchbox
              ;
       before_init                  # placed before main’s call to init
              initialize myvar1 to null
              initialize myvar2 to null
              initialize myvar3 to null
              ;
       after_init                   # placed after main "open form"
              open window w_cust_zm at 4,5 with form "cust_zm"
              attribute (white, border)
              ;
       at_eof                       # placed at end of main.4gl
#######################################################################
function please_wait()
#######################################################################
   # Trap fatal errors
   whenever error call error_handler
   instead of this
   message " Please wait..."
   let’s say this
   message " Have a happy day ..."
end function
please_wait()
            ;
input 1
       define
```

```
        myvar4  smallint,    # placed in globals.4gl:q_ record
        myvar5  smallint,    # leave comma out on last variable
        myvar6  smallint
        ;
static_define
        myvar13  smallint,   # placed at top of header.4gl
        myvar14  smallint,   # leave comma out on last variable
        myvar15  smallint
        ;
on_event date_zm             # placed in llh_input and llh_event
     and infield(order_date)
        if zoom("date_zm","")
        then
        let p_orders.order_date = scratch
        let nxt_fld = "order_date"
        end if
        ;
on_screen_record_prep         # placed in llh_p_prep
        display "p_prep successful"
        sleep 3
        ;
on_disk_record_prep           # placed in llh_m_prep
        display "p_prep successful"
        sleep 3
        ;
on_disk_read                  # placed in llh_read
        display "read successful"
        sleep 3
        ;
on_disk_delete                # placed in llh_delete
        display "delete successful"
        sleep 3
        ;
on_disk_add                   # placed in llh_add
        display "add successful"
        sleep 3
        ;
on_disk_update                # placed in llh_update
        display "update successful"
        sleep 3
        ;
before_input                  # placed before "input" in
        display "start inputing"    # llh_input
        sleep 3
        ;
before_field ship_instruct    # placed before a field
        if menu_item = "add    "    # llh_b_field
        then
            let p_orders.ship_instruct = "will call      "
            let nxt_fld = "ship_charge"
        end if
        ;
after_field ship_charge        # placed after a field
```

```
              if menu_item = "add    "    # llh_a_field
              then
                  let p_orders.ship_charge = "10.00"
              end if
              ;
       after_change_in order_date         # placed after a field
              if p_orders.order_date < "01/01/80"  # llh_a_field
              then
                  error "Invalid date"
                  let nxt_fld = "order_date"
              end if
              ;
       after_input                        # placed in llh_a_input
              if p_orders.ship_charge < 5.00
              then
                  error "ship charge too low"
                  let nxt_fld = "ship_charge"
              end if
              ;
       at_eof                            # placed at end of header.4gl
 Put lots of lovely text here.
            ;
input 2
       define
              myvar7  smallint,     # placed in globals.4gl:q_[] record
              myvar8  smallint,     # leave comma out on last variable
              myvar9  smallint
              ;
       static_define
              myvar16  smallint,    # placed at top of detail.4gl
              myvar17  smallint,    # leave comma out on last variable
              myvar18  smallint
              ;
       on_event prce_zm                  # placed in lld_input and lld_event
           and infield(unit_price)
              if zoom("prce_zm","")
              then
                  let p_items.unit_price = scratch
                  let nxt_fld = "unit_price"
              end if
              ;
       on_screen_record_prep              # placed in lld_p_prep
            display "p_prep successful"
            sleep 3
            ;
       on_disk_record_prep                # placed in lld_m_prep
              display "p_prep successful"
              sleep 3
              ;
       on_disk_read                       # placed in lld_read
              display "this row read"
              sleep 3
              ;
```

```
    on_disk_delete                # placed in lld_delete
            display "this row deleted"
            sleep 3
            ;
    on_disk_add                   # placed in lld_add
            display "this row added"
            sleep 3
            ;
    on_disk_update                # placed in lld_update
            display "update successful"
            sleep 3
            ;
    before_input                  # placed before "input" in
            display "start inputing"    # lld_input
            sleep 3
            ;
    before_insert                 # placed in lld_b_insert
            display "done before insert"
            sleep 3
            ;
    after_insert                  # placed in lld_a_insert
            display "done after insert"
            sleep 3
            ;
    before_row                    # placed in lld_b_row
            display "done before row"
            sleep 3
            ;
    after_row                     # placed in lld_a_row
            display "done after row"
            sleep 3
            ;
    before_delete                 # placed in lld_b_delete
            display "done before delete"
            sleep 3
            ;
    after_delete                  # placed in lld_a_delete
            display "done after delete"
            sleep 3
            ;
    after_input                   # placed in lld_a_input
            if p_items.unit_price < 1.00
            then
                error "unit price too low"
                let nxt_fld = "unit price"
            end if
            ;
  at_eof                       # placed at end of detail.4gl
Put lots of lovely text here.
         ;
This triggers file cannot be more than 699 lines.
```

13

The Featurizer and Blocks

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

- Merging custom code into generated code with the Featurizer
- Customizing generated code with block commands
- Making your programs regenerable

Featurizer Overview

The Featurizer merges custom modifications into 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 performs these four tasks:

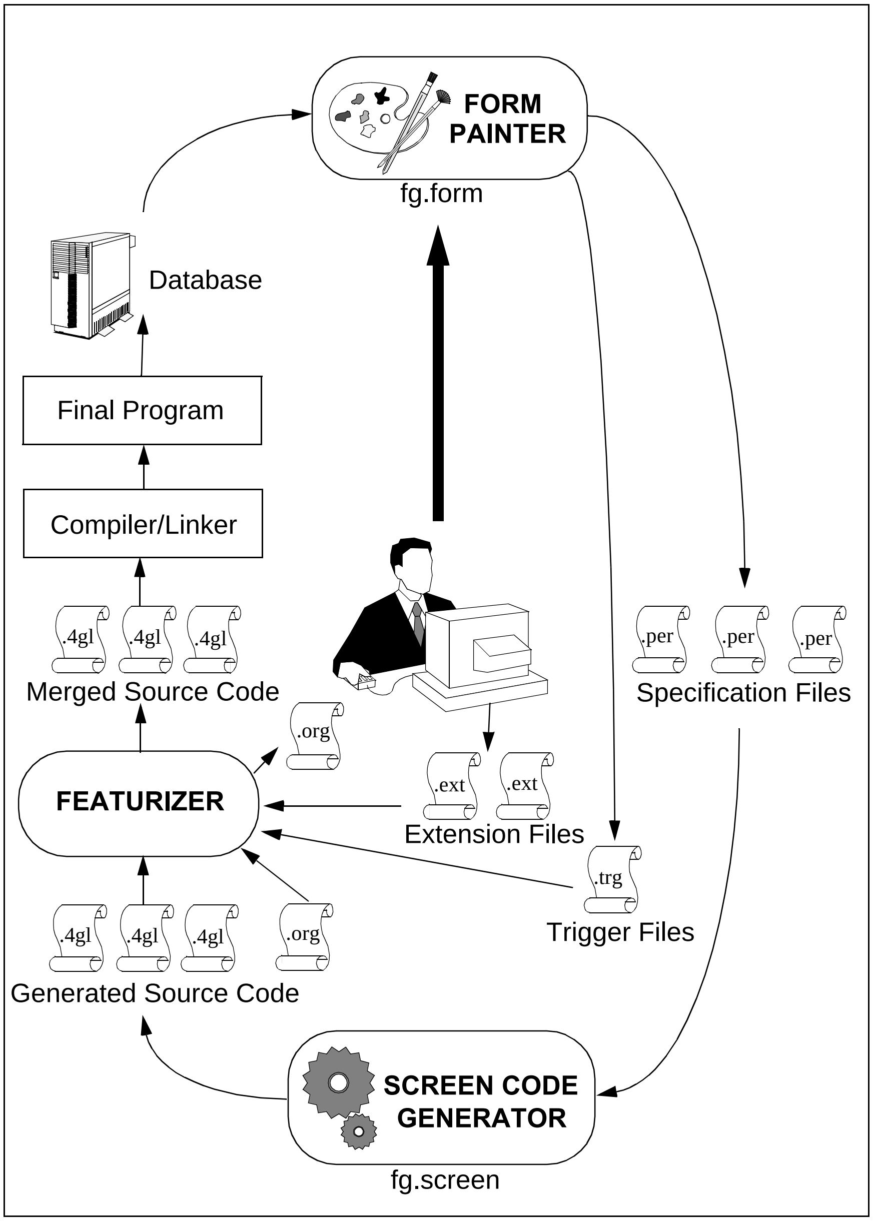
1. Trigger merging
2. Block merging
3. Feature set merging
4. Version control

Before reading this section, you should familiarize yourself first with triggers. See the previous section for information on triggers.

Note

There are some compatibility issues to be aware of between the Trigger Merge Utility and the Featurizer. Please read "Maintaining Backwards Compatibility—The Options Files" on page 2-19.

The following is the Tools Overview diagram modified to include the Featurizer. New files and utilities on the diagram are explained after the diagram.


FORM PAINTER
fg.form
Database
Final Program
Compiler/Linker
.4gl
.4gl
.4gl
Merged Source Code
.per
.per
.per
Specification Files
.org
FEATURIZER
.ext
.ext
Extension Files
.trg
Trigger Files
.4gl
.4gl
.4gl
.org
Generated Source Code
SCREEN CODE GENERATOR
fg.screen

The flow of the preceding diagram is as follows. First you create your input program with the Form Painter. The input program is saved as a .per file. While in the Form Painter you can also define custom modifications to your input program through triggers. Triggers are saved in a .trg file. You can also place source code for custom features in .ext files. .ext files contain triggers and block commands which instruct the Featurizer where and how to modify the generated code. These .ext files must be coded manually with your own text editor.

The next step is to generate code for your program. This is done by invoking the Code Generator on your .per files created with the Form Painter. The Code Generator creates the basic 4GL source code to run your program. This generated code is stored in .4gl and/or .org files. A .org file is generated if the Code Generator finds a .org file with the same name that it is trying to generate anywhere in the directory path.

Next, the Featurizer is run, which searches for the custom code you put into triggers as well as custom code stored in .ext files. The Featurizer merges the code in your .trg and .ext files and places it into the appropriate areas in the source code. This creates merged .4gl files for all of the code needed to run your application. An .org file is also created for each file that has triggers or blocks merged into it. These .org files contain the original code that was generated by the Code Generator and are used by the Featurizer during subsequent merges. Next, you link and compile the .4gls. This creates your completed, ready-to-run program.

Featurizer Terminology

The following is an introduction to key concepts talked about in this chapter.

Regenerability

Regenerability is 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 re-generated. The Featurizer gives you true regenerability.

Triggers

Triggers should be thought of as custom 4GL code that is executed from known places in the application. Triggers are used when you want to customize the original functionality provided by the Code Generator. Examples would be "upon disk update" and "after a screen field has changed."

Triggers allow the generated code to be regenerable. Instead of modifying the physical source code generated by the Code Generator, certain kinds of modifications are placed in .trg files in trigger format. These triggers are then merged into the generated program. This allows the program to be re-generated without losing the custom modifications.

Triggers denote logical locations in the source code.

Source Code Blocks

By 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.

The reason for the definition of source code blocks is that there is not a trigger for every place in the code that may need to be modified. For truly regenerable source code, you may describe changes as alterations to known blocks of source code.

Fitrix *Screen* 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. The more blocks within a source code file the better.

Custom Directories (Version Control)

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. Custom directories and version control are discussed in "Version Control" on page 16-1.

Plug-in 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.

Plug-in 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

Feature sets are simply 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 (`base.set`) include a list of features in the order that they are applied to the source code.

Invoking the Featurizer

The Featurizer can be invoked four ways:

1. From `fg.make.`
2. During code generation.
3. From the Form Painter.
4. 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 triggers and blocks when calling `fg.make`. Refer to "Compiling Generated Code" on page 14-2 for more information on `fg.make`.

Invoking From the Code Generator

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

Invoking From the Form Painter

The Compile option of the Form Painter invokes `fg.make` with the -mf flags. The -mf flags are explained below. Keep in mind that this forces a merge of all related triggers every time the Compile option is selected.

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 set] [-yes]
[-trace] [filename...]
```

-dbname	Specify a database name to use. Overrides `$DBNAME` or "standard" if `$DBNAME` doesn't exist.
-C	Inserts comments into merged .4gl code noting origin of triggers and/or blocks. However, comments are not placed in Makefiles.
-force	Forces merge of all triggers and blocks regardless of file time stamps.
-set	Specify the *.set file to use (to define feature set).
-yes	Automatically overwrite files without write permission set.
-trace	Displays Featurizer activity.
filename	The file(s) to pre-process. If omitted, a list is built of the files that need pre-processing.

The Difference Between Triggers and Blocks

Triggers are tagged locations in the generated source code for inserting requested modifications. Various locations have been identified in the generated code where custom modifications are commonly made. The Code Generator inserts special tags at these locations. The Featurizer takes your logic within a trigger and inserts it into the code at these locations.

When using triggers, you are limited to specific locations in the source code. It is likely that you will have a need to insert custom logic outside of these known locations.

The Code Generator places special markers throughout the generated code which identify "blocks" of source code. Using special block commands, you can perform custom modifications to any part of the code.

The following section explains the three ways in which you could modify a piece of source code for which a trigger does not exist.

The example:

There might be an occasion when you must add custom logic to the `llh_add` function in `header.4gl`.

To assign the current date to an `m_record` variable called "entry_date", include one line of code just before the Informix insert command, as follows:

```
# 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]
```

This simple modification assigns the current date to the m_ record variable `entry_date` Before modifying this program, an `entry_date` column has been added to the orders table. Right after today’s date is assigned to the m_orders.entry_date variable, the insert takes all values in the m_ record and writes them into the orders table, including the value of the entry_date variable.

There is no trigger that allows you to do this. There is the on_disk_add trigger that adds code to the llh_add function, but as you can see from the on_disk_add trigger tag above (#_on_disk_add), any on_disk_add trigger logic is added *after* the insert command is executed. You must insert custom code *before* the insert command. This location is not identified by triggers.

If no trigger location exists for your modification, you have three options, listed in order of least desirable to most desirable:

1. **You can modify the llh_add function directly.** This would require that you manually maintain this .org file whenever further modifications are placed in code. You would have to re-insert this line of code each time the Code Generator was run. This type of modification is *not* regenerable.

2. **You could copy the entire function into the trigger file under the at_eof trigger and then modify it there.** The at_eof trigger is for placing whatever is under it at the bottom of a .4gl source code file. It is commonly used for placing customized functions into source code. Your trigger file would look like this:

```
input 1

     at_eof

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

     let m_orders.entry_date = today

     #_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()
        ;
```

The Featurizer would read the `at_eof` trigger and place the modified function into the source code. Since you are modifying a locally created function and the original `llh_add` function is still in local code, `at_eof` is frequently used with the `do_not_generate` trigger, which removes the original `llh_add` function from source code. This prevents two functions with the same name from existing in local code. If two functions with the same name exist in local code, then the program will not compile. Your `do_not_generate` trigger would appear as follows:

```
defaults

     do_not_generate
         llh_add ;
```

This is a viable solution, but not ideal. If you have to insert one line of code into a very large function, you must maintain the entire function in your trigger file just for that one line of code. For example, in `llh_add` above, you must maintain 10 lines of source code for that 1 custom line of code. Lookup functions (`llh_lookup` and `lld_lookup`) are large, line-intensive functions and frequently must be handled this way because one or two lines of code must be added to make it function a preferred way.

3. **You could use blocks.** Blocks allow you to go into a function and just modify one part of that function. Blocks are pieces of source code within a function that perform a specific task. They could be considered "sub-functions", i.e., modular "functions" within a function.

   Consider the unmodified `llh_add` again:

```
######################################################################
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()
```

There are four main tasks being performed within the `llh_add` function:

1. variables are defined;
2. the insert command is executed;
3. the serial number assigned to the record is displayed;
4. the rowid is re-set.

The source code that performs these tasks is grouped together in blocks. Blocks are groupings of code within a function that perform one task. Blocks in the source code are identified with block tags. Block tags are identified as:

```
#_{block_name}
```

The #_ identifies a block tag, much like a #_ identifies a trigger tag in source code. The following block tag that is seen at the top of `llh_add`:

```
#_define_var
```

identifies the group of source code that defines variables for `llh_add`. The Code Generator automatically places block tags into the generated source code.

This block tag seen in `llh_add`:

```
#_insert
```

identifies the source code that inserts the record.

This block tag:

```
#_serial
```

identifies the group of code that displays the assigned serial number.

Finally, this block tag:

```
#_rowid
```

identifies the group of code that re-sets the rowid.

Anything after the space following the block tag is considered a comment and not read by the Featurizer. Thus for the following block tag:

```
#_define_var - define local variables
```

the "- define local variables" is a comment and not read by the Featurizer.

The `#_on_disk_add` that you see in `llh_add` is *not* a block tag. It is a *trigger* tag, and it is for you to insert custom INFORMIX logic via a trigger (in this case, the `on_disk_add` trigger).

Observe in `llh_add` above the `#_define_var` and `#_local_var` block tags. There can be blocks within blocks. Blocks are delimited by indentation. For instance, the `#_insert` block ends when it encounters the next block *at the same level of indentation*:

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

#_serial - Bring back the serial field & display it
```

Here, the `#_insert` block is terminated with the appearance of the `#_serial` block.

The #_define_var block ends at the #_insert block, not the #_local_var block, because the entire #_local_var block is indented within the #_define_var block:

```
#_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
```

The #_local_var block is terminated with the # Set the serial field commented line, because # Set the serial field *out-indents* the #_local_var block. The #_define_var block is terminated with the #_insert block.

The Code Generator automatically places block tags into the source code that it generates. All lines of source code created by the Code Generator are within blocks. So not only do you have known locations to place logical modifications in source code (through triggers), but you also have control over every line of source code and can make physical modifications (through blocks) to the code.

If blocking conventions aren't followed, the entire source code file is regarded as one block. If block conventions are followed, a source code file may be divided into as many blocks as you desire.

When to Use Blocks

It is important that you gain an understanding of when to use triggers and when to use blocks. This requires an understanding of the difference between "logical" areas of code, and "physical" areas of code.

Logical areas of code can be defined as *anywhere* in the program that a certain event takes place (such as leaving a field, entering a field, writing to disk, etc.). These logical points in the code are defined as "triggers."

Physical areas of code can be defined simply as the address of certain lines of code in the .4gl file. There is a big difference between a physical location and a logical location. Fitrix *Screen* can change quite substantially and still offer the same logical locations (disk write, after field, etc.). Not so for physical locations.

If you write your modifications in triggers (as opposed to block commands), the likelihood that your modification will work in future versions is much greater than if you write your modifications using block commands. We will strive for 100 percent forward compatibility for trigger code. Bottom line: *use triggers whenever you can*. Only use block commands if there's no trigger for what you're doing.

Also keep in mind the physical/logical difference when defining your blocks. Try to describe them in logical terms (what you're doing) vs. physical terms (how you're doing it). Block tags are fairly reliable reference points in the generated code and they should not change with future releases. However when you use strings in block commands to locate and delineate blocks, you gamble that those specific strings won't change during future releases. Since most code is enhanced over time, block commands utilizing strings can not be guaranteed to always be compatible with future releases.

Note

Triggers and blocks can both be put in .ext files, while only triggers can be placed into .trg files.

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. Unlike, triggers, which are placed in .trg files, 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. For more examples refer to "Block Manipulation Examples" on page 13-48. 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()
```

You would use the `before block` command to add this one extra line before the "insert" block. Thus in the .ext file, you would 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 would go 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]
```

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

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

The result in the function `llh_add` would be:

```
#_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()
```

Note that all block commands in .ext files are delimited by semicolons, just like triggers 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 would result 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()
```

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:

```
# 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()
```

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/NULL 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 `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 exactly as the `at_eof` trigger acts—it puts text at the end of files. "TOF", "EOF", and "NUL" must all be uppercase.

See "Block Command Statements" on page 13-23 for a full list of all block commands, their syntax, and examples.

Block Command Files (.ext files)

As mentioned earlier, block commands are placed in .ext files much like triggers are placed into a .trg file. There is some philosophy behind .ext files that makes them a little bit more complicated than .trg files. Basically .ext files serve two purposes: the first is to provide a means of plugging and unplugging features; while the second is to simply hold block commands which always need to be merged into the basic program.

Also, .ext files can contain triggers. This allows you to create independent features.

Note

An .ext file can be named with any combination of letters, numbers and underscores. You cannot 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 13-32.

Specifying Which .ext Files to Merge (`base.set` files)

Unlike .trg files which get merged automatically, 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 13-32.

Specifying Files for Blocks to Work (`start file`)

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:

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

Example:

```
start file "midlevel.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 `midlevel.4gl`. There are no triggers that allow you to add custom logic to functions classified as midlevel, but you can do it with blocks. Here is an example of `mlh_clear` in `midlevel.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" block command to `midlevel.4gl`:

```
    start file "midlevel.4gl"

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

The result of the above block command would be in `midlevel.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** (or NULL) block ID means that there is no associated block tag for this command.
- **a_<*field_name*>** can be used to target "#_after_field <field>."
- **b_<*field_name*>** can be used to target "#_before_field <field>."
- **c_<*field_name*>** can be used to target "#_after_change_in <field>."
- **e_<*event_name*>** can be used to target "#_on_event <event>."

Block Command Statements

This section lists the syntax of each Block Command Statement and it’s 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 Files for Blocks to Work (`start file`)" on page 13-21.

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 50 characters max. 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 50 characters max. 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 should be exercised when using the delete block command since it deletes all existing block tags within the specified block, thus making it difficult to maintain regenerability.

`function define <`*`function name`*`>`

Like the define trigger, 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 trigger variables.

Note

Semicolons: All block commands except "delete block" require additional text (trigger code) 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, because there is no trigger code associated with the block command.

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 triggers and 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 "string", 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", and "thru/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

You may use double quotes in a string block command as long as you backslash it.

The following example DOES NOT work:

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

Backslashing 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

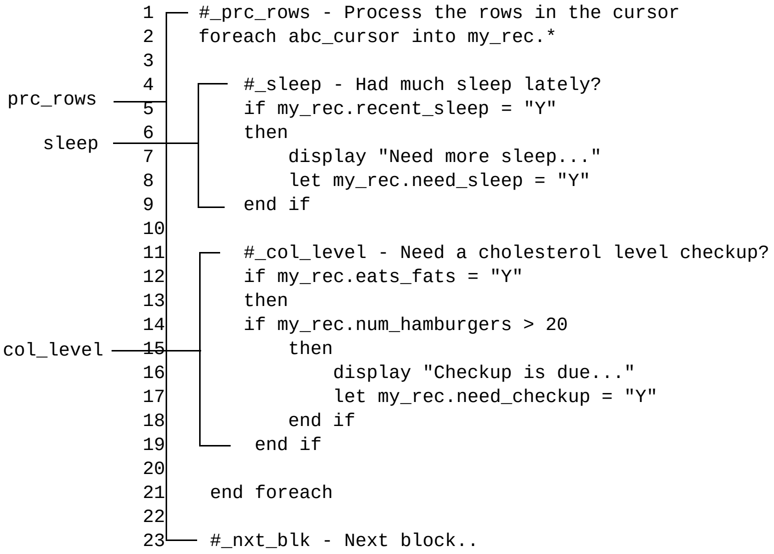
C - An "end function" statement as the first words of the line, or

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` *`blockname`* marker. This is not required by the Featurizer, but it is a convention that should be practiced.

Example: (notice line #19)

```
 1   #_bldcmd - Build the shell command to run that gets a list of
 2   # all .trg and .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 ’?*.trg’ -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.

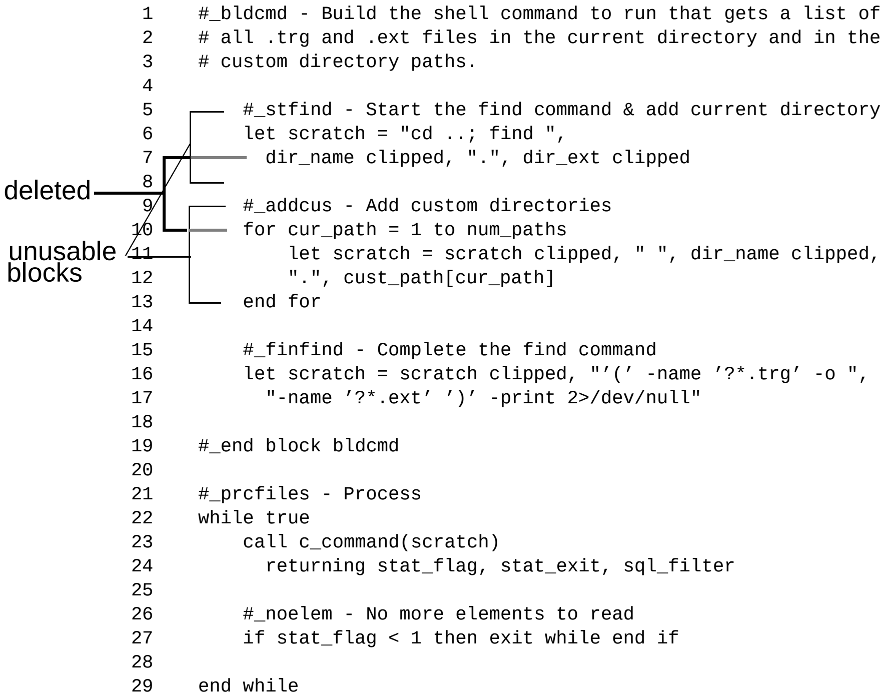
Note on Block Replace and Block Delete

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).

Example: If the following command is specified:

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

on the following file:



```
 1   #_bldcmd - Build the shell command to run that gets a list of
 2   # all .trg and .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 ’?*.trg’ -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
```

Given the file above, lines 7 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 markers contain no white space.
- Block markers must be unique to 20 characters.
- Block markers should be long enough to uniquely identify the block within the function and still be somewhat readable (i.e., no #_zz and no #_real_wordy_block_identifiers, even if they are unique to 20 chars).
- Block markers must consist of only the following characters: [0-9], [a-z], [A-Z], or _(underscore).
- By convention, block markers are lowercase letters followed by a space-dash-space, followed by a verbal block description starting in an uppercase letter.

Examples:

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

The block identifier for a block should never change. The description can change, the code in the block can change, but not the identifier. Others may key off this identifier.

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. .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. Source code is either in the form of blocks or triggers. See the previous section for a discussion of block commands. 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 or triggers. You can specify multiple start file 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.

Extension (.ext) files are very similar to .trg files, except they contain code that drives *one specific feature*. Trigger files mainly contain modifications made to a specific screen, while .ext files contain code that may effect many screens.

Extension files are *not* tied to screens like .trg files. The trigger concept requires that .trg files correspond with the .per files they work on. Thus .trg files have the same prefix as .per files.

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

Note

Extension file names must consist of only the following characters: [0-9], [a-z], [A-Z], and _(underscore).

As an example, we use the `order.trg` file from the `$fg/codegen/demo.4gm/screen5.4gs` program directory. This program contains the order form program, which enters data into the `stores` database. The following is added trigger logic that drives an "approval" feature that requires entry of an approval code on all orders over $500.00. Here is the conventional way the feature is coded into a .trg file (see the `after_input` and `at_eof` trigger):

```
defaults
     switchbox_items
         cust S_cust;

 input 1
     static_define
         upd_cust_filter   char(40);

     after_input
         if p_orders.t_price >= 500.00
         then
             call need_approval()
         end if ;
     on_event add_cust
         call add_on("cust", "A", "", "");

     on_event update_cust
         if menu_item = "update"
         then
             let upd_cust_filter = "customer.customer_num = ",
                 p_orders.customer_num clipped
             call add_on("cust", "U", upd_cust_filter, "")
             let p_orders.customer_num = get_vararg()
             if llh_lookup("customer",false) then end if
             call llh_display()
```

```
        end if;

    after_field customer_num
        if p_orders.customer_num = 0
        then
            call add_on("cust", "A", "", "")
            let p_orders.customer_num = get_vararg()
        end if;

    at_eof

#######################################################################
function need_approval()
#######################################################################
#
#  this function prompts the user for an approval code.  the user
#  is kept within the input command until the proper approval code
#  is entered.
#

    define
        code_entered char(6)

    prompt "Order is over $500.  Enter the approval code: " for code_entere

    if code_entered != "denver"
    then

       error "You entered the wrong approval code. Press [ESC] to try again
        let nxt_fld = "customer_num"

    end if

end function
# need_approval()
    ;
```

Notice how the "approval" feature is interspersed with the other features and embellishments that you see in the trigger file: invoking the add-on screen to add a customer, invoking the add-on screen to update a customer, etc.

Now, with .ext files, the specific logic that drives the "approval" feature can be taken out of the trigger file and placed into an .ext file. This way you can easily "plug in" or "unplug" this feature from your different applications. Here is the "approval" feature coded entirely in the "approval.ext" file:

```
start file "header.4gl"

     after_input
         if p_orders.t_price >= 500.00
         then
             call need_approval()
         end if ;

     at_eof

 #######################################################################
 function need_approval()
 #######################################################################
 #
 #  this function prompts the user for an approval code.  the user
 #  is kept within the input command until the proper approval code
 #  is entered.
 #

     define
         code_entered char(6)


    prompt "Order is over $500. Enter the approval code: " for code_entered

     if code_entered != "denver"
     then
         error "You entered the wrong approval code. Press [ESC] to try
again."
         let nxt_fld = "customer_num"

     end if

 end function
 # need_approval()
     ;
```

Unlike a .trg file, an .ext file has no sections (defaults, input 1, or input 2). Therefore, a "start file" command is always issued in an .ext file to indicate which file to insert the code.

Note

When merging code, the Featurizer always merges the .trg files *first*, and then the .ext files. This is important because you may have code in your .trg file that conflicts with your .ext file 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 "instvals" 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 trigger or block is merged into a 4gl file, a .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 a .org file, the Code Generator creates a new .org file with the same filename prefix. If a .org file is not found, a .4gl file is created instead.

The Featurizer and .org Files

Whenever the Featurizer merges a block or a trigger into a .4gl file that does not have an associated .org file, a .org file is created by copying the .4gl to a .org. If a .org file does not exist for a specific .4gl, such as `header.4gl`, the Featurizer assumes that this particular .4gl does not have any triggers or blocks in it. The Featurizer then copies that `header.4gl` file to a `header.org` file. Once a .org file exists, the Featurizer loads the .org, merges the triggers and 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 Triggers and Blocks from Existing .4gl Files

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

Say you once had a `screen.trg` file with an `after_field` trigger that has already been merged into `header.4gl` and you decide you no longer want it. All you have to do is remove that trigger 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 it's original generated state. The Featurizer does the same if you once had a block or trigger specified in a .ext file, and then decided to remove it.

General Flow of the Featurizer

The following describes the operational flow of the Featurizer.

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

   All .trg and .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.**

   This step is either very simple or fairly complex. 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 .trg and .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 step #2 is complete.

Second, the Featurizer checks each .trg and .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 re-merges that file.

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

A. 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 filename, or by replacing any 3 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 a .org in the current directory. The name of this .org file is the same as the destination filename with any 3 character extension replaced by ".org." If the destination filename does not have a 3 character extension, then .org is appended to the filename to determine the .org filename (up to 14 characters).

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

Commands (CMDs) are triggers and block commands stored in the .trg and .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. For more information on the order CMDs get merged refer to "Version Control" on page 16-1.
- All CMDs in one directory are processed before any CMDs in another directory in the search path.

- Within any directory, CMDs located in .trg files are merged before CMDs located in .ext files. In other words, triggers are merged before blocks.
- 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 .trg and .ext files.

Note

Triggers are physically implemented as "replace block" commands. Any trigger that was inserted in a prior CMD is replaced if that same trigger is defined in a later CMD.

C. 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.

D. 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’s are present, the Featurizer copies the .tmp files into .4gl files.

Do not use .tmp extensions for your own files. If you do, your files will be removed.

Filename Extensions

Extension	File Explanation
.4gm	Application module directory (A/R).
.4gs	4GL source code directory.
.4gc	General custom 4GL source code directory.
.abc	Example used for specific (non .4gc) custom source code directories.
.4gl	4GL source code file.
.4go	RDS compiled 4GL object code file.
.o	C Compiled 4GL object code file.
.4ge	Executable program (run directly from the O/S).
.4gi	Executable program (run from the fglgo or fgldb runners).
.per	Source code for a data entry screen.
.frm	Compiled representation of the .per file.
.trg	Trigger file associated with a screen.
.ext	Source code extension file associated with a plug in feature.
.set	File that contains the list of features in a feature set.
.opt	File that defines the functionality of certain triggers.
.tmp	Reserved for use by the Featurizer and Code Generator.
.org	File that contains the original generated code before the Featurizer merge.

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.

Featurizer Environment Variables

`$fg`: Path to the Fitrix *Screen* install directory (used to find executables so you do not have to be within `$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 .trg and .ext files to merge. For more information refer to "Version Control and the Code Generator" on page 16-8.

`$feature_set:` This optional variable contains the name of a *.set file to use.

`$force_merge:` If set to Y, `fglpp` re-merges all triggers and blocks regardless of time stamps.

`$FGLPPDIR:` Directory containing the `fglpp` executable program.

`$FGLPPOPTIONS:` Directory containing global options file (`fglpp.org` and `fglpp.opt`) with default variable settings. This is used to set arguments such as -C for running fglpp from `fg.make`.

`$fglppflags:` Contains extra flags to pass to the `fglpp` program (whether called with `fglpp`, `fg.make`, etc.) such as -C.

The following backwards compatibility flags are discussed in detail in "Maintaining Backwards Compatibility—The Options Files" on page 2-19.

`$define_trig:` If set to "replace," `define` and `static_define` triggers are replaced by subsequent triggers in the `$cust_path`.

$at_eof_trig: If set to "replace," at_eof triggers are replaced by subsequent triggers in the $cust_path.

$swbox_trig: If set to "replace," swbox_trig triggers are replaced by subsequent triggers in the $cust_path.

$fglpp_fatal_warn: If set to Y, fglpp gives a fatal error if a missing block is found.

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
#_ block definitions in one file	1000	A
lines in the (.org + .trg + all .ext’s)	7500	A,C
triggers and block CMD’s for an .org file	unlimited	D
characters in one line	unlimited	E
block nesting levels	10	A

A This number represents an internal program array limit. It can be expanded in future versions if the number is found inadequate.

B By convention

C This limit represents the total number of lines (excluding blank lines) in the .org file plus the number of lines of code from all .trg .ext files that reference this .org.

D Any number of triggers and block commands may be applied to an .org file—as long as the total number of lines doesn't exceed the limit specified in (C) above.

This represents the number of characters to the right of the indentation level. If this number exceed 70 characters, the lines are (internally) split into as many 70 character lines as necessary. The only effect is that each split internally consumes a new line (of which there are a limited number—see (C) above). By convention, we try to keep our right margin at 70 characters or below for aesthetic purposes. We use the row of 70 pound signs (#) surrounding function declarations as a margin guide.

Featurizer 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: The Featurizer keeps displaying a message stating that my 4GL source is newer than my trigger file and skipping my trigger. Why is this happening and what should I do?

Answer: The Featurizer is designed to behave in a similar fashion to the `make` and `fg.make` utilities. It knows if a trigger file has changed since the associated source was last changed and will not merge a trigger that is older than its associated source code. This was done to prevent slowdowns during compilation and linking due to merging of triggers.

If `fg.make` is run with the `-mf` it causes the trigger and 4gl time stamping logic to be bypassed.

If the utility is run directly, it can be invoked with the `-force` option. This causes a forced merge to occur.

Specific triggers can be forced to merge by either writing them in vi or using the touch utility. Use of the touch utility on a file which does not exist creates a zero length trigger file. This causes the utility to remove triggers from your 4GL source.

When the Featurizer is invoked from the Code Generator, a forced merge is used because the newer 4GL code is triggerless and needs to be merged to be brought up to date. The time stamps on the files immediately following code generation do not reflect the current stamp of the 4GL source files relative to the trigger files.

Question: What changes to my program require regeneration of my program vs. 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 schemas.

Question: Are comments acceptable in my triggers files?

Answer: Comments are acceptable in most cases.

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

Answer: Set the environmental variable `no_merge=Y` and export it to your environment.

Example:

```
no_merge=Y; export no_merge
```

Question: How do I force a merge of my trigger files if I just want the merge to always be run when I run fg.make?

Answer: Set the environmental variable `force_merge=Y` and export it to your environment.

Example:

```
force_merge=Y; export force_merge
```

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.

Block Manipulation Examples

The following are some examples of block manipulation commands. These examples use the following function. This function would typically be found in an .org file.

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
        let dup_prep = "Y"
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

In the above example, the function-id is `llh_dupchk`, and there are two block-ids specified, named `dup_sql_stmt` and `prepare_sql`. What follows are examples of some block commands, the impact on the above code, along with any Explanation:.

1. after block *<function name> <block ID>*

The .ext file:

```
after block llh_dupchk dup_sql_stmt
  display "CODE IS PLACED HERE";
```

Resulting code

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
        let dup_prep = "Y"
        let scratch = "select rowid from customer "
    end if
    display "CODE IS PLACED HERE"

end function
# llh_dupchk()
```

Explanation:

Based on the definition of what signifies a block, the block `dup_sql_stmt` is terminated by the "lesser" indentation of the line "end function." Hence, any code that is placed after the block `dup_sql_stmt` is placed after the `end if` line.

2. before block <*function name*> <*block ID*>

The .ext file:

```
before block llh_dupchk dup_sql_stmt
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    display "CODE IS PLACED HERE"
    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
        let dup_prep = "Y"
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

Explanation:

Block code comes before the specified block tag.

3. before block <*function name*> <*block ID*>

The .ext file:

```
before block llh_dupchk prepare_sql
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        display "CODE IS PLACED HERE"
        #_prepare_sql - the preparation of the sql
        let dup_prep = "Y"
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

Explanation:

The custom code is put before the block tag `prepare_sql`.

4. after block *<function name> <block ID>*

The .ext file:

```
after block llh_dupchk prepare_sql
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
        let dup_prep = "Y"
        let scratch = "select rowid from customer "
        display "CODE IS PLACED HERE"
    end if

end function
# llh_dupchk()
```

Explanation:

The end of the block named `prepare_sql` is the `let scratch =` line, based on the "lesser" indentation of "end if" line which follows it. Hence the code is put after the `let scratch =` line.

5. in block <*function name*> <*block ID*> before "*string*"

The .ext file:

```
in block llh_dupchk prepare_sql before "let dup_prep"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
        display "CODE IS PLACED HERE"
        let dup_prep = "Y"
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

Explanation:

From the block tag specified, a pattern/string is searched for. The block command keyword of "before" specifies the .ext code to be inserted before the pattern/string specified. If the pattern/string could not be found, an error is generated. If there is more than one occurrence of the string found, the 4GL Pre-processor uses the first occurrence. In other words, if the string specified had been "let" instead of `let dup_prep`, the above results would have been the same.

6. in block <*function name*> <*block ID*> after "*string*"

The .ext file:

```
in block llh_dupchk prepare_sql after "let"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
        let dup_prep = "Y"
        display "CODE IS PLACED HERE"
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

Explanation:

From the block tag specified, a pattern/string is searched for. The block command keyword of "after" specifies the .ext code to be inserted after the pattern/string specified. If the pattern/string could not be found, an error is generated. If there is more than one occurrence of the string found, the 4GL Pre-processor uses the first occurrence. In other words, if the string specified had been "let" instead of `let dup_prep`, the above results would have been the same.

7. **replace block <*function name*> <*block ID*>**

The .ext file:

```
replace block llh_dupchk prepare_sql
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
            display "CODE IS PLACED HERE"
    end if

end function
# llh_dupchk()
```

Explanation:

The whole block named `prepare_sql` is replaced. Notice that the original block tag is left intact.

8. **replace block *<function name> <block ID>***

The .ext file:

```
replace block llh_dupchk dup_sql_stmt
 display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
        display "CODE IS PLACED HERE"

end function
# llh_dupchk()
```

Explanation:

The whole block named `dup_sql_stmt` is replaced. Notice that the original block tag is left intact.

9. **replace block <*function name*> <*block ID*> after "*string*"**

The .ext file:

```
replace block llh_dupchk dup_sql_stmt after "if dup_prep"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    display "CODE IS PLACED HERE"

end function
# llh_dupchk()
```

Explanation:

Notice this block goes to a certain point in a block, that point denoted by a match in the supplied pattern/string, and then replaces from that point on, the existing block with what was supplied as block text. This is similar to the `in block` command, except it does replacement as opposed to insertion.

10. replace block <*function name*> <*block ID*> from "*string*"

The .ext file:

```
replace block llh_dupchk dup_sql_stmt from "if dup_prep"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    display "CODE IS PLACED HERE"

end function
# llh_dupchk()
```

Explanation:

Same as example #9, except using the `from "string"` causes the line containing the matching string to be replaced as well. In example #9, the use of `after "string"` causes the line containing the matching string to be preserved, and not be a part of the replacement.

11.replace block *<function name> <block ID>* from "*string*" to "*string*"

The .ext file:

```
replace block llh_dupchk dup_sql_stmt
from "if dup_prep" to "let scratch"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    display "CODE IS PLACED HERE"
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

Explanation:

Notice code is replaced for the line containing the matching `from string`, up to, but not including, the line containing the `to string`.

12. replace block <*function name*> <*block ID*> from "*string*" thru "*string*"

The .ext file:

```
replace block llh_dupchk dup_sql_stmt
  from "if dup_prep" thru "let scratch"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    display "CODE IS PLACED HERE"
    end if

end function
# llh_dupchk()
```

Explanation:

Notice code is replaced for the line containing the matching `from string`, up to, and including, the line containing the `thru string`. Also notice that the `replace` block command above is split onto two different lines. You are allowed to do this, as long as a keyword like "after", "from", "to", or "thru" start this new line, as it does in the above example (the keyword "from").

13. **replace block <*function name*> <*block ID*> thru "*string*"**

The .ext file:

```
replace block llh_dupchk dup_sql_stmt thru "let scratch"
  display "CODE IS PLACED HERE";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer
    display "CODE IS PLACED HERE"
    end if

end function
# llh_dupchk()
```

Explanation:

Code is replaced from block starting point thru/through "string."

14.delete block <*function name*> <*block ID*>

The .ext file:

```
delete block llh_dupchk dup_sql_stmt;
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

end function
# llh_dupchk()
```

Explanation:

The block is deleted. Notice that the block-id tags are removed as well.

15.delete block <*function name*> NUL

The .ext file:

```
delete block llh_dupchk NUL;
```

Resulting code:

nothing

Explanation:

The function is deleted. Notice that the block-id tags are removed as well.

16. **delete block <*function name*> <*block ID*> from "*string*" to "*string*"**

The .ext file:

```
delete block llh_dupchk dup_sql_stmt
from "#_prepare_sql" to "let scratch";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        let scratch = "select rowid from customer "
    end if

end function
# llh_dupchk()
```

Explanation:

The block is deleted from target string to target string.

17. delete block <*function name*> <*block ID*> after "*string*" through "*string*"

The .ext file:

```
delete block llh_dupchk dup_sql_stmt
after "#_prepare_sql" through "let scratch";
```

Resulting code:

```
##############################################################
function llh_dupchk()
##############################################################
#
    define
           dup_rowid integer

    #_dup_sql_stmt - the sql used to check for duplicate rows
    if dup_prep is null
    then
        #_prepare_sql - the preparation of the sql
    end if

end function
# llh_dupchk();
```

Explanation:

The block is deleted, preserving the `after "string"`, but removing the `through "string"`.

18. before block TOF NUL

The .ext file:

```
before block TOF NUL
  display "CODE PUT HERE";
```

Explanation:

Using the function name value of TOF (top of file) allows you to put code at the very top of any file. In this case, since this is an "input 1" section, the file modified is header.4gl. Since there isn’t a valid function name to search for, you must specify the keyword NUL for the function name.

19.**before block NUL <*function name*>**

The .ext file:

```
before block NUL prepare_sql
  display "CODE PUT HERE";
```

Explanation:

Since the block ID is NUL, the search for function name (which is "prepare_sql") starts from the top of the file, and "acts" on the first block tag that matches the function name.

20.**after block EOF NUL**

The .ext file:

```
after block EOF NUL
  display "CODE PUT HERE";
```

Explanation:

Block code is put at end of file. This is the command that is issued when you use an `at_eof` block as well. This is to accomplish an `at_eof` block in an .ext feature set file.

21.**function_define <*function name*>**

The .ext file:

```
function_define main
  new_var smallint;
```

Explanation:

The result is the variable `new_var` added to the list of local variables defined in the function named `main`. If no local variables exist before merge, and consequently no `define` keyword in the function `main`, then the `define` keyword is added, along with the block variables specified.

22. before block *<function name>* b_*<block ID>*

The .ext file:

```
before block llh_b_field b_customer_num
  display "Hello world";
```

Explanation:

The b_ special block marker allows you to target the #_before_field block tag.

23. before block *<function name>* a_*<block ID>*

The .ext file:

```
before block llh_a_field a_customer_num
  display "Hello world";
```

Explanation:

The a_ special block marker allows you to target the #_after_field block tag.

24. before block *<function name>* c_*<block ID>*

The .ext file:

```
before block llh_a_field c_customer_num
  display "Hello world";
```

Explanation:

The c_ special block marker allows you to target the #_after_change_in block tag.

14

Compiling and Running Your Programs

This chapter covers the following:

n Compiling your programs

n Using the `fg.make` script

n Compiling and linking libraries

n Compiling your entire application

n Executing the final program

Compiling Generated Code

"Compiling code" means turning Informix forms, 4gl source code, and triggers into a working program. Fitrix *Screen* provides the facilities to do this for a single program or for an entire set of programs.

The script for compiling your generated code is `fg.make`. It has the capability to compile individual programs, all the programs in a module, or even an entire application. The `fg.make` script is capable of compiling programs into pseudo-code (commonly called p-code) object files, if you have the INFORMIX-4GL Rapid Development System, as well as compiling using the Informix c4gl compiler.

If you have both Informix products on your system (4GL and RDS), `fg.make` assumes that you want to compile using the RDS p-code compiler. If you wish to override this behavior, use the -F flag (e.g., `fg.make -F`) to force `fg.make` to use the c4gl compiler.

In general, the steps `fg.make` goes through are:

1. If run at the application level, it compiles each module listed in the application `Makefile`.
2. If run at the module level, it compiles each library and program listed in the module `Makefile`.
3. In a library, `fg.make`:

   a. converts form source (.per) files to form (.frm) files;

   b. converts 4GL source (.4gl) files to object (.4go or .o) files;

   c. loads object files into the archive (.a file or .RDS directory);

   d. removes the object files produced in step b.

4. In a program directory, `fg.make`:

   a. merges trigger logic from .trg and .ext files with .org files to produce .4gl files.

   b. converts form source (.per) files to form (.frm) files;

   c. converts 4GL source (.4gl) files to object (.4go or .o) files;

d. links the object files together with objects from library archives listed in the program `Makefile` to produce the program (.4gi or .4ge) file.

Note

The `fg.make` script requires that the standard UNIX `make` utility be present on every machine it runs in order to determine whether a file needs to be compiled or not. If your machine does not have a C Development System, then you need to copy the `/bin/make` (or equivalent) script from a machine that contains the C Development System to the machine where you are running your applications.

Differences Between RDS and 4GL Compilation

INFORMIX-4GL can be compiled into two different forms: a binary executable (machine specific) program, or a pseudo-code file that is interpreted by a "p-code runner" program. The process to produce the first, is called a "4GL compile," while the process used to produce the second is called an "RDS compile."

During the 4GL compile, the 4GL source code files (extension .4gl) go through several transformations. The .4gl file is first transformed by an Informix program called `fgl`c to an Informix ESQL-C file (.ec). The file is then transformed into a pure C code file (.c). At this point, compilation is turned over to `cc`, the UNIX C compiler on your system. It produces an object file (.o). Finally, `cc` runs `ld`, the UNIX linker which links .o files with each other and with objects stored in a library archive file. This process produces a binary file (.4ge) that is directly executable on your computer.

The 4GL compilation process:

Compile phase

.per → .frm

.4gl → .ec → .c → .o

Link phase

.o libraries (.a) → .4ge

The INFORMIX-4GL Rapid Development System (RDS) goes through a somewhat different process. An Informix program called `fglpc` transforms the .4gl file into a p-code object file (.4go). These p-code object files need only be concatenated using the UNIX `cat` command. However, there is no Informix program to find p-code objects located in libraries. Instead, a shell script called `linkrds.sh` is used. It emulates the behavior of `ld` and searches the library archives specified in the program `Makefile` to locate .4go files needed to complete the compile.

This process produces a p-code file (.4gi) that is interpreted by an Informix program called `fglgo`. (Note that Informix also provides a p-code debugger program called `fgldb` that can interpret the .4gi file).

The RDS compilation process:

Compile phase

.per → .frm

.4gl → .4go

Link phase

.4go libraries (.RDS) → .4gi

By the way, as a developer's tool, RDS is wonderful. It has a first class debugger, it compiles quickly, and the p-code it produces is completely portable between machines. RDS generally works very well for your end user, as well. It is an excellent idea to have RDS and the RDS Debugger on your users' or customers' systems.

Using `fg.make` to Compile Your Program

The `fg.make` shell script assumes that a file called `Makefile` exists in your current directory. In the program directory, this `Makefile` is one of the generated files. The `Makefile` is discussed on page 14-11.

In itself, `fg.make` is not a terribly complicated script. It has two purposes: it accepts command line flags and uses them to set up some environment variables, and it runs the appropriate program to do the actual compiling. The programs that do the compiling use the 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.

Here's an example: As mentioned earlier, if you have both RDS and 4GL on your system, `fg.make` assumes you wish to compile using RDS unless you use the -F flag. The underlying environment variable is called `make_method`. If you set this variable to "4GL", then `fg.make` defaults to the 4GL compile. Then, to do an RDS compile, you use the -R flag (e.g. `fg.make -R`).

Here is a brief summary of all the flags available with `fg.make`. Each of these flags can be set to the default option by setting a variable in your environment.

Usage:

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

-h Prints a help message.

-F (4gl compile): This flag tells `fg.make` to override the default and perform a 4gl compile. Environment variable equivalent: `make_method=4gl`.

-R (rds compile): This flag tells `fg.make` to perform an RDS compile. Environment variable equivalent: `make_method=4gl`.

-L *library* The -L flag allows you to specify the names of any additional libraries you want to link in. These libraries will appear in your `Makefile` above the upper level libraries. Environment variable equivalent: `xtra_lib=`*`library`*.

-M *makefile* This flag allows you to specify a name other than "Makefile" for the Makefile. This is useful when testing.

-T *type* This flag lets you specify which type of makefile to create. You can create the following types of Makefiles: application, module, library, program, shell, and make.

`-m{n|o|f|fo}` (merge): The -m flags allow control over how the Featurizer is run.

`-mn` (no merge): The -mn flag prevents `fg.make` from performing a merge. The Featurizer will not be called. Environment variable equivalent: `no_merge=y`.

`-mo` (merge only): This runs the Featurizer without a subsequent compilation. Environment variable equivalent: `merge_only=y`.

`-mf` (force merge): This flag overrides the time stamp comparison logic and forces a merge. Environment variable equivalent: `force_merge=y`.

`-mfo` (force merge only): This flag is used to force a merge and override the time stamp comparison logic without compiling or linking.

`-o` ***`execname`*** In library compiles, this specifies the name of the target archive (outname.a or outname.RDS). In program compiles, this specifies the program name (`outfile.4ge` or `outfile.4gi`). It strips off any extensions you might add to it. This is useful for testing.

`-l`

(link only): RDS only. When compiling under RDS, the -l flag instructs the program to link the object files together into a .4gi file, with no checking for modification between .4gl and corresponding .4go files.   Environment variable equivalent: `link_only=y`.

The -l flag is used when a local .4gl source file has been modified and compiled (with `fglpc`) into a `.4go` object file, with the remainder of the application source code held constant.

This flag causes `fg.make` to skip the `fglpc` and `form4gl` parts (i.e., compilation of 4gl files and form files is skipped) and run only the link part of the `fg.make` suite of shell scripts. fg.make run in `link_only` mode always rebuilds the `filelist.RDS` in the local program directory. When compiling a library, link_only means to just rebuild the library archive catalogs.

`-f`

(fast link): RDS only. Much of the work done by the `fg.make` script need not be done each time it is run for a program. The `linkrds.sh` part of the compile creates a list of files that must be concatenated with the local .4go files to create the .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 the next time `fg.make` is run. The Featurizer is still run when the `fast_link` option is used. Environment variable equivalent: `fast_link=y.`

`-D`

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

`-r` (recursive link): RDS only. The -r flag causes `linkrds.sh` to make multiple passes through the library list when making a program. If functions were not found on the first pass, they may be found on subsequent passes. This should never be necessary. It can be useful for debugging library problems. Also note that this could allow you to write non-portable code. This flag has no meaning if `fast_link` has been specified. Environment variable equivalent: `recursive_link=y`.

`-u` (list unresolved): RDS only. The -u flag causes `linkrds.sh` to warn the user of any function calls it was unable to resolve. This flag has no meaning if `fast_link` has been specified. Environment variable equivalent: `list_unresolved=y`.

`-a` This flag causes all files to be recompiled regardless of dependencies. Environment variable equivalent: `no_use_make=y`.

`-c` When this flag is used in a program directory, `fg.make` stops after compiling the source code. The linking phase is skipped and no program is produced.

When this flag is used in a library directory, `fg.make` stops after it compiles the source code. The archive is not loaded (either the .a file or the .RDS directory).

`args` Objects to be compiled. The default is the list in the Makefile

Many of these flags work together. Some are mutually exclusive. `fg.make -mof (or -mfo)`, for example skips all compilation except the Featurizer trigger merge, and passes the force merge flag to the Featurizer. Likewise, `fg.make -fl (or -lf)` skips the compile phase and goes right to the linkage (-l) and uses the list of library files (filelist.RDS) produced by the last link rather than producing a new list (-f).

On the other hand, specifying -l implies -mn and overrides the -mo and/or the -mf flags. Likewise, if -R and -F are specified, the last one on the command line takes effect.

Just as an aside, the single character flags can be listed together and so can the -m flags if they are last in the crowd. This, for example, is legal: `fg.make -iur-flmof` where `mof` is equivalent to `-mo` and `-mf`. The three character flags must stand alone.

Speeding Application Compiling

When you are working with programs, you can speed up the compilation process by using only the parts of the process that relate to the changes you are making. You can make four different types of changes that might shorten the compilation time so that you don't have to run a complete `fg.make`. The assumption here is that you *are* using RDS because the main issue is time. If you are not using RDS, compiles are going to take a lot of time and these strategies do not save much of it. The four types of changes are:

1. A change to only certain source files, not all of them.
2. A "cosmetic" change to a screen form that doesn't change any of the data on it.
3. A change to the trigger files that require a re-merge of triggers.
4. A change to the `Makefile` that requires new libraries to be compiled into the code.

When you make these types of changes, you can control `fg.make` through the use of different flags, to only do the steps you need done, and/or use the programs that `make` calls. In doing so, you can dramatically cut the time it takes to test programs if computer speed is an issue.

Changing Only One Source File

This is the most frequent type of change. The `fg.make` utility only compiles source files that need it, but it checks them all and this can take time. It also builds a new list of library files needed in the compile. Here, it is assumed you are adding no calls to new library functions. Since you know what source files you have changed, you can specify these files as arguments.

```
fg.make -f name (.4gl is optional)
```

The -f is the fast link flag. This puts together the compiled local programs and all the library functions. It assumes you have run a complete fg.make at some time in the past on this program to create the list of library files (filelist.RDS) needed, that you haven't changed the Makefile to require a different set of libraries, and that the libraries haven't changed.

At this point you can run the program.

Cosmetic Form Changes

Cosmetic form changes, are those that adjust the position of information on the screen but doesn't change that information in any way. For these changes, you type the form4gl command with the name of the form file changed.

```
form4gl form
```

The above line creates a new compiled screen form or .frm file.

A Change to a Trigger File

Note

See the section on the "The Featurizer and Blocks" on page 13-1 for more information about merging triggers.

After a trigger file is made, it must be merged with the .org files that are generated from the generator. There are several flags on the fg.make command that merge triggers and then do various other steps in the compilation process depending on what your intention is.

When you just want to merge the .4gl file, you run the following:

```
fg.make -mo
```

The -mo flag means merge only and it just merges triggers and blocks and does nothing else.

Normally, fg.make checks to make sure that the .trg file is newer than the .4gl before it does a merge. In other words, it can see if a particular merge is necessary or not.

The following examples show how the time-stamp comparison logic works:

Example 1:

```
-rw-rw-rw    1 gordona  informix    3777 Aug  6 11:18 order.4gl
-rw-rw-rw-   1 gordona  informix     596 Aug  6 11:05 order.trg
```

A merge of `browse.trg` into `browse.4gl` is not performed because `browse.4gl` is more current than `browse.trg`.

Example 2:

```
-rw-rw-rw    1 gordona  informix    3777 Aug  6 11:18 order.4gl
-rw-rw-rw-   1 gordona  informix     596 Aug  6 11:25 order.trg
```

A merge of `browse.trg` into `browse.4gl` is performed because `browse.trg` is more current than `browse.4gl`.

When you want to force a merge, no matter what the file dates, use:

```
fg.make -mfo
```

The `-mfo` means merge forced only and it means that the `fg.make` utility performs only the merge and forces it even if dates on .4gls are newer than the triggers.

When you want to force a merge, but do all of the other compilation as well, use:

```
fg.make -mf
```

The `-mf` flag indicates a forced merge, but it doesn’t only do a merge. It does the following steps during compilation—creating object files, linking, and so on.

The Makefile

The `fg.make` script reads a description file that contains the information it needs to produce a program. By default the name of this file is `Makefile`.

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

```
# Screen Generator version: 4.11.UA1

# Makefile for an Informix-4GL program

#_type - Makefile type
TYPE = program
```

```
#_name - program name
NAME = davidh.4ge

#_objfiles - program files
OBJFILES = globals.o browse.o cust_zm.o detail.o header.o \
           main.o midlevel.o options.o stk_mnu.o stockzm.o

#_forms - perform files
FORMS = browse.frm cust_zm.frm order.frm \
        stk_mnu.frm stockzm.frm

#_custpath - version control path
CUSTPATH = djh:4gc:4gs

#_libfiles - library list
LIBFILES = ../lib.a \
           $(fg)/lib/scr.a \
           $(fg)/lib/scr.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."
```

The following variables are contained in a Makefile.

TYPE: The type of the Makefile. There are six types of Makefiles: program, library, application, module, shell, and make.

NAME: The name of the compiled executable. For an RDS compile, `fg.make` converts the extension to .4gi.

OBJFILES: The list of local object files to be linked together to produce the executable.

FORMS: The list of .frm files used by the executable.

CUSTPATH: The version control path used to create this executable. This macro does not appear in your Makefile if the `$cust_path` environment variable is not set before the generation.

LIBFILES: 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`. This can only be done when there have been no new function calls added but it does result in a faster compile.

The `fg.make` script automatically appends `c_lib.a` and `stubs.a` to the end of the list of library archives. If any library in the list does not exist, it is silently ignored.

GLOBAL: An entry for the `globals.4gl`. All local object files depend on this file.

all: This is a `make` rule. It is listed here to inform you not to use the UNIX `make` utility.

Modifying the LIBFILES Macro to Use Custom Libraries

You can modify the `LIBFILES` macro in your `Makefile` to use your own custom libraries by using triggers. Here are three examples:

The `libraries` Trigger

The `libraries` trigger places your library after the `../lib.a` line. For example, this trigger:

```
defaults

libraries
        ../../all.4gm/lib.a;
```

produces this `LIBFILES` macro:

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

The `custom_libraries` Trigger

The `custom_libraries` trigger places your library before the `../lib.a` line. For example, this trigger:

```
defaults

custom_libraries
          ../libadv.a;
```

produces this `LIBFILES` macro:

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

Changing the LIBFILES Macro with Block Commands

You can also 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/scr.a \
              $(fg)/lib/user_ctl.a \
              $(fg)/lib/standard.a
              ;
```

Using the -L Flag to Link Custom Libraries

All three of the methods described above result 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 have written some useful functions that you would like to put in a $fg/lib/standard.cus library directory (see LIBRARIES below). You can physically change your Makefiles using the methods shown above to change your Makefiles for all your programs, or you can use the -L flag with fg.make to avoid this:

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

This effectively acts as if you had changed the LIBFILES macro to look like this:

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

You can use more than one -L flag, for example:

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

The previous example produces the same effect as changing LIBFILES to look like this:

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

It is possible to modify the path name:

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

The previous example produces the same effect as changing `LIBFILES` to look like this:

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

You can add new libraries to the end. If you do, don’t use the period:

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

The previous example produces the same effect as changing `LIBFILES` to look like this:

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

and you needed to insert something in front of the second occurrence of `lib.a`, include more than the word "lib" in your prefix. The question mark can be used instead of the slash so `fg.make` does not interpret the slash to mean pathname.

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

The above line would produce the same effect as changing `LIBFILES` to look like this:

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

Here are the -L rules:

- The argument prefix is used to find where to insert the library.
- The argument suffix is used as part of the library name.
- A slash in the argument causes the argument to be treated as a path name. This doesn't affect where the name is inserted.
- The question mark can replace the slash if the slash is needed as part of the insertion criteria.
- If there is no match, the library is put at the end with no change.

Note

Wildcards can be used in the -L argument.

Linking in Libraries with `$cust_path`

The `fg.make` script automatically inserts libraries that correspond with entries in `$cust_path`. For example if `$cust_path` contains djh:4gc:4gs and there exists a `../lib.djh` library directory and its archive `../libdjh.a`, it will be inserted before `../lib.a`. Thus custom libraries need not be inserted specifically into the Makefile when using Version Control.

Compiling Libraries

Much of the RDS program compile parallels the 4GL compile. The form .per files transform into .frm files, the source .4gl files transform 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 4GL compiles.

With regard to Fitrix *Screen*, there are two classes of libraries. One class consists of the `scr`, `standard`, and `user_ctl` libraries, which provide the flow of control of generated programs and a number of specialized functions that provide the features of these programs.

Briefly, `scr` contains the flow control type functions. These are where the ring menus and the functions for adding, finding, updating and deleting documents are located. The `standard` library contains many common functions, many of which you might want to call yourself for various purposes. The `user_ctl` library of functions is used to extend the functionality of your programs in many ways. This is the only library where the source code is not provided. There is another library called `stubs` that supplies stub functions if the `user_ctl` library is not installed.

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 for modifying the behavior of functions provided with Fitrix *Screen*.

`fg.make` 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 of Fitrix *Screen*. 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 should be physically located, and what sequence it should be linked into your program.

Consider a library of functions that are common to a family of programs. You would have a program source directory for each program. Create a directory called `lib.4gs` along with your program directories. If you review the example `Makefile` above, you should note that the first entry in the `LIBFILES` list is `../lib.a`. This `lib.4gs` 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. Here's an example of a library `Makefile`.

```
########################################################################
# Makefile for an Informix function library
########################################################################
TYPE = library

LIBFILES = \
    $(LIB)(fn_name1.o)\
    $(LIB)(fn_name2.o)\
    $(LIB)(fn_name3.o)\
    $(LIB)(fn_name4.o)

FORMS = fn1_frm.frm fn2_frm.frm fn3_frm.frm \
        fn4_frm.frm

LIB = ../lib.a

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

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

TYPE: The type of the Makefile. There are six types of Makefiles: program, library, application, module, shell, and make.

LIBFILES: The list of object files to be put into the library archive.

FORMS: The list of .frm files used by the library functions.

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

```
LIB = ../scrcus.a
```

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

It is also possible to use the same name in the LIB macro for different libraries. In this example, your Makefile could contain `LIB = ../scr.4gs.` This would cause your objects to be loaded into the same archive as the code generated objects. Just remember you must recompile your library after a new installation.

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

To create the library archive, run `fg.make` in `lib.4gs`.

If you have any forms, they are converted into .frm files, which remain in `lib.4gs`. The open form statement in your function should probably say `../lib.4gs/fm_name` so any programs calling the functions are able to find the forms. Two other choices are to use the full path name, or to just use the form name, but include the library directory in `$DBPATH` when you run your program.

When `fg.make` performs 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 lib.a archive file in the directory above. It creates the archive if it doesn’t exist.

When `fg.make` does an RDS compile, it creates .4go files rather than .o files. These are then moved over to an archive directory, `lib.RDS`. 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 `lib.4gs` and `lib.RDS`. First, the .4gl source file is needed in the archive for creating the RDS catalogs. Second, it is convenient when using the debugger. Even if you have changed 4gl files in the library source directory, the 4gl files in the archive match the function objects that are linked into your program. In the debugger, this keeps the source consistent with the objects.

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 files in this directory and their functions. During the linking phase of a program RDS compile, `linkrds.sh` 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 `linkrds.sh` 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. `linkrds.sh` 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 should be rebuilt every time `fg.make` does an RDS compile in the library.

If you have modified a .4gl file in `lib.4gs`, but your modification does not include changes to function names, nor added, deleted, or changed function calls, it is not necessary to rebuild the catalogs in the .RDS directory.

There is a shortcut. When the -f (fast_link) flag is used with `fg.make` to compile a library, `fg.make` skips the catalog creation step. You can specify the specific files you wish compiled.

```
fg.make -f func1 func2 (.4gl extension optional)
```

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. As an example of applications, here are two: accounting and codegen. Examples of modules in the accounting application include general ledger, accounts receivable, payroll and quite a few more. 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 rcvbls`).

Compiling a Module

Put your program directories in the module directories. The names of these program directories would normally have an extension of 4gs. 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. Here is an example `$fg/myapplication/sales.4gm/Makefile` which 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` are compiled.

For example, if `invoice` is listed in the module Makefile and `$cust_path` = djh:4gc:4gs, then `invoice.4gs`, `invoice.4gc`, and `invoice.djh` are compiled if they exist. (They are compiled in reverse sequence of `$cust_path`).

Running Your 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 will address these arguments, and explain the invocation of programs compiled with 4GL and RDS. Later, the usage of the `run` UNIX shell script is explained.

Invoking Compiled Programs

The method of executing a program depends on whether INFORMIX-4GL or INFORMIX-RDS is used to compile the source.

The INFORMIX-4GL system 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 a UNIX prompt. The following is an example of the required syntax:

```
screen3.4ge [args]
```

INFORMIX-RDS compiles 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). The following provides an example of the required syntax:

```
fglgo <program_name.4gi> [args]
```

A number of command line arguments can be used when invoking a program generated by Fitrix *Screen*.

fglgo *program_name*.4gi [-dbname *database*] [order *order_by_clause*] [filter *filter_clause*] [-a] [A] [-u] [-U] [f]

-dbname Specifies the database to run against.

order Specifies the order of initial selection.

filter Limits the initial selection.

-a Enters directly into the Add mode.

-A Same as above only exit after adding.

-u Enter directly into the Update mode.

-U Same as above only exit after updating.

-f Enters directly into Find mode.

The database can be selected on the command line. Use the following argument to name the database in which the program will run:

```
-dbname database
```

Example:

```
fglgo screen3.4gi -dbname stores
```

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

Other command line arguments allow the user to pass a filter clause and order by clause to the program. This controls the selection and order of documents appearing on the data-entry form upon invocation. The filter argument only selects items in the main table for the header portion of the form.

You can define the initial filter for the selection of documents by specifying the filter on the command line. Use the following syntax:

```
filter "filter clause"
```

Example:

```
fglgo screen3.4gi 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 like the following example:

```
fglgo screen3.4gi filter "po_num='100'"
```

If the filter is "`1=0`", *no* rows are initially gathered. You may wish to pass the filter "`1=0`" if you are using the `-a` (Add mode) argument.

You can also specify a command line argument to sort the initial selection of documents. You may sort by any data entry column, though the columns must be in the main table. The syntax follows:

```
order "order by column(s)"
```

Example:

```
fglgo screen3.4gi order "po_num"
```

Nulls come before any other data so rows that have a null value for the "order by" column appear first. The column will be sorted according to ASCII conventions.

The following set of command line arguments control the mode in which the user enters upon start-up of the program, (-a, -A, -u, -U, -f). For example:

```
fglgo screen3.4i -a
```

puts the user directly into Add mode.

You can use these arguments to create new navigation events. For instance, you can define a navigation event as `Add a customer`. That event could be called from invoice entry, or any other part of the application. The order that arguments appear on the command line is insignificant.

Arguments must be separated by spaces.

Incorrect:

```
-Ud stores
-dstores
```

Correct:

```
-dbname stores
-A -dbname stores
```

Executing Programs When Using Version Control

When using Version Control, the following startup scripts should be used *instead* of `fglgo` or `fgldb` when executing your compiled code. Both of these scripts automatically set your `$DBPATH` variable so that the programs can correctly locate the necessary .frm files used with Version Control.

`fg.go`: runs the program created using Version Control. This script determines if the program is a .4gi or a .4ge and runs it accordingly.

`fg.db`: runs the program under the INFORMIX-4GL Interactive Debugger.

For more information refer to "Invoking Programs That Use Version Control" on page 16-20.

15

Advanced Features

This section explains how to perform a variety of modifications to your programs including:

- Event handling logic
- Creating custom libraries
- Creating application help text
- Creating BLOB field types
- Using skip field logic
- Cursor handling philosophy
- Generic text picker/editor
- The `fg_err` and `lib_error` functions
- Creating a post-processor
- Example of `ok_delete`
- Modifying `lib_message`

Event Handling Logic

Events are actions that can occur while a program is running, such as shelling out to the operating system, running another program, and displaying help text. An event describes the start or end of a particular activity. This section describes how event logic is handled by code generated with the Code Generator, and how you can add your own events to your programs.

Types of Event Handling Logic

There are two main types of events that can be performed within Fitrix *Screen* code:

1. **External events:** External events occur outside of the program. They are typically operating system commands such as viewing mail and checking disk space. The instructions for external events are UNIX operating system commands.

2. **Internal events:** Internal events are accessed within the program only. Examples of internal events include Zooms, Notes and User Definable Fields. Instructions for internal events are created with INFORMIX-4GL code.

   There are two types of internal events:

   1. **Local events:** Local events only occur at a particular point in a running application. Typically these internal events occur within an input statement. For instance, an event performed only when inputting on a detail record is considered a local event. This local event (and the hot key mapped to it) cannot be executed at any other point in the program.

   2. **Global events:** Global events are internal events that can occur at any point in the running application. Global events can occur:

   - when inputting onto the header record.
   - when inputting onto the detail record.
   - at the ring menu.
   - when executing a Zoom.
   - when selecting a navigation item.

- when performing any User Control feature, etc.

Event Flow

When hot_local() and hot_key() are called from local code, a plethora of activity occurs within these functions.

Note

The hot_local() function is only called if you use an on_event trigger.

Some of the key event handling functions and their flow are mentioned here (all of the event handling functions are located in the file $fg/lib/user_ctl.4gs/l_event.4gl. The following explains how scr_funct is set for a local event and how the navigate and hot key tables are used during event processing.

Flow for events:

1. Within the input screen (header or detail section), an event (any event) is executed for the first time. The hot_local() function is called.
2. hot_local(): If this is the first time hot_local() is called, hot_local() calls the event handling function event_init(), which:
   - Prepares and declares SQL statements for all of the above tables. SQL statements are not executed at this time, except for the SQL statement for the Hot Key Definitions Detail Table (stxhotkd); this statement is executed to fill an array (called key_map) telling what key is mapped to what event.
   - Loads the local event array (called local_evnt) with hardcoded local events. These events are tab, btab, cancel, accept, zoom, null

Note

Zoom is a hardcoded local event. Thus when generating code there is never a call to `hot_local()` for a Zoom. Once `event_init()` loads the event array with local events, `hot_local()` proceeds to add its event (passed to it as an argument) to this array. After all calls to `hot_local()` are through, the local event array contains all local events for the entire program. After all calls to `hot_local()`, `hot_key()` is called.

3. `hot_key()`: Its job is to map the key to the event.

   Back in local code before the calls to `hot_local()`, the global variable `hotkey` is set to the number of the key upon pressing that key:

   ```
   # Event trapping logic
   on key (control-b) let hotkey = 2  goto event
   on key (control-e) let hotkey = 5  goto event
   on key (control-f) let hotkey = 6  goto event
   on key (control-g) let hotkey = 7  goto event
   on key (control-i) let hotkey = 9  goto event
   on key (control-n) let hotkey = 14 goto event
   on key (control-o) let hotkey = 15 goto event
   on key (control-p) let hotkey = 16 goto event
   on key (control-t) let hotkey = 20 goto event
   on key (control-u) let hotkey = 21 goto event
   on key (control-v) let hotkey = 22 goto event
   on key (control-w) let hotkey = 23 goto event
   on key (control-y) let hotkey = 25 goto event
   on key (control-z) let hotkey = 26 goto event
   on key (f5)  let hotkey = 105 goto event
   on key (f6)  let hotkey = 106 goto event
   on key (f7)  let hotkey = 107 goto event
   on key (f8)  let hotkey = 108 goto event
   on key (f9)  let hotkey = 109 goto event
   . . .
   ```

   The function `hot_key()` examines the `hotkey` global variable and retrieves its respective event name out of the `key_map` array (remember the `key_map` array is filled with values from the Hot Key Definitions Detail table (`stx-hotkd`)). Then `hot_action()` is called.

4. `hot_action()`: Its job is simply to set the global variable `scr_funct`. It sets `scr_funct` based on answers to the following questions, in the following sequence:

- Is it in the local event array with a hot key mapped to it? The local event array includes the events passed as arguments to `hot_local()`.

  If YES, `scr_funct` is set to the event name specified in the Hot Key Definition Detail table (`stxhotkd`) (via the array `key_map`). The event is then processed locally in the local event function `llh_event()`, `lld_event()`, or `EV_scrid`.

  If NO,

- Is the event disabled?

  If YES, don’t process it. If NO,

- Is it a global event?

  If YES, process it as a global event using the `global_events()` function in `main.4gl`.

  If NO,

- Is it an Enhancement Toolkit function?

  Here are the event names of the User Control features. These are all processed with separate functions:

- navigate (Navigation)
- help (User-updatable Help Text)
- hot (User-definable Hot Keys)
- info (Program Information)
- notes (Freeform Notes)
- ack (Software Acknowledgement System)
- status (Program Status Monitor)
- feature (Feature Request System)
- bang (Operating System Exit)
- errbrws (User-updatable Error Text)

- todo (Personal To-Do List)
- funct_edit (Edit Current 4GL Function)

  If NO,

- Is the os_command column in the Navigation Event Reference table (stxactnr) for this event null?

  An SQL query is done on this table for this event. If YES (the event is null), scr_funct is set to the event name in the column act_key of the Navigation Event Reference table. The event is processed locally (in llh_event, lld_event(), or EV_scrid).

  If NO, the event is an external event. The event is processed externally using the INFORMIX-4GL run statement.

Coding Local Events

Event handling logic is not performed by standard INFORMIX-4GL branching logic (if-then, case statements, etc.). Fitrix *Screen* generates special labels in the code to handle events. There are places in the code that are labeled "events" and code branches to them via a "goto" statement, for example:

```
# Local event processing
label event:           # this event is labelled "event"

after input            # this event is labelled "end_input"
    label end_input:
```

Lowlevel functions called llh_event()/lld_event()/EV_scrid take care of event handling logic.

To create a local event, you need to create an on_event trigger. The on_event trigger allows you to place custom INFORMIX-4GL logic for an internal event into your program, for example:

```
on_event show_message
         display "Logic from my internal event is executing now."
         sleep 3;
```

For more information on the on_event trigger refer to "on_event" on page 12-14.

Before coding a local event trigger, the event should be added to the Fitrix *Screen* tables with the navigation feature or by using ISQL.

To incorporate the local event into the program, some lines of source code should be added. These lines of code can be added automatically by specifying a trigger in a triggers file (see "on_event" on page 12-14):

- Call `hot_local("event")`, where event is the value in the `act_key` column of the two tables, if the event was added via ISQL or 4GL. This function should be called just before the call to function `hot_key()`. A call to `hot_local()` tells `hot_key()` that there is custom logic coded just for this event and perform this custom logic instead of the logic that is usually performed. This custom logic is added to the function `llh_event()/lld_event()/EV_scrid`, and this function is called right after calls to `hot_local()` and `hot_key()`. Any usual logic can be overridden by a call to `hot_local()`, even logic for an event that occurs in the upper level functions.
- The call to `hot_key()` simply maps the key pressed to the function being called, performs the usual logic if a call to `hot_local()` is *not* specified, sets `scr_funct`, and returns. The `scr_funct` variable is set to the event name, or `act_key` column value.
- Custom logic for handling the local event is coded in the function `llh_event()/lld_event()/EV_scrid`. An additional "when" clause of the CASE statement is added.

Example of source code added to handle a local event:

In `llh_input()`:

```
        # Local event processing
        label event:
a->          call hot_local("date_zm")  # Look to llh_event for custom logic
            call hot_key("llh_input")  # Map the key to the event
            call llh_event()
```

In `llh_event()`:

```
    case
        when scr_funct = "zoom" and infield(customer_num)
            if zoom("cust_zm","")
            then
                    let p_orders.customer_num = scratch
                    let nxt_fld = "customer_num"
```

```
              end if
     a->   when scr_funct = "date_zm" and infield(order_date)
     a->       if zoom("date_zm","")
     a->       then
     a->           let p_orders.order_date = scratch
     a->           let nxt_fld = "order_date"
     a->       end if
          when scr_funct = "accept"
              let nxt_fld = "exit input"
          when scr_funct = "tab" or scr_funct = "btab"
              let tab_pressed = true
              let nxt_fld = "exit input"
          when scr_funct = "cancel"
              let int_flag = true
              let nxt_fld = "exit input"
       end case

 ** a-> = lines that were added
```

When running the application, you can choose the event from the navigation menu. This event only runs at the point specified in the running program. The event does not work anywhere else. For instance, the above lines of code are executed only during inputting on the header. If this event is chosen anywhere else besides during inputting on the header, it does nothing.

Coding Global Events

Global events can occur at any point in a running application.

Global events are located in a function called `global_events()` at the end of `main.4gl`.

There are two ways to add a global event:

1. Add the event via the Navigation Event feature. Then go into ISQL and null out the `nav_user` column. This allows the event to be accessible by all users.

2. The second way is to add rows to the two navigation tables as specified in adding local event code above. This can be done through ISQL or 4GL.

Either way, in `main.4gl`, you need to add to the `global_events()` function an additional "when" clause to the case statement. If there is no `global_events()` function, copy the example below or use the `on_event` trigger in the `defaults` section of a triggers file (explained below) to create one automatically.

Example:

```
######################################################################
function global_events(act_key, p_funct)
# returning true if it runs the event, otherwise false
######################################################################
# This function’s job is to run all events that need to be run
# on a global (program wide) basis.  If you have defined an event
# that needs to be run at the menu level in addition to the local
# input level, the event must be listed here.
# If you wish to know the function name that called hot_key, it
# is passed as p_funct.
#
    define
        act_key char(15),       # Action to process
        p_funct char(15)        # Current function name

    # Trap fatal errors
    whenever error call error_handler

    # Process the events based on act_key
    case
      # There must be at least 1 statement listed.
      when act_key = "info" call fg_info()
      otherwise return false
    end case
    return true
 function
# global_events()
```

The Event Tables

The following describes the four tables you need to become familiar with in order to understand event handling and navigation.

1. Navigation Event Reference Table - `stxactnr`

   This table contains all events that are displayed on the navigation menu. This table is the "header" table for navigation events. When the user adds a navigation event while running the program, a row is added to this table for that event. The event name is stored in the `act_key` column.

   There may be events in the program that do not appear in this table.

Here is a sample row from this table:

column	**data**
language	ENG
act_key	shipto
description	update shipping addresses
os_command	
press_enter	N

The "shipto" event in the previous example can be identified as a local event because the `os_command` column is blank.

The following is an example of an external event. Event `cust_info` might have a row in `stxactnr` that contains the following information. External events have something in the `os_command` column.

column	**data**
language	ENG
act_key	cust_info
description	Customer Information
os_command	cd ../../ar.4gm/i_custr.4gs; fglgo *i -c $company
press_enter	Y

2. Navigation Event Detail Table - `stxnvgtd`

This table is the "detail" table for navigation events. It holds the program, module, and user for the event. The `line_no` column may be manipulated to alter the order that items appear on the navigation menu. The `nav_user` column may be filled in with a user to allow *only* that user to execute that event.

The contents of the table `stxnvgtd` determine the scope of the event. Remember that the definition of an event as global or user global does not automatically make it available to an individual program. The logic necessary to invoke the event must still be added to each program that uses the event.

An event can be:

- **user/program specific**

  A specific user can use this event in a specific program.

- **user global**

  A specific user can use this event in all programs.

- **program specific**

  This event can only be used with a specific program.

- **global**

  This event can be used with all programs and by all users.

The following table shows example entries in the `stxnvgtd` table:

Type of event	act_key	line_no	nav_user	nav_program	nav_module
user/program specific	`cust_info`	`null`	`gordona`	`screen1`	`demo`
user global	`cust_info`	`null`	`gordona`	`null`	`null`
program specific	`cust_info`	`null`	`null`	`screen1`	`demo`
global	`cust_info`	`null`	`null`	`null`	`null`

If `line_no` is blank for an event, this event shows up *above* any events that are numbered.

3. Hot Key Definitions Reference Table - `stxkeysr`

   This table assigns a unique number to a key on the keyboard. It assigns numbers for control keys, function keys, the tab key, and delete key. The number assigned to control keys corresponds with the numeric sequence of the alphabet ([CTRL]-[a] is 1, [CTRL]-[b] is 2, [CTRL]-[c] is 3, etc.). Function keys start with the number 100. Here is a sample from this table:

key_code	key_desc
2	[CTRL]-[b]
5	[CTRL]-[e]
6	[CTRL]-[f]
7	[CTRL]-[g]
9	[TAB]
14	[CTRL]-[n]
15	[CTRL]-[o]
16	[CTRL]-[p]
20	[CTRL]-[t]
21	[CTRL]-[u]
22	[CTRL]-[v]
23	[CTRL]-[w]
25	[CTRL]-[y]
26	[CTRL]-[z]
101	[F1]
102	[F2]
103	[F3]
104	[F4]
105	[F5]
106	[F6]
107	[F7]
108	[F8]
109	[F9]
110	[F10]
111	[F11]
112	[F12]

4. Hot Key Definitions Detail Table - `stxhotkd`

This table stores each event and its respective hot key number for the hot key that initiates that event. Re-assigning hot keys merely changes this number. The module, program, and user is also stored so there can be duplicate numbers. Here is a sample from `stxhotkd` table:

hot_key	act_key	hot_module	hot_program	hot_user
2	btab			
5	hot			
6	udf			
7	navigate			
9	tab			
14	notes			
15	bang			
16	null			
20	todo			
21	null			
21	payto	a	p_vendr	
21	add_info	ar	i_invce	
22	null			
22	acctg_info	ar	i_invce	
23	help			
25	info			
26	zoom			
101	insert			
102	delete			
103	page_down			
104	page_up			
105	mail			
106	payto	ap	i_vendr	
106	add_info	ar	i_invce	
107	acctg_info	ar	i_invce	
134	btab			
135	accept			
136	cancel			

There are two ways to add an event to Fitrix *Screen* tables: via the navigate feature or ISQL.

1. **The navigate feature:** When running the application, you can add a local event by executing the Navigate feature of the User Control Library. For more information on using the navigate feature refer to the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

2. **ISQL:** ISQL or 4GL can be used to add rows to the two navigate tables, `stxactnr` and `stxnvgtd`.

The `act_key` column in `stxactnr` uniquely identifies the event in `stxactnr` and `stxnvgtd`.

Creating an Event that Calls a Program

The best way to create a navigation event that calls another program is to use the Fitrix *Menus* `mz -i` command. If you use the `mz -i` command you do not have to cd to the program directory and then run it, and you do not have to determine if the program is a .4gi or a .4ge. The `mz -i` command does that automatically. Another benefit you get when using the `mz -i` command is that `$cust_key` and other environmental variables are maintained.

The `mz -i` command can only be used if you have Fitrix *Menus* and the program you want to run appears on a menu somewhere. The reason the program has to appear on a menu is because the `mz -i` command utilizes a Menu Item Instruction file.

For example, you could set up a navigation event to run the Error Message Translation program, `$fg/codegen/utility.4gm/i_terorh.4gs/i_terorh.4gi`. The following line would appear on the Operating system command line on the Navigation Commands form.

```
Operating system command:
mz=$fg/codegen/utility.4gm;export mz;mz -i a menus transmenu
```

Note that when you use the `mz -i` command, you must first set the `$mz` variable to point to the directory above the project directory. Then you need to specify the project directory and the menu the item appears on.

For more information on the `mz -i` command and Item Instruction files refer to your Fitrix *Menus* documentation.

Moving Events to Your Customer's System

You must have entries in all three of the following tables in order for an event to be called by a hot key.

`stxactnr` - this table stores the navigation event

`stxnvgtd` - this table stores which programs can use the event

`stxhotkd` - this table stores which keys call the event

Because the `stxactnr` table does not have a way to specify `module.program`, you need to transfer the whole table to your customer's site. If your customer's site has custom events that differ from your development platform, then there is a problem, because you would wipe out all of their custom events.

Currently there is no easy way to handle this. What you need to do is unload your customers's `stxactnr` table, unload your `stxactnr` table, then do a `sort -u` on the `act_key`. You would redirect this to an output file and then would end up with an unload file that contained all events.

Record-level Validation

Record-level validation logic is considered event logic. The very last activity to take place in an input statement is after the "after input" statement is executed. This is an appropriate place to put record-level validation logic. When the user presses [ESC], control passes to the event `end_input`, which is part of the after input statement. Within `end_input` there is a call to `llh_a_input()/lld_a_input()`. Within this function, validation logic can be inserted to be performed on all input fields (as a whole). After this, control exits input, the variable `scr_funct` is set to null and the input function terminates.

Control can return back into the input statement in `llh_a_input()/lld_a_input()` by manipulating the value of the variable `nxt_fld`. You can set `nxt_fld` to a field within the input statement to move control to that field.

How to Assign Default Hot Key Settings

When a user assigns a hot key definition, the system automatically inserts the user id into stxhotkd. This restricts that definition to be used by that user alone.

In order to set up a global hot key that can be used by all users of a particular program, a null value needs to be placed into the hot_user column. Modifying the hot key definition in this way should be done via ISQL, since the system automatically assigns the user id.

To map a global hot key go into ISQL and add your definition to the stxhotkd table with a null value in the hot_user column. The stxhotkd table contains the following columns:

```
hot_key
act_key
hot_module
hot_program
hot_user
```

By nulling out the hot_module, hot_program, and hot_user columns, you can define a hot key that can be used by any user within any application.

To map a default hot key:

1. **Create a file containing your hot key definition.**

   Example 1:

   ```
   107|summ|oe|i_invce||
   ```

   This example allows all users of the Order Entry `i_invoice` program to access this hot key definition.

   Example 2:

   ```
   107|summ||||
   ```

   Example 2 allows any users running any application to use this hot key definition.

2. **Run ISQL and load the contents of your unload file into `stxhotkd`.**

   Example:

   ```
   load from "filename" insert into stxhotkd
   ```

Creating and Using Custom Libraries

Creating a library to store generic functions rather than maintaining them at the local level simplifies and streamlines the source code.

There is no functional need to segregate functions—the same result can be achieved independent of the organization of the code. Yet the benefits gained by establishing and building a custom library are clear: reduction in the total number of lines of code within an application, greater modularity of source code, and facility of subsequent modification.

When functions shared by more than one module are moved to a custom application library, the number of lines of source code specific to that program is reduced.

Modularity is a desirable goal in source code development. Much of the power and ease of modification found in applications generated with the Code Generator is rooted in modular organization. Resulting source code is more compact and more understandable to those other than the author. When modifying library source, it is easier to isolate the effects of code changes—you can more easily trace program flow. Applications can be "grown" in a more orderly, concise manner. Rather than "reinventing the wheel," you can take advantage of existing functions to fulfill similar objectives within an application.

Global modifications are made by changing the code in one place, rather than in all the local places the code appears.

Creating a Custom Library

Custom libraries created at the module level are found in a directory named `../mylib.4gs` (relative to the .4gs source code directory).

To create a custom library, you must follow these steps:

1. Create the `mylib.4gs` directory at the module level.
2. Place the library function(s) into `mylib.4gs`.
3. Copy a `Makefile` into `mylib.4gs`. The `Makefile` can be created from scratch, but it is far easier to copy an existing `Makefile`. The best type of `Makefile` to use for this purpose is found in `$fg/lib/scr.4gs`.

   For more information on compiling libraries refer to "Linking in Libraries with $cust_path" on page 14-17.
4. Modify the `Makefile` copied into the `../mylib.4gs` directory. Edit the `LIBFILES` section to include your function filenames. Make sure the last `$(LIB)` entry does not end in a backslash (\). Next, change the `LIB=../scr.a` line to the following:

   ```
   LIB=../mylib.a
   ```
5. After the `Makefile` file has been modified, run `fg.make` in the custom library directory to compile your function(s).
6. Run `fg.make` on your local program to link your library functions into your program.

After completing these simple steps, you can use those functions in your local program directories as if they resided there. The `fg.make` linker finds them and makes them a part of the executable program.

The `libraries` trigger lets you append your custom directory after the `../mylib.a` line but before the rest of the Fitrix *Screen* libraries. For more information on this trigger refer to "libraries" on page 12-18.

The `custom_library` trigger allows you to place any custom library before the `../mylib.a` library. For more information on this trigger refer to "custom_libraries" on page 12-19.

The hypertext functionality provided for all applications generated by the Code Generator also extends to custom libraries. For more information on hypertext functionality, see the section "Locating Source Code."

The `libraries` Trigger

If you have built a library of custom functions and wish to use this custom library with the current application, you can do so by specifying the `libraries` trigger in a triggers file. Any libraries specified here are automatically placed into the `LIBFILES` section of the local `Makefile`. Example:

Specifying the libraries trigger like so:

```
defaults
   . . .
       libraries
            $(fg)/lib/mylib.a
       ;
   . . .
input 1
   . . .
   . . .
input 2
   . . .
   . . .
```

results in the local Makefile looking like this:

```
#  Makefile for an Informix-4GL program
NAME =      . . .

OBJFILES =  . . .
```

```
FORMS =     . . .

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

GLOBAL =    . . .
      . . .
```

Note that any library specified through a triggers file places that library before the Fitrix *Screen* libraries, so that if there are any functions that have the same name across libraries, the functions found in the earlier LIBFILES entry are executed.

The `libraries` trigger must always be placed in the `defaults` section of the triggers file. The `libraries` trigger is the only trigger that does not affect a .4gl source code file.

Customizing Library Functions

A highly useful example of customizing library functions in local code is disabling ring menu options (Add, Update, Delete, etc.). This is done by using library functions called `ok_` functions. These `ok_` functions are:

- `ok_add`
- `ok_update`
- `ok_delete`
- `ok_find`
- `ok_browse`
- `ok_next`
- `ok_prev`
- `ok_tab`
- `ok_options`

- `ok_exit`
- `ok_bang` (for shelling out)

Before a ring menu option is executed, there is a call to its respective `ok_` function. If there are no restrictions on performing that ring menu option, the library `ok_` function is called and it returns a value of true, which means it is OK to perform that ring menu option.

However, if you wish to place restrictions upon a ring menu option (for example, disabling the Delete option), the `ok_` function handling that ring menu option may be copied out of the library into local code and modified. A ring menu option may be completely disabled if this local `ok_` function returns a value of false. During compilation, the linker finds the local `ok_` function first and uses it instead of the library `ok_` function.

Here is an example of an `ok_` function that has been modified. When the user is trying to delete a customer with the Delete ring menu option, a warning comes up and then the user is asked to verify deletion of the record:

```
##################################################################
function ok_delete()
# returning true or false based upon ok to delete
##################################################################
#
#
    define
        prompt_response char(1)


    open window delete_rec at 14,16 with 9 rows, 51 columns
        attribute (border, blue, prompt line last)
    display "                  W A R N I N G !                  " at 1,1
        attribute (reverse)
    display "===================================================" at 2,1
    display " Deletion of this record will also delete all      " at 3,1
    display " orders and their items associated with this       " at 4,1
    display " customer.                                         " at 5,1
    display "===================================================" at 8,1


    let prompt_response = null
    while prompt_response matches "[^yYnN]" or
          prompt_response is null
        prompt " Continue? " for prompt_response
    end while
    if upshift(prompt_response) = "N"
    then
```

```
          close window delete_rec
          return false
        end if

        close window delete_rec
        return true


    end function
    # ok_delete()
```

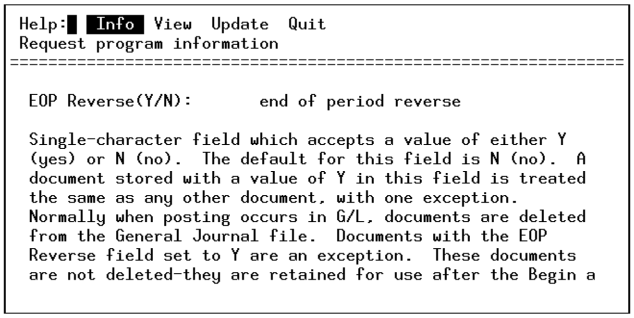
Notice how the user’s response to verification is interpreted. If the response is "N", a "false" value is returned and the delete is *not* performed.

The `ok_` functions are classified as midlevel functions.

Creating Application Help

Fitrix *Screen* generated applications have multiple levels of context sensitive online help. You can define unique help at virtually any point in your application, whether it be at the menu level, the ring menu level, or the field level. This section focuses on creating ring menu and field level help.

When an application is run, help can be displayed by pressing [CTRL]-[w]. This displays a screen containing help text relating to your current position. The following is an example help screen:



```
Help:  Info  View  Update  Quit
Request program information
================================================================

 EOP Reverse(Y/N):       end of period reverse

 Single-character field which accepts a value of either Y
 (yes) or N (no).  The default for this field is N (no).  A
 document stored with a value of Y in this field is treated
 the same as any other document, with one exception.
 Normally when posting occurs in G/L, documents are deleted
 from the General Journal file.  Documents with the EOP
 Reverse field set to Y are an exception.  These documents
 are not deleted-they are retained for use after the Begin a
```

The commands on the command line of the Help form are explained below:

Info: This option leads to the Program Information Menu, which contains a list of five selections. The Program Information Menu and its selections are documented in the "Program Information Menu" on page 3-19.

View: This option scrolls through the text displayed on the Help form. The INFORMIX-defined cursor movement keys (arrow keys, [F3], [F4]) are available while viewing the text.

Update: This option is selected to enter or modify help text. To store text entered on the form, use the [ESC] key.

Quit: This option exits the Help form and returns the cursor to the original position prior to entering the Help form.

The Fitrix *Screen* Online Help System

This section describes how the help system works. The text that is displayed on help screens is stored in the database. The logic that links the data in the database to the particular blocks of text differs slightly depending on where help is called from.

The `stxhelpd` table contains all of the help information for the application. This table contains both the actual help text itself, but also special keys that link the strings of help text to various locations in the program. The `stxhelpd` table contains the following columns:

```
stxhelpd:
    language char(3),
    userdef char(1),
    hlp_module char(18),
    hlp_program char(18),
    hlp_number smallint,
    line_no smallint,
    hlp_text char(60)
```

The `hlp_module, hlp_program, hlp_number,` and `line_no` columns make up the unique key for the text. The data in these columns link the text to the correct locations in the program.

The following describes the four ways help text is linked to various parts of the program.

1. Ring menu help

   If you're at the ring menu, help text is keyed from the module and progid (this was the initial intent of the help system - hence the `hlp_module` and `hlp_program` column names in `stxhelpd`):

   At the main ring menu:

	hlp_module	**hlp_program**	**hlp_number**
logic	program_module	program_name	1
example	all	i_alias	1

   This way, you can have help text that relates to the main screen in the program.

   The whole idea behind `hlp_number` was that if a programmer wanted to shift contexts within the screen (say, a different help in the options menu, or in find mode, or something), they could programmatically call help with another number, and still keep the context to the module and program. The `hlp_number` is usually 1.

2. Special screen types

   Since the advent of add-on screens, you can have many screens in one application. When add-on's were conceived, a decision was made to tie screen level help to the `scr_id` for these types of screens.

	hlp_module	**hlp_program**	**hlp_number**
logic	program_module	scr_id	1
example	all	acct_zm	1

   A screen is identified as an add-on type if the `scr_id` is not default. If the `scr_id` is not default, it's of the add-on family (including extensions and zooms), so the `scr_id` is used for the `hlp_program` instead of the `program_name`.

   The zoom screen is the only type of additional screen that has a ring menu. So for all practical purposes, zoom screens are the only additional screens that have screen level help.

3. Field level help

   The most common type of help is field level help. Field level help is when you are updating a form, and you press [CTRL]-[w] on a field. In this scenario, your help is keyed as follows:

	hlp_module	**hlp_program**	**hlp_number**
logic	table_name	column_name	1
example	stxalisr	gl_alias	1

4. Formonly field help

   The fourth level is a little bit more abstract. This is where a screen field isn't attached to a database column. These screen fields are of formonly type.

   If you press [CTRL]-[w] on a formonly type screen field, help text is keyed to the screen, not the database (for obvious reasons). Here's the logic:

	hlp_module	**hlp_program**	**hlp_number**
logic	moldule.program	field_name	input_area
example	all.i_cashe	gl_p1	2

Notice the `hlp_number` is keyed to the input area number (in this case, the detail section - area 2). This is because you can have the same field_name in both input areas, meaning two different things, so you have to uniquely key them.

Like the add-on example above, if the screen isn't the main screen in the application, the `hlp_module` is keyed to the screen vs. the program:

	hlp_module	**hlp_program**	**hlp_number**
logic	moldule.scr_id	field_name	input_area
example	all.inc_exc	gl_p1	2

Note

There is a way to create field-level help that is unique to a program. Refer to "Creating Field Level Help That is Unique to the Program" on page 18-2.

How to Create Help Text for Your Applications

There are several ways to create help text for your applications. One way is to use the Table Information form from the Database Administration option that is available in the Form Painter. This program allows you do define help and attach it to your database columns. Another way is to create unload files with your own editor such as vi and then load them into your database. Defining application help through the form painter is discussed first.

You can also create help text by copying pre-defined help from another module or program. For more information on copying help text refer to the Fitrix CASE *Tools Enhancement Toolkit Technical Reference*.

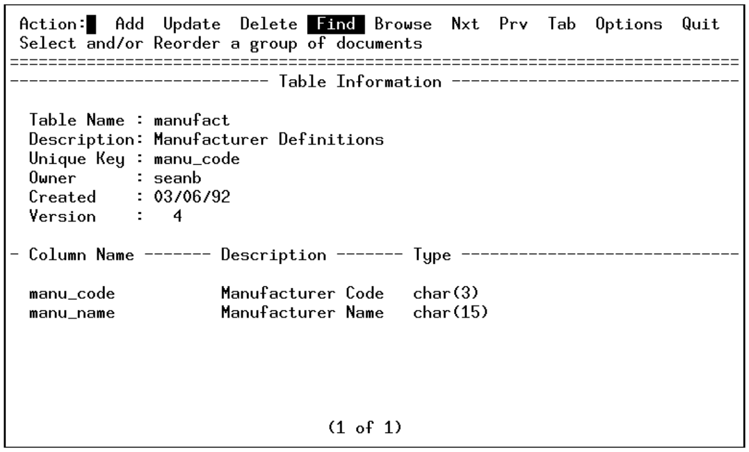
Defining Application Help Through the Form Painter

The ability to define help for an application is part of the form painting process as a matter of convenience; application help text can be modified at any time, even after the 4GL code has been compiled.

Field level help text can be defined at the column level through the Database Administration feature. Once help text has been defined for a particular column, any field on any data-entry form that references that column accesses the help text defined for the column.

The first step in defining application help is to select the Database option from the Form pull-down menu. Before you can define help text for columns, you must make a table current on the Table Information form. Use the Find command on the ring menu to help make a table current.

The following example shows the `manufact` table from the `stores` sample database:

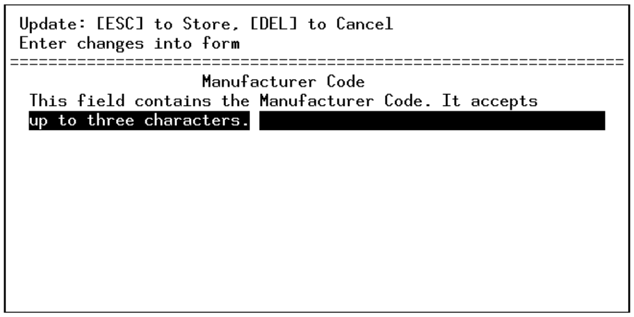


```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Select and/or Reorder a group of documents
==========================================================================
-------------------------- Table Information -----------------------------

  Table Name : manufact
  Description: Manufacturer Definitions
  Unique Key : manu_code
  Owner      : seanb
  Created    : 03/06/92
  Version    :   4

- Column Name ------- Description ------- Type ----------------------------

  manu_code           Manufacturer Code   char(3)
  manu_name           Manufacturer Name   char(15)




                               (1 of 1)
```

In the example above, help text defined for columns in the `manufact` table can be accessed later from application data-entry form fields referencing columns in the `manufact` table.

To define help text for columns you must update the record for the table containing the columns. Records shown on the Table Information form contain two sections: header and detail. The detail section, on the lower portion of the form, contains information about columns comprising the table. With the cursor on a row in the detail section, press [CTRL]-[n] to create, view, or modify help text for the particular column listed on that row.

Pressing [CTRL]-[n] while on a column in the detail section of the Table Information form displays the following form:



```
Update: [ESC] to Store, [DEL] to Cancel
Enter changes into form
===============================================================
                   Manufacturer Code
 This field contains the Manufacturer Code. It accepts
 up to three characters.
```

The Help form shown here is the form used to define help text for database columns and, thereby, for applications. Enter help text for data-entry fields that reference this database column.

Creating Help Text Through Unload files.

The other way to create help text for your programs consists of creating unload files. Unload files are ASCII files that contain database information. The information in an unload file is loaded into a database using SQL. Unload files are used to add information to existing databases.

Online help text exists in an unload file called `stxhelpd`. This file contains all of your original help text and is loaded into your customers database. The `stxhelpd` unload files do not contain any user defined help. A typical `stxhelpd` file looks like this:

```
ENG||colm_def|coldesc|1|1|The Column Name field contains a descriptive name
for the|
ENG||colm_def|coldesc|1|2|database column.  You may enter a name that
better|
ENG||colm_def|coldesc|1|3|describes the database column.|
ENG||colm_def|collabl|1|1|This field contains the name of the column that
will appear|
ENG||colm_def|collabl|1|2|on the report.  Enter the column name the way you
want it|
ENG||colm_def|collabl|1|3|to appear on the report.|
ENG||dgrp_def|data_desc|1|1|This field contains the name of the Data Group.
Data|
ENG||dgrp_def|data_desc|1|2|Groups allow you to find the main table for the
report and|
ENG||dgrp_def|data_desc|1|3|restricts the SQL relationships to a manageable
set.  By|
ENG||dgrp_def|data_desc|1|4|choosing a Data Group, all related tables are
automatically|
ENG||dgrp_def|data_desc|1|5|retrieved.  Enter a descriptive name for the
Data Group.|
```

The first six columns in this unload file make up the "key" to the help text. This key is what matches the help text to a particular point in a program. The first word, ENG, represents the language tag. This tag allows you to display help text in other languages.

Next you see a "|" (pipe). The pipe is a field separator in an unload file. Pipes separate different columns in the database. In this example, you see two pipes next to each other indicating that the second "field" is null.

This second slot is reserved for "User Defined." If you were to unload your `stxhelpd` table from your database and a user had modified some field description, a Y would appear here indicating that help is User Defined. When help is marked as User Defined, future upgrades of the software do not overwrite the help definition.

The third column contains the database table name identifying the particular field. Each field on a form is identified by a table and a column.

The fourth column contains the column identifying the field.

The fifth column usually contains the number 1.

The sixth column contains the line number of the help text. Each line of help text for a field is numbered sequentially.

Creating BLOBs

The Code Generator and Form Painter both can utilize `byte` and `text` OnLine engine data types. However, these data types can only be used when running the OnLine engine.

If you are using the OnLine engine, you can create an application that uses BLOB (Binary Large OBject) technology. A BLOB is a text file, graphics file, sound file, or another application. The Code Generator provides most of the work for BLOB functions, but it is up to the programmer to specify the program needed to invoke the BLOB, whether it calls a read file, displays a graphic, plays a sound, or runs another application.

For example, you may want to create an application that accesses a Wingz spreadsheet. What you do is create a field in your application that is defined in the database as type `byte`, choose the method for invoking the spreadsheet, and finally determine if the spreadsheet can be modified.

When the generated application is run, an asterisk appears in the BLOB field when data is available for that field. To display the data in the field, or in this case to run the program to invoke Wingz, enter the field and press [CTRL]-[z]. The Wingz spreadsheet appears on the screen. To return to the original application, perform a regular exit for the BLOB application.

To use a BLOB, create a field with text or byte as the field type. You must then enter the program and edit permission. Program and edit permission is entered using the Form Painter. For more information on creating BLOBs with the Form Painter refer to "Creating BLOB Fields" on page 7-18.

The following example shows the format required in the FGSS section of the .per form.

```
blobdef = blobbyte, Wingz, Y
```

`blobdef`	is the keyword.
`blobbyte`	identifies the BLOB as a byte type.
`Wingz`	specifies the name of the program to call.
`Y`	is the edit permission flag.

A typical program for a text BLOB might be vi or your standard editor. A typical program for a byte BLOB might be xloadimage.

NOTES:

1. Field types of byte and text are only supported in the input 1(header) region of header and header/detail forms.
2. BLOBs cannot be part of a detail table.
3. Full maintenance of byte and text BLOB fields is only generated for BLOB fields which are in the main table for the form.
4. Full maintenance of byte and text BLOB fields is not provided for formonly BLOB fields. You must provide additional code for maintenance of these formonly fields.
5. You cannot have a table with only a serial field and a BLOB field. Another field type such as char or integer must be present when using a serial field and BLOB field together in a table.

The Code Generator creates a temporary O/S file that contains the BLOB, and passes the name of this file as the first argument to the program. When the BLOB is exited, the temporary file is automatically removed.

If a BLOB is edited, it is updated in the table when [ESC] is pressed. If the [DEL] key is pressed, any changes to the BLOB are not recorded.

Custom 4GL Functions and BLOBs

If you want to run a custom 4GL function rather than a UNIX program you need to specify the name of the function and any arguments in parentheses. The following is an example:

```
my_funct("blobfield", 22, "Y")
```

In the above example, when [CTRL]-[z] is pressed in the field containing the BLOB, `my_funct()` is called with the three parameters: "blobfield", 22, "Y". Parameters are optional, but you must provide the parentheses to inform the Code Generator that this is a custom 4GL function and not a program name. You must also provide the function in the `at_eof` section of a trigger file.

Your custom 4GL function communicates with the generated code via the `scratch` variable. Set `scratch` to one of the following before returning from your function:

1. **null:** A null `scratch` specifies that no edit was performed. Your 4GL function must take care of the removal of the temporary file (if any).
2. "**(delete)**"**:** If `scratch` contains "(delete)", then that specifies a delete request to the generated code. The BLOB is deleted from the current row.
3. **O/S filename:** If anything else is specified in `scratch`, it is assumed to be a UNIX filename where the BLOB resides.

An example 4GL function that handles BLOBs is found in `$fg/lib/standard.4gs/run_blob.4gl` and is named `run_blob()`.

Sample BLOB Application

The following example explains how to add a byte BLOB field to an application that calls up an Informix Wingz spreadsheet. This example shows what is contained in the .per specification file. For more information on creating BLOBs with the Form Painter refer to "Creating BLOB Fields" on page 7-18.

1. Define a byte field in the input 1(header) region of your .per form.

   For example, specify something like the following in the SCREEN section of the .per:

```
    byte column    :[A]
```

Then specify the following in the ATTRIBUTES section:

```
    A  = <table>.blobbyte, comments = "Wingz field";
```

2. Add the following line to the FGSS section of the .per form:

```
    blobdef = blobbyte, Wingz, Y
```

The Code Generator creates the necessary code to maintain the byte field and invoke Informix Wingz with [CTRL]-[z] when the user is in the byte field.

In the current implementation of the Code Generator, Informix Wingz is invoked with a temporary file name that does not have a ".wks" extension. Because of this, the first time Wingz is run with a file the "Save as" option must be used to remove the ".wks" extension that Wingz attempts to append to the filename with which it was invoked.

Example of a .per form using byte and text fields:

```
{
######################################################################
# Copyright (C) 1991
# Fitrix, Atlanta, Georgia
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  %Z%  %M%  %I%  Delta: %G%
#######################################################################

# Screen Generator version: 4.11.UC1
}
DATABASE gordona

SCREEN
{

-------------------------- blob test table ---------------------------

non-blob column:[A1                            ]
byte column    :[A]
text column    :[B]
}

TABLES
gordo

ATTRIBUTES
A1 = gordo.nonblob, comments = "Non-blob test field";
A  = gordo.blobbyte, comments = "Wingz field";
```

```
B  = gordo.blobtext, comments = "Vi text field";

INSTRUCTIONS
screen record s_daren (gordo.nonblob, gordo.blobbyte, gordo.blobtext)

delimiters "  "
{
######################################################################
FGSS
######################################################################

defaults
module     = demo
type       = header
init       = 1=0
attributes = border, white
location   = 2, 3

input 1
table   = gordo
filter  = 1=1
blobdef = blobbyte, Wingz, Y
blobdef = blobtext, vi, Y
}
```

Creating Skip Field Logic

The Skip Function allows you to skip a specified field on your form during data entry when a specified condition is met. In other words, you can use conditional logic in conjunction with the SK_ skip function to cause an enterable field to be skipped during data entry.   For example, say you have written skip logic for the Address field on your order form that instructs the program to skip the Address field if the Customer Name field is null. If you add a new order and leave the Customer Name field blank, the program skips over the Address field and places you in the next enterable field.

To create skip logic for a field:

1. Identify the fields that use skip field logic.

You can mark a field as a skip field on the Define Fields form in the Form Painter. Marking skip fields in the Form Painter creates the following skip instruction in the .per file:

```
skip = col1, col2, col3, col4
```

Fields requiring skip logic need to be specified in the .per file to cut down on the size of the generated function. By specifying which fields use skip logic, code is generated only for those fields, rather than for every field.

2. Write the skip logic for each field and put it in a .trg file.

To call the skip function you must specify the condition that causes the field to be skipped and place the following code into your .trg file:

```
before_field {column name}
if {condition}
then
    call llh_skip(prv_fld)
end if
```

The location of the field determines the exact syntax of your skip statement. For example, if you need to skip a field in the input 2 section of your form, then use the following:

```
before_field {column name}
if {condition}
then
    call lld_skip(prv_fld)
end if
```

If you want to call the skip function from a form type other than header or header/detail forms, then use:

```
call SK_scr_id(prv_fld)
```

Cursor Handling Philosophy

Code generated with Fitrix *Screen* uses cursors as temporary tables to assist with the manipulation of data within the data-entry application.

The cursor philosophy has many advantages:

- You can use database transactions and the OnLine engine. (The library code handles the begin/commit/rollback work statements.)
- Cursors can be programmed easily—all of the work is done in the user interface libraries.
- You can use the `ring_sort()` routine from the options menu to have the user define the sort criteria for the browser.
- You can watch the computer gather the documents (in increments of 100).
- You can press [DEL] during the document gathering process without canceling the cursor. The cursor retains the documents gathered before [DEL].
- You can pass a filter clause on the command line.
- You can pass an order clause on the command line.

Creating a Generic Text Picker/Editor

A text picker is a list box that displays a list of items to the user. Once the user selects the data from the picker, the data is returned into `scratch`, and then `scratch` can be set to the field into which the data is returned. Before a picker is run, picker items must be put into a system-maintained array. Picker items can be hard coded into the array with the `textput()` function:

```
call textput("Harpo")
call textput("Groucho")
call textput("Chico")
call textput("Zeppo")
call textput("Karl")
```

`textput()` can also accept the argument of "(see scratch)" to pull a picker item out of scratch . The `textpick()` function is called to pull up the picker with the values loaded. The argument of `textpick()` is the heading of the picker. For example:

```
if textpick("Pick a Marx Brother") > 0
        then let p_orders.marx_brother = scratch
end if
```

`textpick()` automatically determines what window size is needed based on the widest picker item or header. The picker scrolls if there are more than 6 items. `textpick` also returns the number of items it found, so in the example, if there are no items that come up the picker does not come up.

`textsel()` is like `textput` except that it expects `scratch` to contain a valid SQL statement, and it loads the picker array with the result of that SQL statement. `textsel` returns the number of rows that have been selected.

Here is an example of how `textsel()` and `textpick()` called before the zip-code field is entered on a customer input screen. `textsel` interprets the city and state entered for the customer and `textpick()` pulls up a picker with zipcodes for that city and state. In the trigger file, simply add the following:

```
input 1
static_define
        quote char(1)
       ;
before_field zipcode
       let quote = " ""
       let scratch =
        "select distinct zipcode from customer where customer.city = ",
               quote, p_stomer.city, quote, "and customer.state = ",
               quote, p_stomer.state, quote, "order by zipcode"
               if textsel() > 0 then
                    if textpick("Select a Zipcode") then
                       let p_stomer.zipcode = scratch
                    end if
            end if
       ;
```

The Text Editor: The text editor is a generic set of routines that display data in a window where it can be edited and then retrieved.

The size of the text editor window is determined when the `textedit()` function is called. Text edit windows can be opened in increments of 20, 30, 40, 50, 60, or 74 characters.

The Text Picker: The text picker is a generic set of routines that display a window and allow a user to scroll through a list of data and retrieve a line.

The text picker determines the size of the window to open based on the longest line of data in the array.

If there is only one item in the menu, that item is picked automatically and the menu is not displayed.

The flow for these windows is as follows:

1. Fill the array with data set (via `textput()` or `textsel()`)

- call `texthelp()` if you want to define help text

- call `textdefault()` if you want default data for the editor (not for picker)

2. Call the editor or picker

3. Return the edited or picked data

The following functions, which can be found in `$fg/lib/standard.4gs/lib_text.4gl`, are included in the picker/editor group:

`textinit()` — call to reset the text array

This function rarely needs called. If no initial data is loaded into the editor, or if you are looping through `textput()`'s and the loop *may* not execute once, then this function is called before any other `text*` calls to initialize the text arrays. This function is called by `textput()` and `textsel()`. If you are using these functions, you do not need to call `textinit()`.

`textput("`*`text`*`")` — place "text" into next available slot

This function puts text into the next available array element. After the picker/editor is called, it cleans out the array for future `textput()` calls. If the value of text = "(see scratch)", then the value to place into the array element is in the scratch variable.

Example:

```
call textput("Apples")
call textput("Oranges")
let scratch = "Peaches"
call textput("(see scratch)")
```

`textsel()` — provide text via an SQL select statement

This function expects the scratch variable to contain a valid SQL statement. It executes that SQL statement and loads the array with the database values. It then returns the number of rows that have been selected. If the number of rows returned from the SQL statement is greater than the number of elements in the array, `textsel()` returns -1. If the SQL statement fails for any reason, `lib_error()` is called, and 0 is returned.

Example:

```
let scratch = "select tabname from systables order by tabname"
if textsel() > 0
then
    if textpick("System Tables") > 0
      then let tabname = scratch
    end if
end if
```

`textdefault("`*`default key`*`")` — define default text for the editor

This function defines the unique key to use to place default text into the editor. The default text is placed into the editor if there are 0 lines of text to be edited upon entry into the editor. If there is a default key, the (Zoom) message appears. If Zoom is pressed during text entry, the default text can be edited.

`texthelp("`*`module`*`","`*`program`*`",`*`number`*`)` — define help text used

This function defines for the picker/editor the module/program/number of help text to use instead of the generic help text.

`texthlp(`*`number`*`)` — define help text used (shortcut - uses `progid`)

Same as `texthelp()`, only it gets module/program from `progid`.

`textedit("`*`heading`*`",`*`width`*`)` — enter the text editor

This function invokes the text editor on the array. It places the heading at the top of the window. The length defines the length of the variables in the array. You may pass any length, though windows are limited to the following lengths: 20, 30, 40, 50, 60, and 74. If the length of the variables exceed the length of the window, then data may be truncated. Upon return, the `scratch` variable contains the data in the array element selected by the user (by pressing [ESC]). If [DEL] is pressed, the function returns false. It returns true if [ESC] is pressed.

`textpick("`*`heading`*`")` — enter the text picker

This function invokes the text picker. It places the heading at the top of the window. The window size is determined by the longest value in the array, or the heading size (whichever is longest). A window is opened displaying the data in the array. If the user presses [ESC] to pick an item, `scratch` is filled with data from that array element, and the function returns true. If [DEL] is pressed, `scratch` is nulled, and the function returns false. If [CTRL]-[z] is pressed, the function returns false (nothing picked) but places the string `zoom` in `scratch[1,4]` and the text of the line the cursor was on (when [CTRL]-[z]) in `scratch[5,80]`.

Example:

```
if textpick("Pick a Fruit") > 0
  then let fruit = scratch
end if
```

This example opens a window with "Pick a Fruit" at the top. It then waits for the user to respond. If the user picks an item, the variable `fruit` contains the selection.

If there is only one item in the menu, that item is picked automatically and the menu is not displayed.

`textget()` — get the next line of the text array

This function is called after `textedit` (or `textpick`) to get the data in the array elements. `textget()` is the opposite of `textput()`. A call to `textget()` returns the "next" item from the picker. The "next" item is always item #1 the first time `textget()` is called for a given picker and is incremented for each call after that. This function is useful when you use the function `textsel()` to load the picker instead of a series of calls to `textput()`. `textsel()` automatically loads the picker for you using the SQL query given as an argument. Later, if you need the results of the query for some reason, it is faster to collect the items from the picker instead of performing the query again. When you call `textget()` the "next" item in the picker is copied into `scratch` and `textget()` returns true. Once there are no more items to return, `textget()` returns false.

Example:

```
let line = 1
while textget()
    let my_array[line].text = scratch
    let line = line + 1
end while
```

This example loads the array `my_array` with the contents of the picker.

`textview()` — view form of textpick

This function behaves in the same manner as the `textpick()` function except for the message displayed on the form. The `textview()` message appears as follows:

```
View:   [ESC] or
[DEL] to Quit
```

`textzoom()` — insert the (Zoom) message

This function tells the text picker to present the (Zoom) message.

A few examples:

```
# Pick from a list
call textput("Apples")
call textput("Oranges")
call textput("Peaches")
call textput("Bananas")
call textput("This is a long fruit name that requires a wider window")
if textpick("Choose a Fruit") > 0
then let fruit = scratch
end if
```

- or -

```
while textpick("Choose a Fruit") = 0
  error "You need to pick a fruit"
end while
let fruit = scratch

# Pick a table from systables
let scratch = "select tabname from systables order by tabname"
if textsel() > 0
then
  if textpick("Database Tables") > 0
  then
      let tabname = scratch
  end if
end if

# Edit a known set of data
let scratch = "select text, line_no from custnotes
 where cust_key = ", cust_num, " order by line_no"
if textsel() < 0
then
  error "Too many lines to edit."
else
  call texthelp("ar","customer",4)  # Define the help text
  call textdefault("table: custnotes") # Define a unique key for default text
```

```
   if textedit("Customer Notes",50) = true
   then
       delete from custnotes where cust_key = cust_num
       let line_no = 1
       while textget()
          insert into custnotes values(cust_num, scratch, line_no)
          let line_no = line_no + 1
       end while
   end if
 end if
```

Error Handling Functions (fg_err and lib_error)

There are two error functions commonly used when coding data validation logic: `fg_err()` and `lib_error()`. Which one is used depends on whether or not you want to duplicate the "common" error you have defined in multiple programs.

Example:

```
call fg_err(3)
```

The `fg_err(#)` (where # is the actual error number) function is used when you want to have a specific call to an error unique to the application you are running. It assumes the module name and the program-id from the variables loaded at run time. Only the error number you want to use is passed to `fg_err()`. `fg_err()` looks to the variables for `module` and `progid`.

The `lib_error("module","program-id",#,"")` function is called to access any error message in the system.

Example:

```
call lib_error("gl","i_genjrn","2","")
```

The extra "" are for storing a technical message that is displayed optionally when errors occur during execution of the program.

The `lib_error()` and `fg_err()` functions are in the Enhancement Toolkit, and since there is no source code available for Enhancement Toolkit, you cannot modify the way they work.

Using a Custom Error Message with Verification Lookups

If you want to incorporate a more specific message when a value is not found, you must "hand-code" the lookup into .ext files. There are two steps to creating a lookup with a custom error message:

1. **Change the second argument of the call to `llh_lookup` to "false" to have it continue without calling the standard error message.**
2. **Create specific logic to handle the "value not found" condition.**

This example uses the library function `fg_err()` to call the error message. This error message number is "1." Here is an example:

```
# After data_changed logic
if data_changed
then
    case
      when scr_fld = "state"
        # Perform Lookups
        #_customer_state_lookup
        if llh_lookup("state_lk",false) = false and     # do nothing
          length(this_data) != 0                        # upon error
        then
            call fg_err(1)      #No State Code for what you entered
            let nxt_fld = "state"
            return
        end if
    end case
end if
```

This modification is made with block commands. First you need to create an .ext file and create a block command to replace the generated lookup with your custom version.

Create a replace block command:

```
start file "header.4gl"

replace block llh_a_field customer_state_lookup

            if llh_lookup("state",false) = false and  # change second
              length(this_data) != 0                  # argument to "false"
            then
                call fg_err(1)     # No State Code for what you entered
                let nxt_fld = "state"
                return
            end if
            ;
```

Creating a Post-Processor

The Code Generator allows you to customize generated code by running a post-processor on the code after generation. This type of customization is useful for global changes that affect many programs.

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

For example, assume you have written a more relevant initialization routine (say, `chg_init()`) than the generic `init()` function that is created by the Code Generator. You want `main.org` to call `chg_init()` rather than the `init()` function. You can set up a post-processor to change the initialization call in `main.org` to `chg_init()`. The steps in setting up this type of post-processor program are as follows:

1. Write a program (`chg_init` for example). It might be a shell script that runs "sed" on `main.org` as follows:

   ```
   #chg_init
   sed "s, call init, call chg_init," < main.org > main.tmp
   mv main.tmp main.org
   ```

2. Set your `$local_scr` environment variable to the name of the post-processor script (you might want to do this in your .profile file):

```
Bourne Shell
local_scr=chg_init; export local_scr

C Shell
setenv local_scr chg_init
```

Once the Code Generator completes the generation of the application, the local `main.org` file contains the function call `chg_init()` rather than `init()`.

Note

Post-processors must operate on the .org files. If you try to change something in a .4gl file, those changes are lost because the Featurizer copies the contents of the .org file into the .4gl file before merging the .ext and .trg files.

The `lib_message` Function

The message function `lib_message()` displays certain messages at fixed locations on a screen. Here is a list of events that can be passed (as arguments) to `lib_message` and their results:

`zoom_off` displays "======" at line 3

`zoom_on` displays "(Zoom)" at line 3

`note_off` displays "=======" at line 3

`note_on` displays "(Notes)" at line 3

`sort` displays "Sort: [ESC] when..." message at line 1

`find` displays "Find: [ESC] to Find..." message at line 1

`zoom` displays "Zoom: [ESC] to Select..." message at line 1

`update` displays "Update: [ESC] to Store..." message at line 1

`add` displays "Add: [ESC] to Store..." message at line 1 `tab` displays ", [TAB] Next Window" at line 1

`scroll` displays "Scroll: [TAB], [DEL],..." message at line 1

`scr_bottom` displays the value in scratch at calculated screen bottom

`textview` displays "View[ESC] or" message at line 1

`choose` displays "Choose: [ESC] to Select," message at line 1

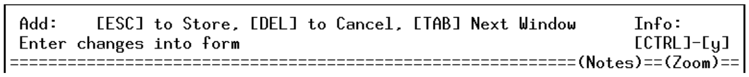
`errchose` displays "Errors: [CTRL]-[z] to..." message at line 1

`cur_path` displays "Cursor Path: [ESC] to ..." message at line l

`help` displays "Help: " at line 1

For some of these events, the variable `scratch` can be set to a message just before the call to `lib_message`. The result would be the message being displayed at the fixed location on the screen. As always, care should be taken when manipulating the value of `scratch`.

Here is an example of how `lib_message` appears on a running application:



```
 Add:    [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Info:
 Enter changes into form                                          [CTRL]-[y]
============================================================(Notes)==(Zoom)==
```

Just below the ring menu options on the right side of the menu line, ("=======") is the result of `lib_message`. The argument `note_on` and `zoom_on` are passed to `lib_message` causing (Notes) and (Zoom) to appear on the right side of the menu line.

Modifying `lib_message`

All of the messages that are accessible through `lib_message` are held in the Fitrix *Screen* message table, `stxmssgr`. Messages such as (Zoom), (Notes), Update: [ESC] to Select, Errors: [CTRL]-[Z] to ... are stored in this table. If you have messages that you want to appear on the screen, you can add your messages to the message table and call your message with `lib_message`. Say you have a simple message called "Gee, I'm happy today" to display to the screen. Using `lib_message` involves a three-step process:

1. Add the message to the `stxmssgr` table.

2. Modify the library function `lib_message`.

3. Call `lib_message` to make your custom message appear.

Step 1 - Adding the message to `stxmssgr.`

All messages used in Fitrix *Screen* generated programs are held in the table `stxmssgr`, a schema of which looks like this:

```
language char(3),
mssg_module char(8),
mssg_program char(8),
mssg_number smallint,
message char(132)
```

Notice how messages are uniquely defined by module, program and number, just like error text, help text, navigation items, and hot key definitions. You can specify any module or program you wish. If your message is throughout several input programs, you may wish to use the generic Fitrix *Screen* module and program and make the `mssg_module` and `mssg_program "lib_scr"` and `"message"` respectively. You can add your row to `stxmssgr` as follows (in unload file format):

```
ENG|lib_scr|message|39|Gee, I'm happy today|
```

Notice how a unique `mssg_number` of 39 is assigned. There is a static array for `lib_message` that holds each message with `mssg_module` of "`lib_scr`" and `mssg_program` of "`message`." This array size is currently at 38, so in this example number 39 was assigned to this message.

Step 2 - Modifying `lib_message`.

The library function `lib_message` needs to be modified to access your new message. Since it is a library function, make a custom library adjacent to `scr.4gs` and copy in `message.4gl` (the .4gl file that contains `lib_message`). Make your modifications there. Building and compiling custom libraries is documented in "Creating and Using Custom Libraries" on page 15-17.

As an overview of the function `lib_message`, two things occur:

1. Upon the first call to `lib_message`, `lib_message`'s static array is filled with all of Fitrix *Screen*'s generic messages (`mssg_module` of "standard" and `mssg_program` of "message").
2. The message is displayed to the screen at the location specified. The location is indicated by the argument passed to `lib_message` . Within `lib_message`, this argument is called "funct."

So, to modify `lib_message` to accommodate your new message ("Gee, I'm happy today"), you add code to `message.4gl` as follows:

1. At the very top is the static array that holds all the messages. Bump this up from "38" to "39" to make room for our new message:

```
#   arr_mesgs array[38] of record    # old Message text
    arr_mesgs array[39] of record    # Message text
        mssg_text char(132)
    end record,
```

2. An "if" statement loads this static array the first time `lib_message` is called ("if mssg_prep is null..."). In `lib_message`, the function `fg_message()` is called passing `mssg_module`, `mssg_program`, and `mssg_number` (just like `lib_error`). `fg_message` does the select on `stxmssgr` and returns the result. At the very end of this "if" statement, add the call to `fg_message` to access your message #39. Make sure and add the comment so you know what message is in element number 39 of the array:

```
       let arr_mesgs[39].mssg_text = fg_message("lib_scr","message",39)
#39: "Gee, I'm happy today."
       let mssg_prep = "Y"
     end if
```

3. A CASE statement that evaluates the argument passed to `lib_message` ("funct"). Each WHEN clause evaluates "funct" and displays the message at the hard-coded location. For this example just display the message to the screen. The default location of the lower left hand corner is used:

   ```
   when funct = "gee"
     display arr_mesgs[39].mssg_text clipped
   ```

   The WHEN clause is placed anywhere in the CASE statement.

   When specifying a location, there are some helpful library-level variables that can be used, such as `num_rows` and `num_cols`. `num_rows` stores the number of rows in the current window (1-24). `num_cols` stores the number of columns in the current window (1-80). The library function `ring_clear` simply clears the menu lines on the very top.

The preceding are all the modifications necessary to `lib_message`. After compiling, this modified library function is ready to be called from your local program directory.

Step 3 - Calling `lib_message` from the program directory

The call is made entirely with triggers. Place the call anywhere you want it to appear in the input program. For this example, it is put in after-field logic:

```
after_field po_num
      call lib_message("gee")
       ;
```

Notice how the argument "gee" is passed to invoke our "Gee, I’m happy today" message. If you have put the modified `message.4gl` file in a custom library, don’t forget to include the custom library in the list of libraries to link in (LIBFILES in the Makefile). Use the `libraries` or `custom_libraries` trigger to do this. Regenerate, recompile, and test.

Shell Escapes and UNIX Commands

When running a generated program in the standard UNIX Bourne shell you can "drop out" to the operating system by typing an [!] and then the command sh.

When the [!] is pressed, the menu prompt changes to:

```
System:  Enter command or [DEL] to quit
:
```

Typing a command at this prompt is equivalent to typing the same command before entering the Fitrix *Screen* program.

Fitrix *Screen* has implemented this as an event that can be used with Navigation, or assigned to a hot key. The default hot key definition for this event is [CTRL]-[o].

Screen output from operating system commands appear at the bottom of the screen and force the Fitrix *Screen* form to scroll upward (unless the command redraws the screen. When the command is completed, the program prompts:

```
Press [ENTER] to continue.
```

Pressing any key causes the program to redraw the form and return to the Fitrix *Screen* program.

Preventing Shell Escapes

You may prevent users from access to the shell by setting the `$SHELL` variable to false. To prevent your users from shelling out of your Fitrix *Screen* generated program set the following variable:

```
SHELL=false; export SHELL
```

This variable should be set in the `$fg/bin/fg.startup` script. If you use Fitrix *Menus* to run your program, the `$SHELL` variable is automatically set to `/bin/sh` in `fg.startup`. Therefore you need to put the SHELL=false; export SHELL command just before the mz command in `fg.startup`.

16

Version Control

This chapter discusses version control; a concept that allows you to easily maintain different versions of your application programs without duplicating code. This chapter explains:

- What files are affected by version control
- How to organize your applications to take advantage of version control
- What special triggers are affected by version control
- How the featurizer works with version control
- How to run programs when using version control
- How Fitrix *Menus* works with version control

Introduction to Version Control

Version Control is a key to designing applications that grow with the users needs rather than becoming outdated as technology advances and user needs and desires change.

Version control is useful whenever it is helpful to distinguish two or more versions of the same program(s). These situations include:

- When the programs are to be used by two or more sets of users, who may have different desires regarding functionality.
- "Co-development" arrangements in which two or more groups of programmers are contributing features and fixes to the code.
- When a base product is to be modified by others.
- When an application or module is to be offered in various "suites," usually varying with the size of the company or corporation.
- When different sets of features are selected by different groups of users, as in different business units and offices of a corporation.
- When a module in live use by customers is being upgraded. New versions can be "turned on and off" during testing and review.

The multi-version features of Fitrix *Screen* improve management of these situations and reduce duplication of code and work in creating and maintaining multiple versions of programs.

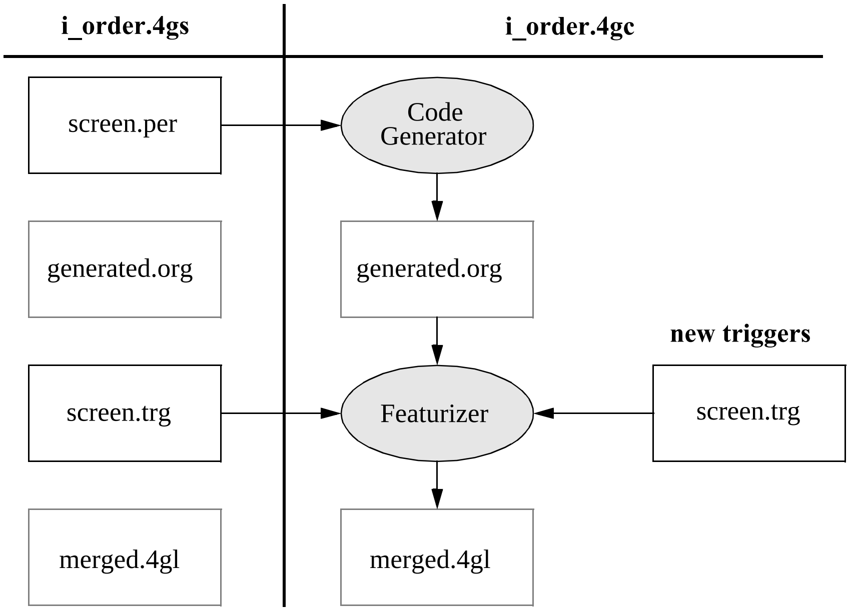
Fitrix *Screen* allows you to easily maintain different versions of your application programs without duplicating code. Version control allows you to share common files such as .per forms, .trg, and .ext files. What this means is that you do not have to copy these files to every program directory if you are creating multiple versions of your programs. Without version control, you would need to have all .per, .trg, and .ext files present in every program directory. Duplicating code consumes space and becomes a nightmare to maintain.

In order to take full advantage of version control, be sure you read everything in this chapter. You must learn what files are used and when, and which triggers or blocks are used to build your program.

There are several facets to version control.

1. **Directory structure.**
2. **The directory search path - `$cust_path`.**
   - changing the default `$cust_path`
   - which .per forms to generate code for
   - which .trg files are merged
   - trigger precedence
   - which .ext files are merged
   - block precedence
3. **Using non-generated .4gl files.**
4. **Running programs when using version control.**
5. **Fitrix *Menus* and version control.**

The following diagram gives a simple scenario of how version control works. In this example, the base program exists in the `i_order.4gs` directory. In order to customize this program by adding new triggers, an `i_order.4gc` directory is created. All changes to the .4gs program are added to a new trigger file in the .4gc directory. The new value added program is created by running the Code Generator and then compiling the program, which invokes the Featurizer. The Code Generator and Featurizer use code found in the base directory along with the new triggers to create the new version of the program.

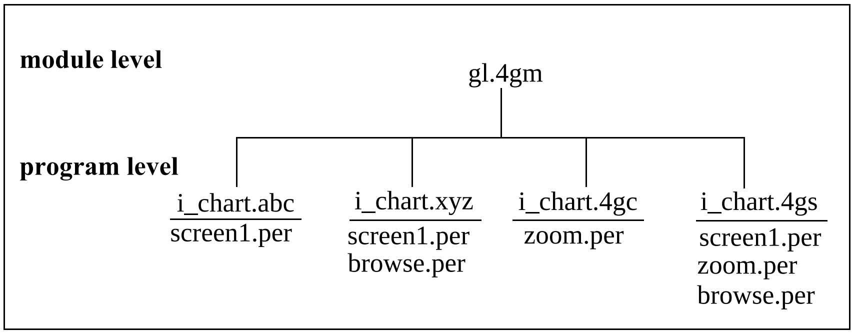


Required Directory Structures

The discussion of version control begins first with how it works with .per files only. This serves to introduce you to the concept of version control. Once you understand the basic concept you learn how version control works with triggers, blocks, and the Featurizer.

The program directory extension, known as the `cust_key`, is stored in the database along with the module basename, the program basename, and the screen basename. The directory extension is the key piece of information used by version control to determine where it is and where it needs to go. Think of directory extensions as road signs. Both the Code Generator and the Featurizer identify where .pers, .trgs, and .exts are located by the extension of the current directory.

In order to use version control effectively, you must follow the module-program directory structure illustrated below. In this example, the module is General Ledger, and the programs are different versions of an `i_chart` program. Listed under the directory name are perform file names.



Version control requires extensions on your program directories. We recommend the following conventions:

> **4gs extension** -The program directory that contains the base version of the program. In the example above, the base product contains `screen1.per`, `zoom.per`, and `browse.per`.

4gc extension - The program directory for the “value-added” directory. This contains additional changes on top of your base package that you want to apply to all your customers. In the example above, you may have decided to modify the base `screen2.per`. Perhaps you added an extra field. You would want this changed `screen2.per` to apply to all your clients, so you place it in the .4gc directory.

Any other extensions are considered to be client-specific program directories, created to further specialize programs for specific client needs:

xyz extension - The special program directory for your “xyz” client. For example, this client may want both a different browse screen, and additional fields to the `screen2.per` that the value-added `screen2.per` does not have. Both custom perform files exist in this .xyz directory.

abc extension - The special program directory for your “abc” client. For example, this client may want a field removed in `screen1.per`.

Version Control and the Code Generator

When using version control, the Code Generator automatically locates all of the .per files needed to generate the program, even if they are not contained within the current directory. In order for the Code Generator to automatically find the correct .per files, you need to invoke it as follows:

```
fg.screen -dbname database
```

Notice that no perform files are specified. When invoked in this manner, the Code Generator makes a list of the perform files in the current directory, and then looks in the next directory down in the hierarchy. The default hierarchy of directories is as follows, from highest to lowest:

```
client-specific  - in the example above, “abc” and “xyz”
value-added      - in the example above, “4gc”
base             - in the example above, “4gs”
```

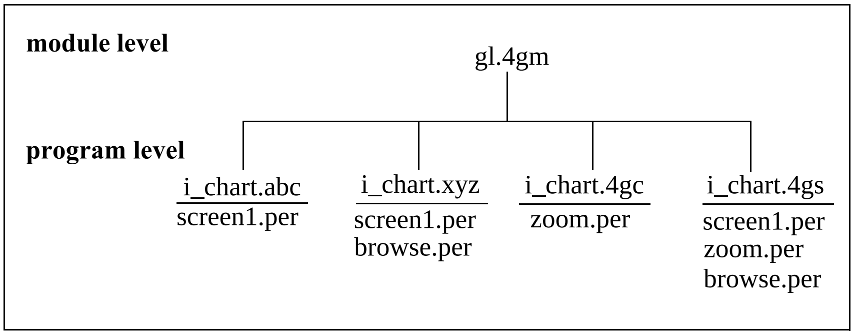
The directories are always scanned (when `$cust_path` is not set) from the bottom of the hierarchy up to the current location.

If you run the Code Generator in the base (.4gs) directory, which is the lowest, it does not look in the value-added (4gc) directory for perform files.

If you run the Code Generator in the .4gc directory, the .pers in the .4gs directory are found first, and then the .pers in the .4gs directory. If any .pers exist in the .4gc directory that have the same name as a .per found in the .4gs directory, the .per in the current .4gc directory is used instead.

You can change the search path the Code Generator uses to find .per files by setting an environment variable called `$cust_path`. For more information on the `$cust_path` variable refer to "Changing the Version Control Search Path (`$cust_path`)" on page 16-11.

By running the Code Generator without specifying the .per files and without setting the `$cust_path` variable, the following files are processed:



When the Code Generator is run from `i_chart.4gs`, it uses only the .per files contained there.

```
i_chart.4gs: The files used would be:
      ../i_chart.4gs/screen1.per
      ../i_chart.4gs/zoom.per
      ../i_chart.4gs/browse.per
```

When the Code Generator is run from the `i_chart.4gc` directory, the `zoom.per` in the .4gc directory takes precedence over the `zoom.per` in the `i_chart.4gs` directory and it is used to create the application.

```
i_chart.4gc: The files used would be
      ../i_chart.4gc/zoom.per
      ../i_chart.4gs/screen1.per
      ../i_chart.4gs/browse.per
```

When the Code Generator is run from the `i_chart.xyz` directory, the `zoom.per` and the `browse.per` contained there take precedence over the `zoom.per` and the `browse.per` contained in the value-added (.4gc) and the base (.4gs) directories.

```
i_chart.xyz: The files used would be
      ../i_chart.xyz/zoom.per
      ../i_chart.xyz/browse.per
      ../i_chart.4gs/screen1.per
```

When the Code Generator is run in the `i_chart.abc` directory, it uses the local (.abc) `screen1.per`, the `zoom.per` in the value-added (.4gc) directory, and the `browse.per` from the base (.4gs) directory.

```
i_chart.abc: The files used would be
      ../i_chart.abc/screen1.per
      ../i_chart.4gc/zoom.per
      ../i_chart.4gs/browse.per
```

The `../directory/screen_name` format is used in the creation of the `Makefile FORMS=...` line. The correct perform files are then compiled with `fg.make`.

If the Code Generator is invoked with perform files specified on the command line, the Code Generator does *not* search other directories in the hierarchy as described above. The Code Generator creates the `FORMS=...` line in the `Makefile` with *only* those perform files found in the current directory.

Preventing Code Generation on a Base .per Form

There may also be cases where you want to disable a particular screen for a client. To do this you need to copy that screen into that client's directory, and edit the form by adding the line `non_source_form`, so that code is *not* generated from it. You need to copy the form into the clients directory, because if you don't, the Code Generator finds the real perform file further down the hierarchy.

The `non_source_form` statement is placed in the .per immediately following the copyright statement within the first set of braces {}.

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

non_source_form}
```

The Featurizer and Version Control

The Featurizer merges triggers and blocks into generated source code and supports version control. The Featurizer allows you to merge triggers and blocks found in different directories into source code in your current program directory. If you are developing in multiple directories, you do not need to recopy trigger files or .ext files into your current directory. This avoids keeping duplicate code in multiple versions of the same program, which saves disk space and simplifies maintenance.

When using version control, the goal is to keep only the differences of a version in a program directory. For instance, for user ABC Company's version of the `i_invce` program, the .trg and .ext files in the `i_invce.abc` directory would contain only the custom triggers and features needed to create ABC Company's custom version. All of the .trg and .ext files that are part of the base product remain in the .4gc and .4gs directories.

The Featurizer looks at the `$cust_path` variable to determine where to look for files to merge. If you run the Featurizer in a custom directory such as .abc without setting the `$cust_path` variable, files are processed from program directories in this order:

1. .4gs
2. .4gc
3. any other 3 character extension (for example, .abc)

When you invoke the Featurizer from an ".abc" directory, the Featurizer immediately looks for a .4gs directory with the same prefix (`i_invce` in the above example) and then loads all code from the .trg file and all code from any specified .ext files. Remember the feature must be plugged in via a `base.set` file for the Featurizer to load the .ext file for that feature.

Next, the Featurizer looks for a .4gc program directory with the same prefix (`i_invce` in the above example). Again, it loads .trg logic and .ext logic as instructed. If the .4gc directory does not exist, then it skips this step.

Finally, the Featurizer returns to the user-specific directory from which it was invoked and loads in any remaining .trg and .ext logic from this directory.

The end result of pre-processing is a suite of .4gl source code files containing plugged in features from .4gc or .4gs program directories, plus any trigger or feature logic specific to that user only.

Trigger and Block Command Priority

The order that a trigger or block command is merged determines its priority. Like .per files in the current directory take precedence over all .per files found elsewhere in the directory path, block commands and most triggers replace identical commands. However there are some special triggers that can be specified to either replace or append subsequent triggers. These special triggers are described on page 16-15.

Specifying Which .ext Files to Merge

When using version control and .ext files, you must remember to place a `base.set` file in every directory you generate or compile in in order to merge selected .ext files. With version control, .ext files are not automatically merged from parallel directories like triggers. You must specify every .ext file that you want to merge into your local program. This means that you have to list every .ext file in every directory in the search path.

The best technique for making sure the necessary .ext's are merged is to copy the `base.set` file from the previous directory in the search path into the local directory. After copying the `base.set` file, you then add to the list of .ext's contained within it any local .ext's.

Changing the Version Control Search Path (`$cust_path`)

You can set a UNIX environment variable called `$cust_path` to inform the Code Generator and Featurizer which order and which directories to search for .per, .trg, and .ext files. The default `$cust_path` is `4gc:4gs`. The `$cust_path` variable accepts three character extensions separated by semi-colons.

For example, if you set `$cust_path` to `4gc:abc:4gs`, and you generate or compile from the .4gc directory, the tools first search the 4gs directory, then the abc directory, and finally the .4gc directory for .per, .trg, and .ext files to process.

If you generate in a directory *not* specified in the `$cust_path`, the generator looks at the last directory in the `$cust_path` first and then continues up the `$cust_path` until it finally reaches the current directory.

For instance, you could instruct the Featurizer to read .trg and .ext files in this order:

1. .4gs
2. .abc
3. .4gc

or perhaps:

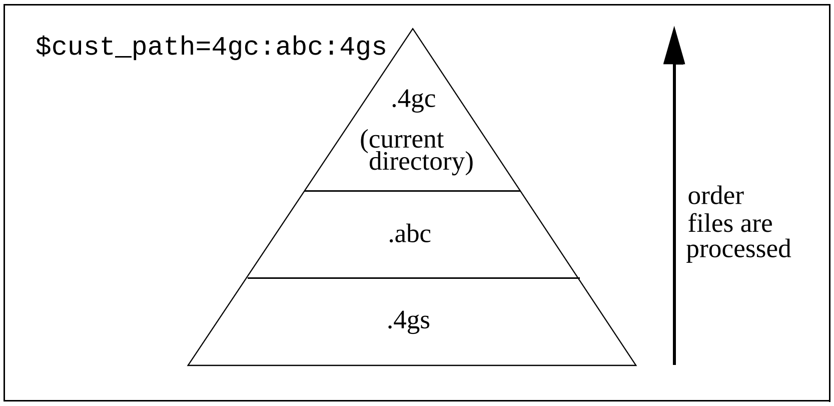
1. .abc
2. .4gc
3. .4gs
4. .xyz

The `$cust_path` variable also determines the order .trg files and .ext files are merged into the .org code. Although the current directory is the top of the hierarchy and the triggers found there take precedence over all similar triggers in the `$cust_path`, it is the last one processed. The Featurizer processes the triggers in the last directory in `$cust_path` first, then works its way up to the current directory.

Note

The order directories are searched in the `$cust_path` is from the last extension listed to the first. This gives the first extension in the path precedence.

An example of trigger hierarchy follows:



If you have your `$cust_path` set to `4gc:abc:4gs`, and you run the Featurizer from the .4gc directory (the .4gc directory is current), then the Featurizer first searches the .4gs directory for any .trg and .ext files and merges them into the .4gl files in the current directory, then it searches the .abc directory and merges those .trg files into the .4gl files in the current directory, and finally the .trg files in the current directory are merged into the current .4gc directory.

The hierarchy depends on what directory you are in when you invoke the Featurizer. Triggers in the current directory take precedence over all triggers from there on down the `CUSTPATH` hierarchy.

The Makefile's `CUSTPATH` Variable

Setting the `$cust_path` environment variable before code generation causes the contents of the variable to be written to the `CUSTPATH` variable in the resulting `Makefile`. If a `CUSTPATH` setting already exists in a `Makefile`, re-generation does not cause the current setting of `CUSTPATH` to be overwritten. The new `Makefile` keeps the `CUSTPATH` setting in the old `Makefile`. For more information on the `$cust_path` variable refer to page 16-11.

Another way to specify `CUSTPATH` is by setting it in the `Makefile`, as follows:

```
#  Makefile for an Informix-4GL program
```

```
NAME =      davidh.4ge

OBJFILES =  globals.o browse.o cust.o cust_zm.o \
            detail.o header.o main.o midlevel.o \
            stk_mnu.o stockzm.o

FORMS =     ../davidh.4gs/browse.frm

CUSTPATH =  4gc:abc:4gs

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

GLOBAL =    globals.4gl

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

all:
         @make -f $(fg)/Make/program NAME="$(NAME)" \
         LIBFILES="$(LIBFILES)" OBJFILES="$(OBJFILES)" \
         FORMS="$(FORMS)" GLOBAL="$(GLOBAL)" $(NAME)

clean:
         @make -f $(fg)/Make/program NAME="$(NAME)" \
         LIBFILES="$(LIBFILES)" OBJFILES="$(OBJFILES)" \
         FORMS="$(FORMS)" GLOBAL="$(GLOBAL)"clean
```

This CUSTPATH variable is set in your program, module, or application directory Makefile. If a CUSTPATH does not appear in the Makefile of the current program directory, then the Featurizer looks at the Makefile in the Program directory for a CUSTPATH. If it can't find it there, then it looks in the Makefile for the application. The Featurizer, when invoked, searches for CUSTPATH values in the following order:

1. CUSTPATH setting in program directory Makefile
2. CUSTPATH setting in module level directory's Makefile.org
3. CUSTPATH setting in application level directory's Makefile
4. $cust_path UNIX environmental variable

Special Trigger Processing

If the same trigger is specified in a custom directory that exists in a .4gs or another lower custom directory in the `CUSTPATH`, the trigger in the custom directory replaces all similar triggers found before it. The `CUSTPATH` determines which triggers are applied first. If a trigger is applied to a section of code where a trigger has already been placed, it acts as a `replace_block` command, and replaces the previous trigger with the more "local" trigger.

An example would be if your `CUSTPATH` is `4gc:4gs` and you are in `i_chart.abc`. If you specify a "before field" trigger and that same trigger is specified in `i_chart.4gc` and `i_chart.4gs`, the Featurizer first inserts the trigger from the .4gs directory, then replaces it with the trigger in the .4gc directory, then replaces that with the trigger in the local .abc directory.

Triggers react differently when using version control.

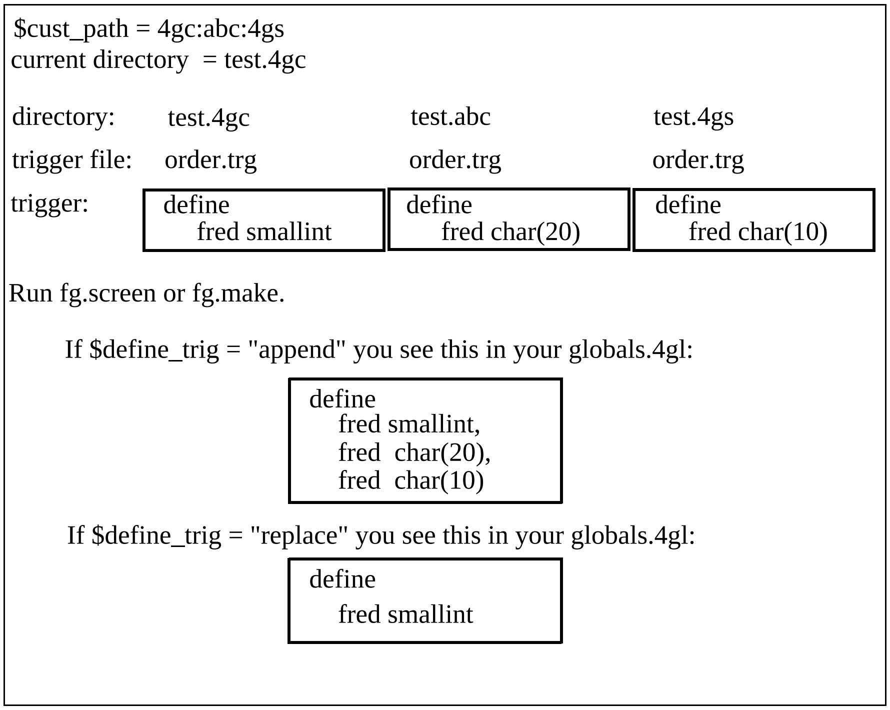
1. Most triggers replace identical triggers.
2. There are some special triggers, however, which you can specify to either append or replace their counterparts.
3. There are other special triggers that do not replace their original counterpart if specified in custom directories and always append.

Here are some examples:

- The `switchbox_items` trigger either replaces or appends to other `switchbox_items` triggers in higher order directories.
- The `define`, `static_define` and `function_define` triggers either replace or append to like triggers in higher order directories.
- The `do_not_generate` trigger acts as a "delete block" command, and once deleted, they can't be un-deleted in other custom directories.
- The `at_eof` trigger either replaces or appends to other `at_eof` triggers in higher order directories.

You can determine how these special triggers function in order to preserve backward compatibility with older generated code. For information on selecting the append/replace mode refer to "The Featurizer Options File (fglpp.opt)" on page 2-19.

The following diagram illustrates how the Featurizer works with `$cust_path` and how the `define` trigger is treated.

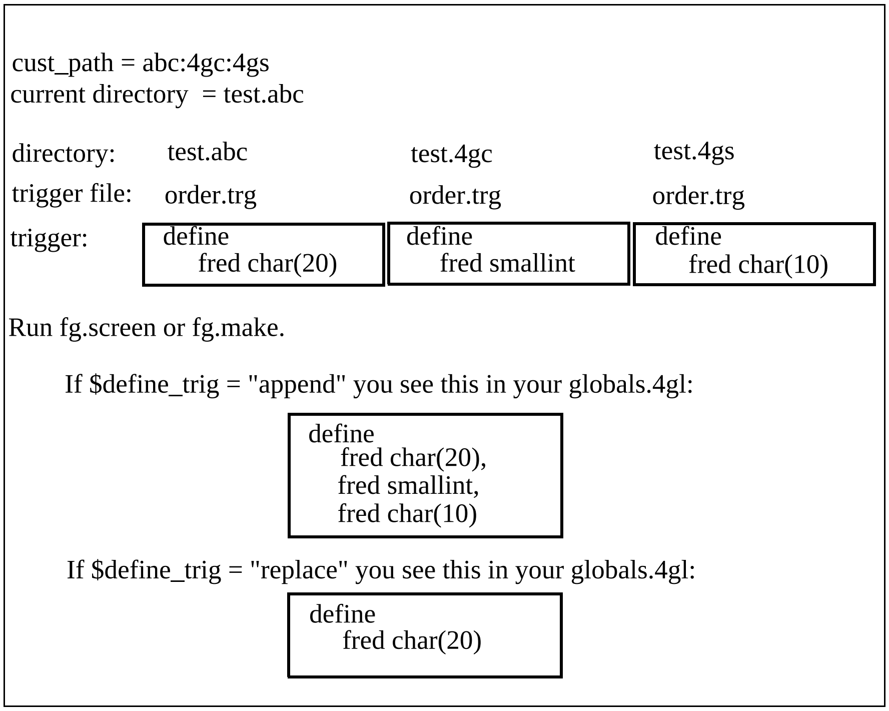


```
$cust_path = 4gc:abc:4gs
current directory  = test.4gc

directory:      test.4gc            test.abc            test.4gs
trigger file:   order.trg           order.trg           order.trg
trigger:        define              define              define
                   fred smallint       fred char(20)       fred char(10)

Run fg.screen or fg.make.

     If $define_trig = "append" you see this in your globals.4gl:

                    define
                        fred smallint,
                        fred  char(20),
                        fred  char(10)

     If $define_trig = "replace" you see this in your globals.4gl:

                    define
                        fred smallint
```

Note

The `$define_trig` variable is specified in the `fglpp.org` file in `$fg/codegen/options`. This variable controls how the Featurizer handles the special `define` trigger when encountered using version control. For more information on the `$define_trig` refer to "The Featurizer Options File (fglpp.opt)" on page 2-19.

This diagram is similar to the previous diagram, only a different `CUSTPATH` is used.



cust_path = abc:4gc:4gs
current directory = test.abc

directory:	test.abc	test.4gc	test.4gs
trigger file:	order.trg	order.trg	order.trg
trigger:	define fred char(20)	define fred smallint	define fred char(10)

Run fg.screen or fg.make.

If $define_trig = "append" you see this in your globals.4gl:

```
define
    fred char(20),
    fred smallint,
    fred char(10)
```

If $define_trig = "replace" you see this in your globals.4gl:

```
define
    fred char(20)
```

The `define` Trigger

When using multiple directory search paths, the Featurizer either *replaces* or *appends* `define` trigger definitions processed previously. The action that the Featurizer takes depends on the current setting in the `fglpp.opt` file. For more information refer to "Maintaining Backwards Compatibility—The Options Files" on page 2-19.

The `static_define` Trigger

When using multiple directory search paths, the Featurizer either *replaces* or *appends* `static_define` trigger definitions processed previously. The action that the Featurizer takes depends on the current setting in the `fglpp.opt` file. Refer to page 2-19 for more information.

The `at_eof` Trigger

When using multiple directory search paths, the Featurizer either *replaces* or *appends* `at_eof` trigger definitions processed previously. The action that the Featurizer takes depends on the current setting in the `fglpp.opt` file. Refer to "Maintaining Backwards Compatibility—The Options Files" on page 2-19.

The `switchbox_items` Trigger

When using multiple directory search paths, the Featurizer either *replaces* or *appends* `switchbox_items` trigger definitions previously processed. The action that the Featurizer takes depends on the current setting in the `fglpp.opt` file. Refer to "Maintaining Backwards Compatibility—The Options Files" on page 2-19.

Using Non-Generated .4gl files With Version Control (`fg_funcs.4gl`)

When using version control, the Code Generator and the Featurizer automatically handle the copying of generated .4gl files from your base directory to your custom directory. However, if you have custom .4gl files such as an `fg_funcs.4gl` in your base directories, you have to take a special step in order for that custom .4gl file to get used in your local directory.

You can simply copy your custom files into the local directory by hand but this would mean duplicating code and would require double maintenance. Every time you needed to modify your base custom .4gl file you would need to re-copy that file to each custom directory that needs it.

Version control can automatically copy these custom.4gl files provided you place a start file "fg_funcs.4gl" command in an .ext file in your custom directories. This causes the Featurizer to automatically copy the file into the directory that needs it.

Thus when you make a change in the base directory and subsequently regenerate code in the custom directory, the change in the base directory is automatically propagated to your custom directory.

Invoking Programs That Use Version Control

When a program is run it needs to be able to locate compiled .frm files for each .per file that makes up the program. When using version control, not all the perform files used in code generation are in the local directory; the program must be told where these non-local perform files exist. There are two ways you can inform the program where to find non-local perform files:

1. modify $DBPATH to include all directories that contain a source form or,
2. use the fg.go/fg.db script to run the program.

Modifying $DBPATH

By modifying the environment variable $DBPATH to include the directories where these perform files exist, your program is able to open these perform files at runtime.

Using the previous example, if you go into i_chart.abc and run the program there, your $DBPATH would have to include both the i_chart.4gc and i_chart.4gs directories, so that the program could open screen2.frm and browse.frm. The $DBPATH variable would look like this:

```
DBPATH=../i_chart.4gc:../i_chart.4gs:$DBPATH
```

Using `fg.go` and `fg.db`

When running a finished program from the command line, there are two shell scripts you can use that set up your `$DBPATH` variable for you, so you do not have to pre-set it as described above. These scripts automatically prepend the path to the .4gc and .4gs directories to the `$DBPATH`. So if you run `fg.go` from the .abc directory, the program looks first for the presence of all .frm files in the local directory, and if any .frm file is not found, then the .4gc directory is searched, and then the .4gs.

`fg.go`: modifies `$DBPATH` and runs the program. This shell script detects if the program is an RDS program and runs it with `fglgo` or as an executable.

Examples:

```
fg.go program.4gi -dbname database
fg.go program.4ge -dbname database
```

`fg.db`: runs the finished program under the RDS Debugger, if the finished program is an RDS program.

Example:

```
fg.db program.4gi -dbname database
```

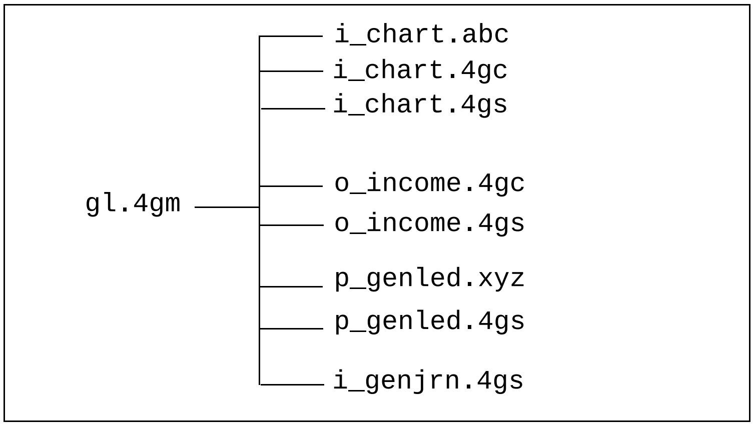
Again, the use of these two shell scripts is required when running the finished program from the command line. As mentioned before, these scripts adjust your `$DBPATH` setting properly, and then restore `$DBPATH` to its original setting upon exit of the program.

Running Programs with Fitrix *Menus*

Fitrix *Menus* can take advantage of version control by allowing you to pick and choose which particular version of a program you want to run. You can do this by setting an environment variable called `$cust_key` to the directory extension the program resides in. For example, you could set `$cust_key` to "abc" to inform Fitrix *Menus* to run the program in a `i_chart.abc` directory.

Fitrix *Menus* runs the program contained within the directory matching the `$cust_key` variable. If a particular version of the program doesn't exist for the `$cust_key`, Fitrix *Menus* searches the program directories for the program to run in the same hierarchical fashion as used by version control in the Code Generator.

Take the following example:



Above is a general ledger module (`gl.4gm`), with different program directories underneath. Multiple versions of some of the programs exist. The following diagram depicts which executable would be selected based on the `$cust_key`.

```
cust_key=4gs
  i_chart.4gs
  o_income.4gs
  p_genled.4gs
  i_genjrn.4gs

cust_key=4gc
  i_chart.4gc
  o_income.4gc
  p_genled.4gs - since there is no p_genled.4gc
  i_genjrn.4gs - since there is no i_genjrn.4gc

cust_key=xyz
  i_chart.4gc  - since there is no i_genjrn.xyz
  o_income.4gc - since there is no i_genjrn.xyz
  p_genled.xyz
  i_genjrn.4gs - since there is no i_genjrn.xyz or i_genjrn.4gc

cust_key=abc
  i_chart.abc
  o_income.4gc - since there is no o_income.abc
  p_genled.4gs - since there is no p_genled.abc or p_genled.4gc
  i_genjrn.4gs - since there is no i_genjrn.abc or i_genjrn.4gc
```

So at any time, you can set the `$cust_key` variable, run Fitrix *Menus* , and see what the product looks like for any particular customer.

The Relationship Between `$cust_key` and `$cust_path`

The previous examples all demonstrate what happens when `$cust_key` is set and the default `$cust_path` of `4gc:4gs` is used. It is also important to understand how the `$cust_key` variable relates to `$cust_path`.

If `$cust_key` is not set, Fitrix *Menus* looks at the `$cust_path` variable to determine where to look for the program. If both `$cust_key` and `$cust_path` are not set, then the default path of 4gc:4gs is used.

If the `$cust_key` is in the `$cust_path`, then Menus looks in the `$cust_key` directory first, and if a program is not found there, it continues down the `$cust_path` until a program is found.

For example, if `$cust_path` is set to `.abc:4gc:4gs`, and `$cust_key` is set to .abc, then Menus searches for a program first in the .abc directory, then the 4gc directory, and finally the 4gs directory.

If `$cust_path` is set to `.4gc:abc:4gs`, and `$cust_key` is set to .abc, Menus searches for a program first in the .abc directory, then the 4gs directory. The .4gc directory is not searched.

Version Control Summary

Pre-Processing: During pre-processing the Featurizer takes the following steps:

1. If the current directory extension is in the search path, then the files there take precedence. The Featurizer then looks for .trg and .ext files in the *last* directory in the `CUSTPATH` first, then it continues on to the previous directory in the `CUSTPATH`, until the current directory is reached.

   For example:

   ```
   CUSTPATH = 4gc:abc:4gs
   ```

   If your `CUSTPATH` is set as above, and your current directory is .abc, then the files in the .4gs and the .abc directory are used. The Featurizer does *not* look in the .4gc directory.

2. If the current directory is not in the search path, the Featurizer begins processing the files in the last directory and then searches each of the directories in the search path for .trg and .ext files, which are each merged into the .4gl files. The files in the current directory are merged last.

Code Generation: When generating code, the Screen Code Generator does the following before it creates the `Makefile`:

1. The Screen Code Generator checks to see if a `CUSTPATH` entry exists in a pre-existing `Makefile`, as in the case of running the Screen Code Generator a second time on a particular program. It uses the `grep` command to check for this `CUSTPATH`. If it finds it, it retains the current `CUSTPATH` setting.

2. If it doesn't find a `CUSTPATH` setting in a `Makefile`, it checks the environment variable `$cust_path` to see if it is set. If so, it uses the value in `$cust_path` when creating this new `Makefile`. The logic explained above retains a possible `CUSTPATH` setting, and does *not* overwrite it when the Screen Code Generator creates a new `Makefile`.

At Runtime: When a program is run, Fitrix *Menus* performs the following:

1. Fitrix *Menus* tries to find a program in a directory that matches the extension set in `$cust_key`. If `$cust_key` is not set, Menus looks for a default 4gc:4gs directory.

2. Fitrix *Menus* also searches the `$cust_path` setting for a program to execute. If `$cust_path` is not set, Fitrix *Menus* uses the default path of 4gc:4gs. If the `$cust_key` is not in the `$cust_path`, Menus prepends `$cust_key` to `$cust_path`. The order that directories are searched depends on the `$cust_path`.

   For example, if `$cust_key` is abc and `$cust_path` is xyz:abc:4gs, Menus looks for the program to run by looking first in a directory with a .abc suffix. If a program is not found there, a directory with the .4gs extension is searched. The .xyz directory would not be searched in this case because it comes before the `$cust_key` setting in the `$cust_path`.

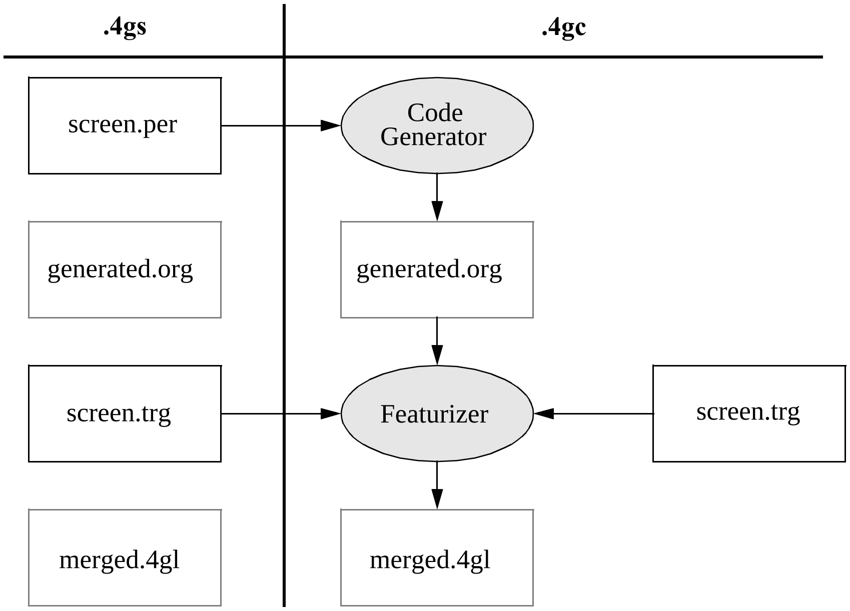
Practical Examples

The following pages contain graphical examples of version control.

Adding a New Trigger to Your Base Program

This example shows you how you can customize your base program by adding a new trigger to a trigger file while utilizing the .per forms and .trg files from the base directory to generate your custom program.

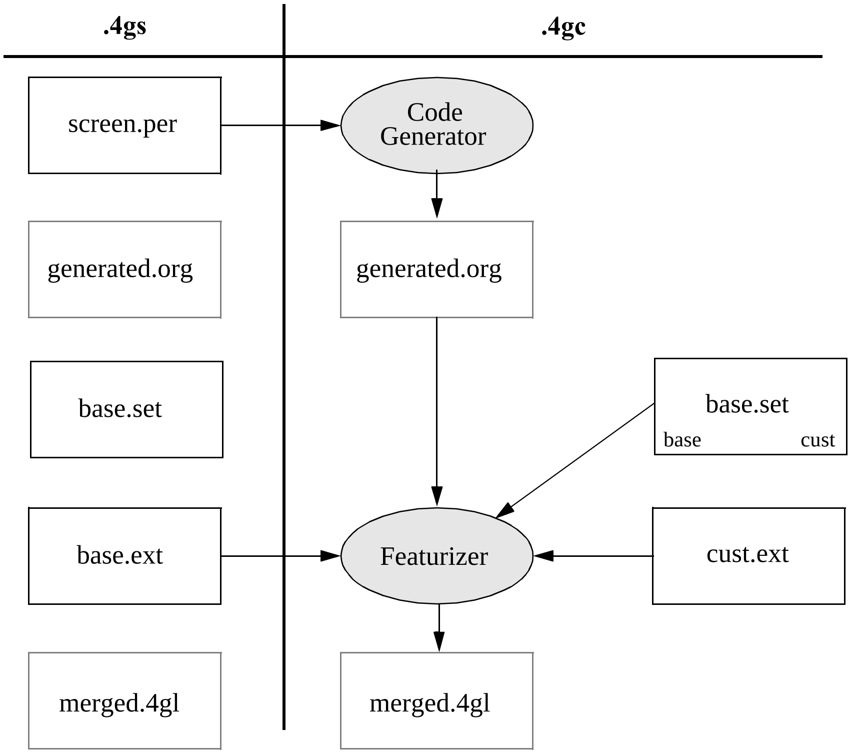
1. **Create a custom directory (.4gc).**
2. **Create a new `screen.trg` file in the .4gc directory.**
3. **Add only the new trigger to `screen.trg` in .4gc. Leave the triggers in the base .trg file alone.**



Customizing Your Base Program with Blocks

This example shows what you need to do to perform a few block operations to customize your base program without having to duplicate the original code.

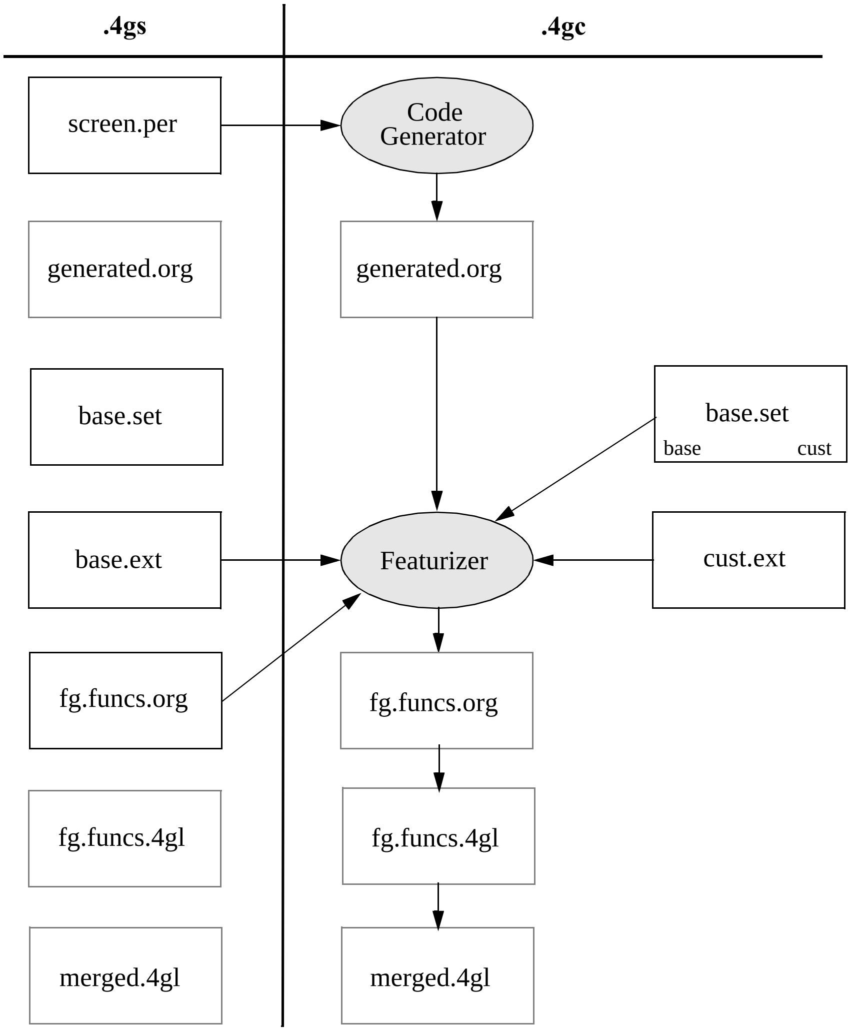
1. **Create a new custom directory (.4gc).**
2. **Create a new .ext file that has a different name from any .ext in the base directory.**
3. **Put the new block command in the .ext in the .4gc directory.**
4. **Copy the `base.set` file from the .4gs directory into the .4gc directory.**
5. **Add the name of the new .ext file to the `base.set` file in the .4gc directory.**
6. **Generate your custom program.**


.4gs
.4gc
screen.per
Code Generator
generated.org
generated.org
base.set
base.set
base
cust
base.ext
Featurizer
cust.ext
merged.4gl
merged.4gl

Pulling a Custom .4gl (`fg_funcs.4gl`) File Into a Custom Directory

If you use version control with a program that utilizes custom 4gl files, (4gl files that are not created by the Code Generator), a special technique is required in order for them to be copied into a custom directory. This tells the Featurizer to automatically copy the .org file associated with the custom file into the custom directory.

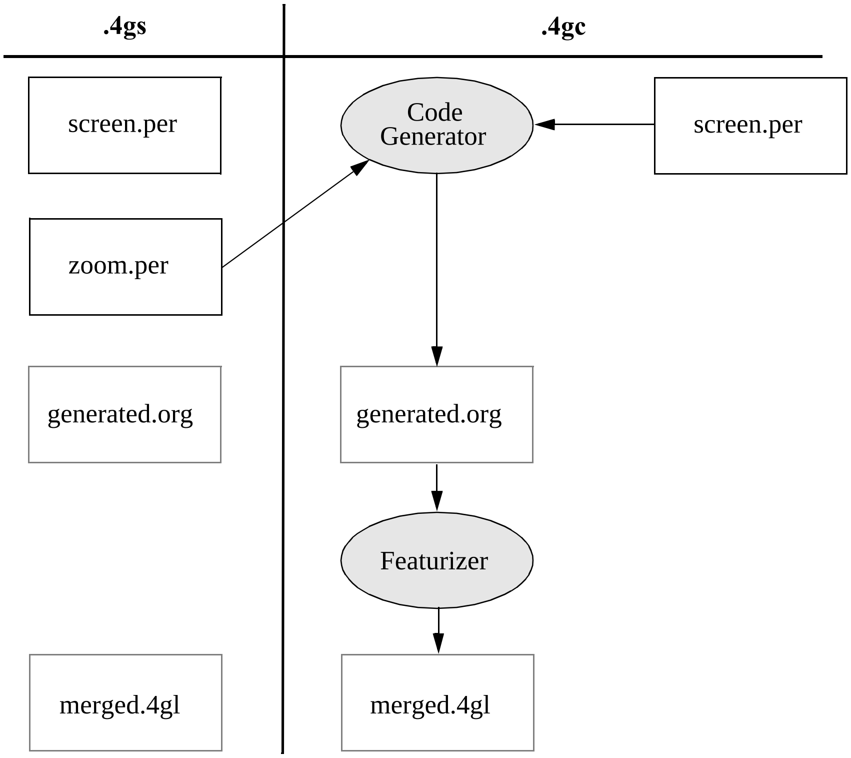
1. **Create a custom directory (.4gc).**
2. **Create an .ext file in the .4gc directory.**
3. **Create a `start_file "fg.funcs.4gl"` command in the .ext.**
4. **Create a `base.set` file and list the names of all .ext's to be merged.**


.4gs
.4gc
screen.per
Code Generator
generated.org
generated.org
base.set
base.set
base cust
base.ext
Featurizer
cust.ext
fg.funcs.org
fg.funcs.org
fg.funcs.4gl
fg.funcs.4gl
merged.4gl
merged.4gl

How to Modify a .per in a Base Directory

Whenever you need to customize a .per form you need to copy it to a custom directory and perform the modifications there. Any .per forms found in the local directory are used in place of any similarly named .per in any other directory found in the `$cust_path`.

1. **Create a custom directory.**
2. **Copy the .per you want to modify from the .4gs directory to your custom directory.**
3. **Modify the .per form in the custom directory.**
4. **Run the Code Generator.**

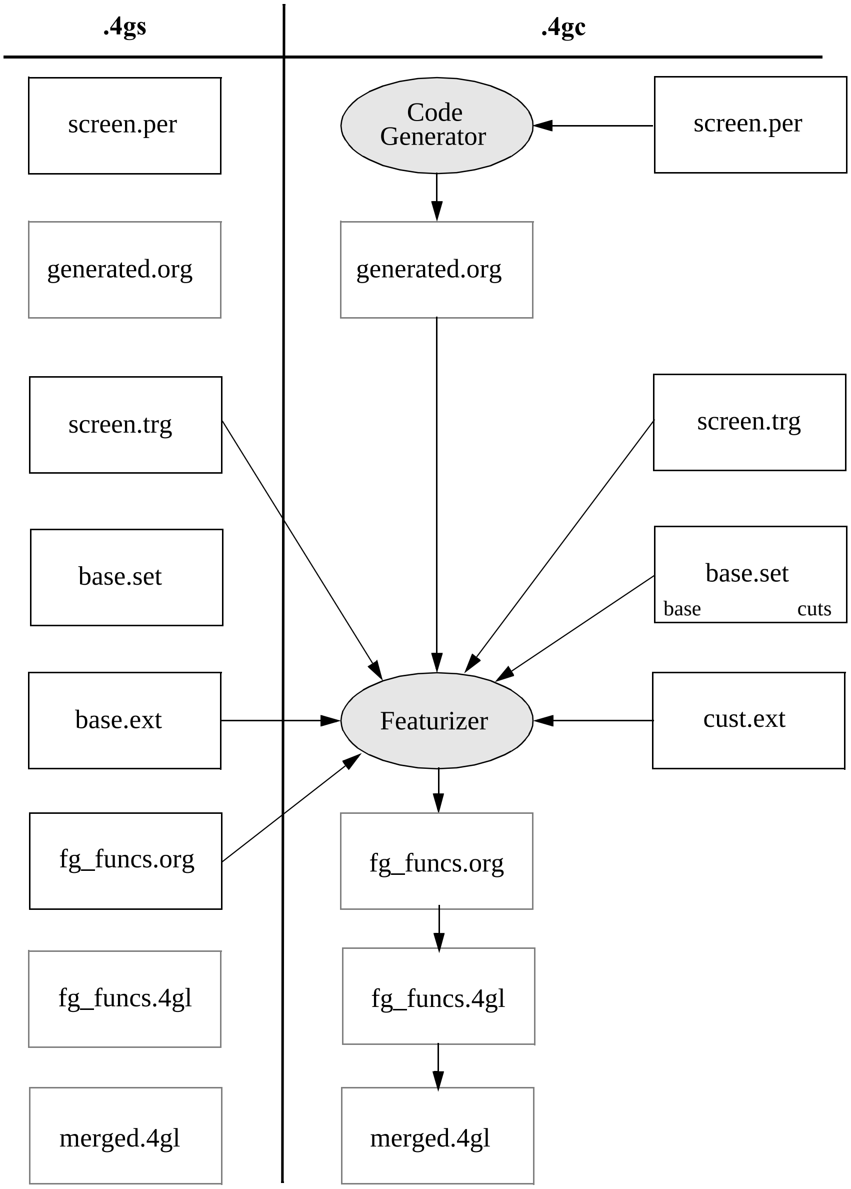

.4gs
.4gc
screen.per
zoom.per
generated.org
merged.4gl
Code Generator
screen.per
generated.org
Featurizer
merged.4gl

A Complex Example Involving .trgs, .exts, custom.4gls, and .per Modifications

This example illustrates what happens when using version control with a heavily customized program. This example illustrates exactly what files are used to build the custom program.

Scenario: A field is added to a copy of the main screen. An `after_field` trigger is added to the screen.trg file. A replace block command is added to the cust.ext file. A `start file "fg_funcs.4gl"` command is added to the cust.ext file.

1. **Create a custom directory.**
2. **Copy the .per forms you want to modify.**
3. **Modify the .per forms.**
4. **Run the Code Generator.**
5. **Create any new triggers and put them in a .trg file in the custom directory.**
6. **Create any new block commands and put them in a .ext file in the custom directory.**
7. **Create a `base.set` file in the custom directory and include the name of all .ext files you want to merge.**
8. **Create a `start file"fg_funcs.4gl"` block command in an .ext file to pull over any custom .4gl files from the .4gs directory.**

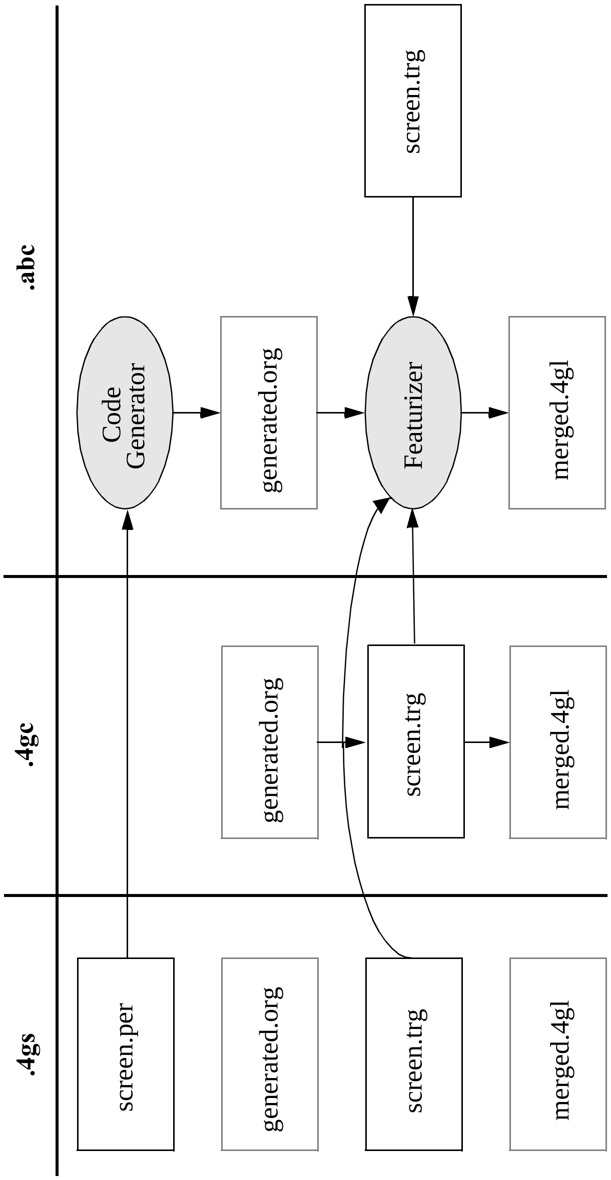

.4gs
.4gc
screen.per
Code Generator
screen.per
generated.org
generated.org
screen.trg
screen.trg
base.set
base.set
base
cuts
base.ext
Featurizer
cust.ext
fg_funcs.org
fg_funcs.org
fg_funcs.4gl
fg_funcs.4gl
merged.4gl
merged.4gl

Using Version Control with Three Directories

This example shows how you can use version control from three or more directories. The most common application of this type of structure occurs when a value-added program needs to be customized for a particular customer.

In this example, the value-added product consists only of a new trigger file. To build a version controlled custom program, various parts are used from both the .4gs and the .4gc directory. These are combined with the contents of the .abc directory to create a custom program.

1. **For this example assume that you have already completed your base (.4gs) and value-added(.4gc) programs.**
2. **Create a new custom directory to hold the customer specific version (.abc).**
3. **Create a new screen.trg file and add your new triggers.**
4. **Run the Code Generator and create the custom program.**


.4gs
screen.per
generated.org
screen.trg
merged.4gl
.4gc
generated.org
screen.trg
merged.4gl
.abc
Code Generator
generated.org
Featurizer
merged.4gl
screen.trg

Advanced Example: Multiple Modifications using Multiple Directories

This example demonstrates how version control works with more than two directories. For this example, the order.per form was modified, new triggers were added for the main screen, and a new .ext file was created to hold some new block commands. Also the `base.set` file was copied over to the .abc directory and the new ord.ext file was added to the file.

1. **For this example assume that you have already completed your base (.4gs) and value-added(.4gc) programs.**

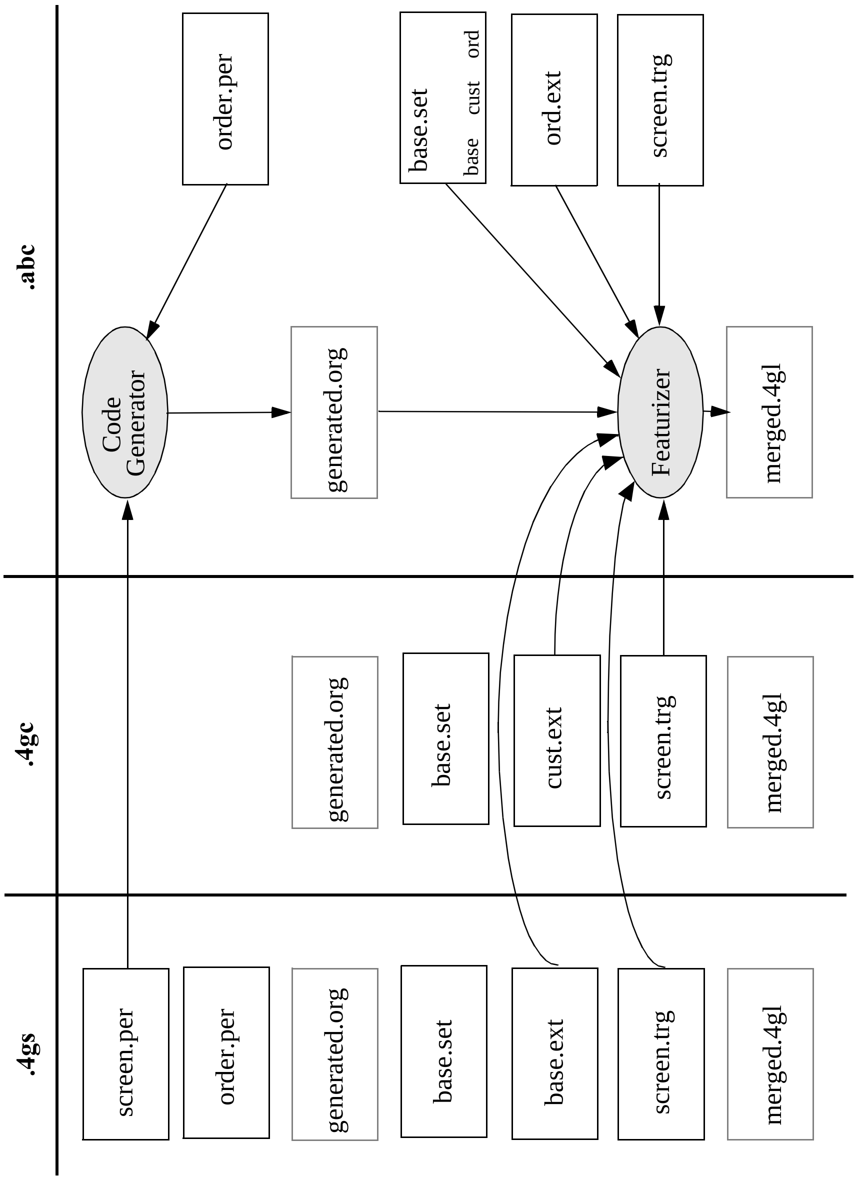
2. **Create a new custom directory to hold the customer specific version (.abc).**

3. **Copy the order.per form from the .4gs directory and modify it.**

4. **Create a new screen.trg and add your new triggers.**

5. **Create a `ord.ext` and add your new block commands.**

6. **Copy the `base.set` file from the .4gc directory and make sure that it contains the name of every .ext file you want to merge in your custom version of the program.**


.abc
.4gc
.4gs
Code Generator
order.per
generated.org
base.set
base
cust
ord
ord.ext
screen.trg
Featurizer
merged.4gl
generated.org
base.set
cust.ext
screen.trg
merged.4gl
screen.per
order.per
generated.org
base.set
base.ext
screen.trg
merged.4gl

17

Language Translation

This chapter covers:

- Creating language independent programs
- Translating .per forms
- Translating values used in data entry
- Translating database strings

About Language Translation

Because of their modifiable nature, Fitrix *Screen* generated applications are easy to translate into other languages. Application translation takes place in four basic areas:

1. Create Language Independent Programs

In order to make translation as easy as possible, all of your programs should reference strings in the database. Do not hard code messages or any type of displayed text. You should keep all displayed text as language independent as possible. The `fg.mssgr` program helps you create language independent programs.

2. Translate .per forms.

The only thing that needs to be translated in your .per forms are the field labels and comments. Once you have your translated .per forms you need to create a subdirectory in your program directory to hold the translated .per forms. This subdirectory should be named after the language the forms are translated into. For example an SPA subdirectory could contain .per forms translated into Spanish.

3. Translate values used in data entry.

Another step involved with language translation is translating the actual data that a user sees and/or enter. Since the underlying code is written in English and uses certain characters and words such as Y/N, it may become necessary at times to display these values to the user in whatever language they are using. For example, if someone is running a French version of an application, they might want to see O/N instead of Y/N. This allows them to enter O which is really stored as Y so the program can understand it.

4. Translate database strings.

Database strings include all of the textual messages that display to the screen, such as error messages and help text. Several programs have been created to allow easy translation of these strings.

Creating Language Independent Programs

All application message strings, error messages, and warning messages should be stored in your database. This makes your programs very modifiable, and easy to switch from one language to another.

The `fg.mssgr` tool should be used when you are creating custom code. Instead of using a constant string to display messages to your users, use variables so your code can remain easily modifiable and language independent.

The `fg.mssgr` tool lets you use language independent programming techniques, yet not have to take time out to do the steps involved in setting up a string variable and value. The `fg.mssgr` tool does these steps:

1. Creates an element in a global record to hold the string value.
2. Adds a call to an initialization function to initialize the string.
3. Inserts the new string into the database.
4. Adds an entry to an unload file to allow for maintenance of the message string.

The `fg.mssgr` tool is intended to run as you are programming. You can call this program from "vi" or from the Form Painter with [CTRL]-[O] as you are creating a trigger or block that uses some new message string. For example, if you are writing a trigger that loads a picker and you need a "title" string for the picker, you can issue the command:

```
fg.mssgr "Choose an Item"
```

or

```
fg.mssgr -q -r chs_item "Choose an Item"
```

To use a string in your code, run `fg.mssgr` and give it the character string to use. You also have optional arguments for finer control.

You then automatically have a str.{string} record variable that you can use in the code immediately. This method is *almost* as easy as using a constant string.

The `fg.mssgr` script adds the message to the `stxmssgr` table, and creates a local `stxmssgr.unl` message unload file. It also makes a . set, and .ext file to build the global and init logic.

The syntax for `fg.mssgr` is as follows:

```
fg.mssgr [-q] [-b] [-dbname database] [-l language]
[-m module_key] [-p program_key] [-n number_key]
[-r record_name] [-x max_length] "message string"
```

-q Suppresses output (quiet mode). Defaults to not quiet.

-b Causes block extension code not to be created. The `-b` flag is used to prepare unload data only. This may be useful in programs where all strings are consolidated in globals.4gl.

-dbname *database* Specifies the database name to use. Defaults to `standard`.

-l *language* Specifies the three character language key. Defaults to ENG.

-m *module_key* Specifies the module name. Defaults to the basename of the parent directory minus the suffix.

-p *program_key* Specifies the program name. Defaults to the basename of the current directory minus the suffix.

-n *number_key* If not given, the number key is taken as one more than the last number key of messages with common language, module, and program keys in the local `stxmssgr.unl` unload file. If there is no stxmssgr.unl file or no messages with a common key, then the number is started at 1. Default is taken from local `stxmssgr.unl` file.

-r *record_name* Names the str record variable to use for the message.

-x *max_length* Sets the maximum length of the string. Default is determined by doubling the string length up to a maximum of 80 characters.

"***message string***"	This is the string being created. The string should be quoted. This argument is required.

Note

`$message_module` and `$message_program` can be used to override the module and program defaults.

Keep in mind that the record element name can be accidentally duplicated in block extension code. The message keys determine if the message is already defined. This approach prevents accidental replacement of valid strings in the `str` record and forces the record to reflect the strings in the database. This sort of error is detected immediately upon compilation and is much easier to detect and correct than an error of accidental omission.

Typing the following command:

```
fg.mssgr -dbname standard -l ENG -m screen -p painter -r scren "This is the
new string."
```

produces this SQL statement:

```
database standard;
delete from stxmssgr where language="ENG" and
mssg_module="screen" and mssg_program="painter" and
mssg_number=1;
insert into stxmssgr values("ENG","screen","painter", 1, "This is the new
string");

Is this okay (y)?
```

Press Y to accept, then you’ll see:

```
Database selected.

0 row(s) deleted.


1 row(s) inserted
```

The `base.set` file looks like this:

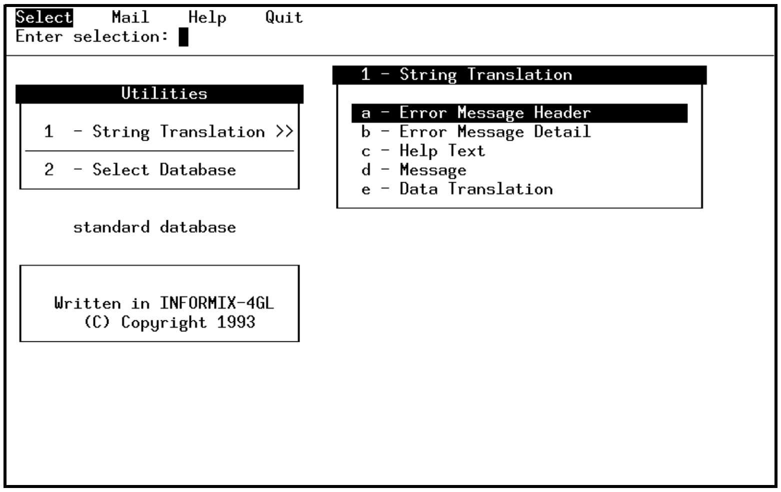
```
str
```

The `str.ext` file looks like this:

```
######################################################################
# Copyright (C) 1992 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%
######################################################################

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

in block TOF NUL after "define"

#_strings_record - Record of constant strings
str record
 scren char(10) # 1 This is the new string
end record,;


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

before block main after_init

#_str_init - Call the string initialization function
call str_init()
;

at_eof_main

######################################################################
function str_init()
######################################################################
# This function initializes all of the static strings
#
 #_init_strings - Get the static strings from the database
 call fg_message("screen", "painter", 1) returning str.scren
 #_end_init_strings

end function
# str_init();
```

Utility Menu

The `fg.tools` program calls a menus interface which provides options for translating database strings into multiple languages. You need Fitrix *Menus* installed on your system in order to use `fg.tools`. If you do not have Fitrix *Menus* installed, then you need to go to the program directory for the various programs and run them individually. The following is the String Translation menu on the Utility menu.



```
Select    Mail    Help    Quit
Enter selection: █

                                  1 - String Translation
          Utilities
                                   a - Error Message Header
  1  - String Translation >>       b - Error Message Detail
                                   c - Help Text
  2  - Select Database             d - Message
                                   e - Data Translation

      standard database


    Written in INFORMIX-4GL
       (C) Copyright 1993
```

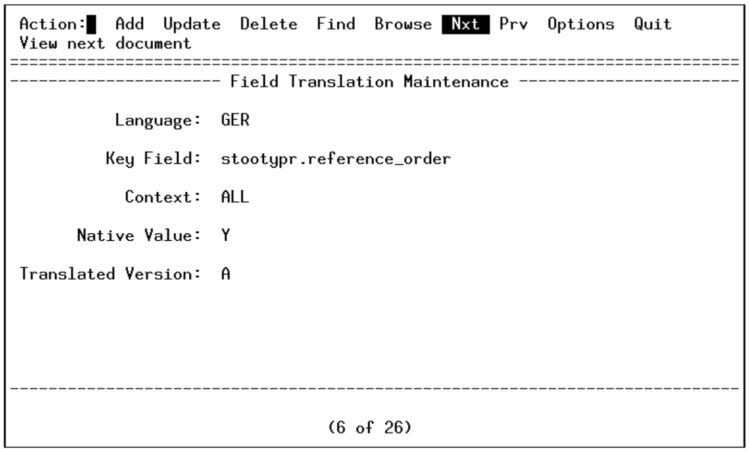
Translating Values Used in Data Entry

Several functions allow for language translation. Although the field labels on .per forms still need to be translated individually and used independently from each other, you can now use and store data in your native language regardless of the language the program is running under. For example, you can have a French version of a perform, which is basically the same as the English version of the .per form, with different field labels and comments. By specifying certain fields as translate fields, and making corresponding entries into the `stxlangr` table for those fields, your users can view and enter data into the translated fields in whichever language is specified when running the program. If the user specifies French, then data is displayed in French. These translation functions are nice because English equivalent (or whatever you use as your native language) gets stored in the database. Say the French user adds a new order in French. Another user on the same system can easily pull up the exact same order and read the translated fields IN ENGLISH.

Although any field can be translated, normally you only translate fields like Y/N, debit/credit and other simple fields like these that display only specific values.

To use language translation, you first need to enter the foreign equivalents to the native language into the `stxlangr` table. This can be done using the Data Translation option on the String Translation Menu. If you do not have Fitrix *Menus* installed, you can invoke this program by running `$fg/codegen/utility.4gm/i_tlangr.4gs/i_tlangr.4g[i|e]`. You can also use ISQL to load the `stxlangr` table.

The Data Translation program:



```
Action:█ Add  Update  Delete  Find  Browse  Nxt  Prv  Options  Quit
View next document
=============================================================================
---------------------- Field Translation Maintenance -----------------------

          Language:  GER

         Key Field:  stootypr.reference_order

           Context:  ALL

      Native Value:  Y

Translated Version:  A





-----------------------------------------------------------------------------

                                 (6 of 26)
```

The next step in translating data is to modify your perform files and specify which fields you want to translate.

Translating the Native Language

For each translated field there must be an entry in the table `stxlangr`. This is a table that holds the native string and the corresponding foreign string. Below is the schema of the `stxlangr` table and a sample row in unload format.

Table schema: `stxlangr`

`language char(3)`	foreign language
`tr_tab_col char(37)`	table.column of the screen field
`tr_context char(10)`	context (default is ALL)
`native char(50)`	native language equivalent
`non_native char(50)`	foreign language equivalent

Sample unload file:

```
GRM|strcustr.gross_entry|ALL|Y|J|
```

The `tr_context` column contains the context that determines where this particular translated value should be displayed. Context allows you to create multiple translations for the same native word, then specify which translation you want to display in any particular field.

For example, say you have two different screens where the English word "debit" is displayed, General Ledger and Order Entry. In your foreign language you might have two different words (and meanings) for the English word debit. On one form you want to display "fred," and on the other form you want to display "larry." What you would do is create the following entries in your stxlangr table:

```
GRM|strcblah.debit_credit|ALL|debit|fred|
GRM|strcblah.debit_credit|oe|debit|larry|
```

Your .per forms would contain the following translated fields.

Order Entry .per form:

```
translate=  debit_credit oe
```

General Ledger .per form:

```
translate=  debit_credit ALL
```

The context column can contain any value, uppercase or lowercase. Specify "ALL" if the foreign value displays the same string in all occurrences of the native value.

Specifying Translation in the .per File

Once all of the values have been translated in the stxlangr table, you can modify your .per forms. The Screen Code Generator generates translation logic only for those fields on your .per forms you have defined as translate fields.

Translate fields can be defined in the Form Painter via the Define Field form.

All that is required is to list the names of the columns you want to translate. Note that the "context" is usually ALL. Logic is generated so that when the program is run with a specific language, those fields defined as translate fields display values in the language specified.

The following line shows how to define translated fields in the .per file:

```
translate = col1 context, col2 context, col3 context
```

Creating Directories for Translated .per Forms

To properly take advantage of Fitrix *Screen*'s language handling techniques, you need to locate all of your .per forms in subdirectories named after the language key. For example, if you have translated the i_custr program into both French and Spanish, then under i_custr.4gs you should create a FRN directory to contain the French .per forms, an SPA directory to contain your Spanish forms, and a ENG directory to store your original English forms. When using translation, no forms should appear in the program directory itself, or those forms will always be used regardless of what the $language variable is set to.

When invoking the program, Fitrix *Menus* knows to find the proper forms based on the $language global variable. The subdirectory matching the contents of the $language variable is added to the $DBPATH.

Alternate forms can also be displayed by invoking the program and passing a -l *language* on the command line. However, you must manually add the correct forms subdirectory to your $DBPATH if the program is run from the command line.

Modified Functions

Several functions allow for language translation. All of the display functions contain calls to two functions, string_to_foreign() and string_to_native(), for each translated field. These functions display the data in the foreign language while retaining the data in the program variables in the native language. For instance when running a program in German, with the native language as English, a J might appear on the screen representing Yes, but the program variable still contains the English equivalent Y. In other words, what you see is not what you get. This allows the program to continue to make logic decisions based on native language strings such as Y and N.

If there are translated fields on the screen, a section that transforms the construct statement is generated below the normal construct statement. The function that transforms the constructed statement is `string_construct()`. Following is an example of the code.

```
#_translate - Change construct for language independence if needed
 if get_scrlib("language") != "ENG"
 then
     #_num_trans - Send the number of fields to be translated
     call put_vararg(3)

     #_send_construct - Send the constructed string
     call put_vararg(scratch[1,512])

     #_like_type - Send translated field and it’s context
     call put_vararg("stootypr.like_type")
     call put_vararg("ALL")
     #_master_order - Send translated field and it’s context
     call put_vararg("stootypr.master_order")
     call put_vararg("ALL")

     #_trans_construct - Rebuild construct for language independence
     call string_construct()
     let scratch = get_vararg()
     let num_trans = get_vararg()
     if num_trans > 0
       then let is_translated = "translated"
       else let is_translated = null
     end if
 end if
```

The translation functions are found in `$fg/lib/standard.4gs/l_trans.4gl`.

Translating Values and Database Strings

The successful creation of language independent programs relies on storing strings of text in the database. You can then create translations of each string and tell the program to use the strings defined for a particular language.

Error messages can now be easily translated into different languages. Two utility interfaces allow you to display the error message to be translated in your native language, while allowing you to enter the translated text. Both of these programs are options on a utility menu discussed next. This utility menu is displayed by typing `fg.tools`. See "Utility Menu" on page 17-7.

The first program, Error Message Header (`i_terorh`), is used to translate the one line error messages that appear when an error is first encountered. The second program, Error Message Detail (`i_terord`), is used to translate the error message detail text. Error message detail text is separated into both problem text and solution text.

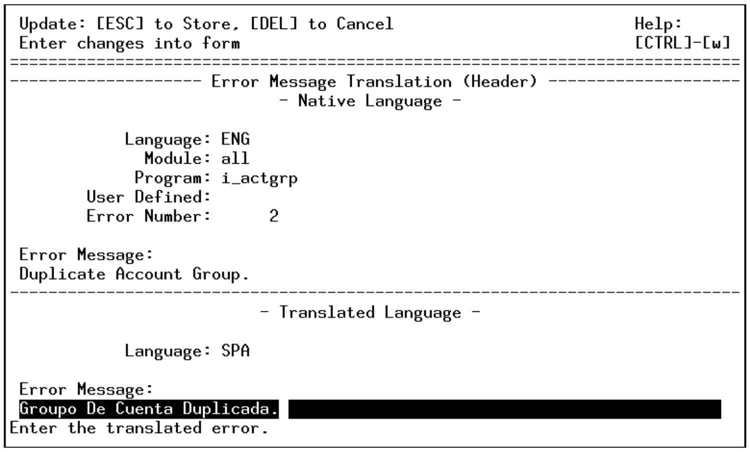
Remember when using any of these translation programs, data is put directly into your database. If you need to move the translated information to other databases then you need to create unload files to dump the data created with these programs.

Translating the Error Message Header

The Error Message Header is the one line message that appears when an error is encountered. To translate the error header message into different languages run `fg.tools` and select the Error Message Header option on the String Translation menu. If you do not have Fitrix *Menus*, run the `i_terorh` program in `$fg/codegen/utility.4gm`.

Find the message you want to translate in whatever native language you choose. The native language is usually English. Specify the language you want to translate into, then enter the translated text.

The Error Message Header translation program.



```
 Update: [ESC] to Store, [DEL] to Cancel                        Help:
 Enter changes into form                                         [CTRL]-[w]
============================================================================
-------------------- Error Message Translation (Header) --------------------
                             - Native Language -

            Language: ENG
              Module: all
             Program: i_actgrp
        User Defined:
        Error Number:      2

 Error Message:
 Duplicate Account Group.
----------------------------------------------------------------------------
                           - Translated Language -

            Language: SPA

 Error Message:
 Groupo De Cuenta Duplicada.
Enter the translated error.
```

The following steps describe how to create error header messages.

1. **Find the native error message.**

   The first step is to find the native message you want to translate. Execute the Find command on the ring menu and then enter the key information that identifies the particular message you want to translate. All of the fields on the top part of the screen make up the key for error message headers (language, module, program, user defined, and error number.)

2. **Select the Update command.**

3. **Enter the foreign language to translate into.**

   After selecting Update, the cursor is put into the translated language field. In this field enter the language you want to translate the message into. If you want to create a Spanish translation, enter SPA.

   After entering a value into the translated language field for the first time, a global variable is set so that when you update the next message the value that you entered before automatically appears. This is so you do not have to keep entering the same information over and over if you are sitting down and trying to translate all of the messages at once. Once a language value appears in this field, press [ENTER] to get to the message line. You can change this value at any time by entering a new language. This is a 3 character field.

4. **Enter the translated text.**

   The Error Message field is a 40 character field that contains the translated error message. If text exists for the translated language and key, it is automatically displayed here. Text can be modified by typing over it.

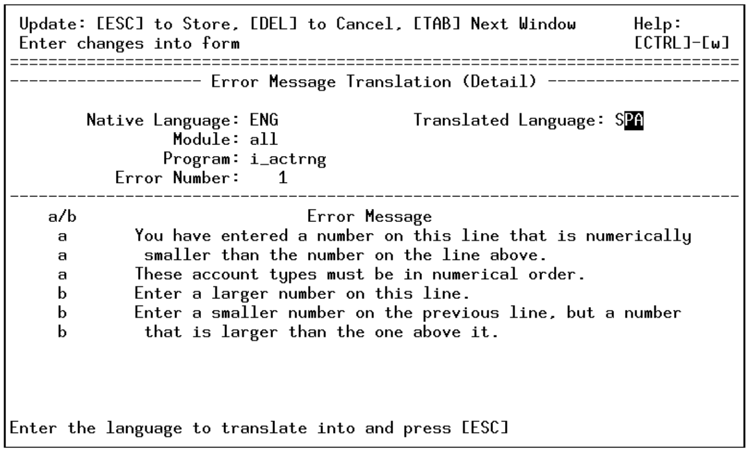
5. **Save the new record.**

Translating Error Message Detail

If you have Fitrix *Menus*, running `fg.tools` and selecting the Error Message Detail option on the String Translation menu allows you to translate the problem and solution detail text of an error message. If you do not have Fitrix *Menus* you can run the executable in `$fg/codegen/utility.4gm/i_terord.4gs/i_terord.4g[i|e]`. Error message detail is the text that is displayed when you press [CTRL]-[z] to get more information about an error after it appears.

The functionality of the `i_terord` program is slightly different from the error message header program. This program consists of two forms. The first form (the main screen) allows you to find the native error text you want to translate and displays the text on the bottom of the screen.

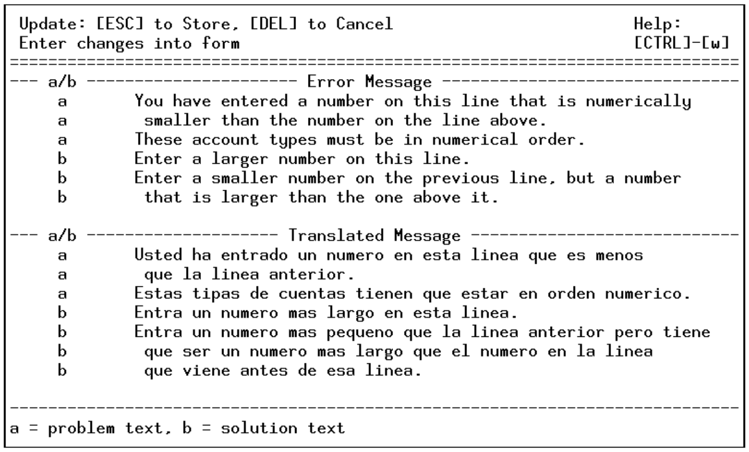
Main screen:



```
 Update: [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
 Enter changes into form                                         [CTRL]-[w]
===========================================================================
-------------------- Error Message Translation (Detail) --------------------

       Native Language: ENG             Translated Language: SPA
                Module: all
               Program: i_actrng
          Error Number:    1
---------------------------------------------------------------------------
    a/b                         Error Message
     a       You have entered a number on this line that is numerically
     a        smaller than the number on the line above.
     a       These account types must be in numerical order.
     b       Enter a larger number on this line.
     b       Enter a smaller number on the previous line, but a number
     b        that is larger than the one above it.




Enter the language to translate into and press [ESC]
```

Like the other error message programs, once you locate the native message you want to translate, you Update the form. Updating the document puts you in the Translated Language field. In this field you enter the language to translate into. Pressing [ESC] calls up the detail form, which displays the native and the translated error text together.

Detail screen:



```
 Update: [ESC] to Store, [DEL] to Cancel                        Help:
 Enter changes into form                                        [CTRL]-[w]
============================================================================
--- a/b ---------------------- Error Message ------------------------------
     a       You have entered a number on this line that is numerically
     a        smaller than the number on the line above.
     a       These account types must be in numerical order.
     b       Enter a larger number on this line.
     b       Enter a smaller number on the previous line, but a number
     b        that is larger than the one above it.

--- a/b -------------------- Translated Message ----------------------------
     a       Usted ha entrado un numero en esta linea que es menos
     a        que la linea anterior.
     a       Estas tipas de cuentas tienen que estar en orden numerico.
     b       Entra un numero mas largo en esta linea.
     b       Entra un numero mas pequeno que la linea anterior pero tiene
     b        que ser un numero mas largo que el numero en la linea
     b        que viene antes de esa linea.

----------------------------------------------------------------------------
a = problem text, b = solution text
```

You can move between the two detail sections by pressing [TAB]. This allows you to use the arrow keys to scroll the native text. The bottom section contains the translated text. Pressing [ESC] saves the translated error text.

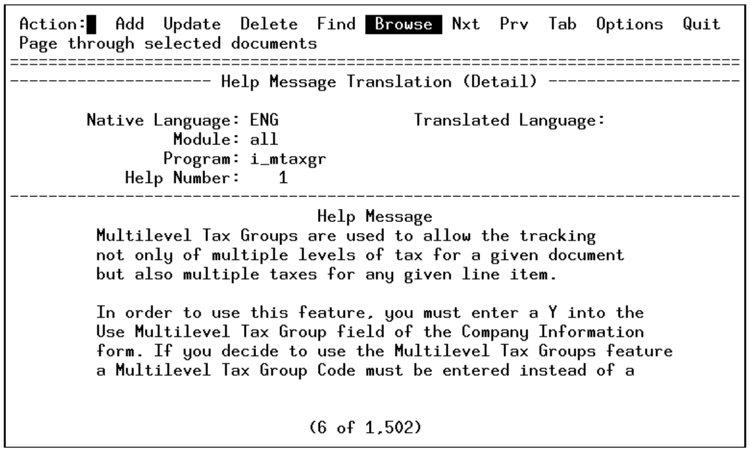
You should be aware that error messages in the `stxerord` (the table this program modifies) and `stxerorh` tables are related to each other. This means that if you translate detail errors, then you need to make sure to translate the header errors as well. A special check has been put into the Error Message Detail program. If you translate an error message detail that does not have an existing entry in the `stxerorh` table for the same key, a warning box appears notifying you of this. You can then type in the corresponding header message into a field in the box if you wish. Or you can simply press [ENTER] and then define the header portion later.

Translating Help Text

If you have Fitrix *Menus*, running `fg.tools` and selecting the Help Text option on the String Translation menu allows you to translate on-line help text for your programs. If you do not have Fitrix *Menus* you can run the executable in `$fg/codegen/utility.4gm/i_thelpd.4gs/i_thelpd.4g[i|e]`. On-line help text is displayed when you press [CTRL]-[z] to get more information about a field in a program.

The `i_thelpd` program consists of two forms. The first form (the main screen) allows you to find the native help text you want to translate and displays the text on the bottom of the screen.

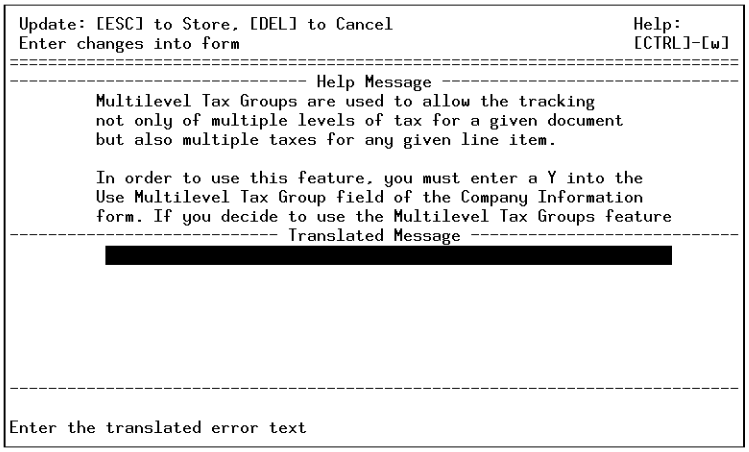
Main screen:



```
Action:█ Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Page through selected documents
===========================================================================
--------------------- Help Message Translation (Detail) --------------------

       Native Language: ENG              Translated Language:
                Module: all
               Program: i_mtaxgr
           Help Number:    1
---------------------------------------------------------------------------
                              Help Message
        Multilevel Tax Groups are used to allow the tracking
        not only of multiple levels of tax for a given document
        but also multiple taxes for any given line item.

        In order to use this feature, you must enter a Y into the
        Use Multilevel Tax Group field of the Company Information
        form. If you decide to use the Multilevel Tax Groups feature
        a Multilevel Tax Group Code must be entered instead of a


                             (6 of 1,502)
```

Like the other error message programs, once you locate the native message you want to translate, you Update the form. Updating the document puts you in the Translated Language field. In this field you enter the language to translate into. Pressing [ESC] calls up the detail form, which displays the native and the translated help text together.

Detail screen:

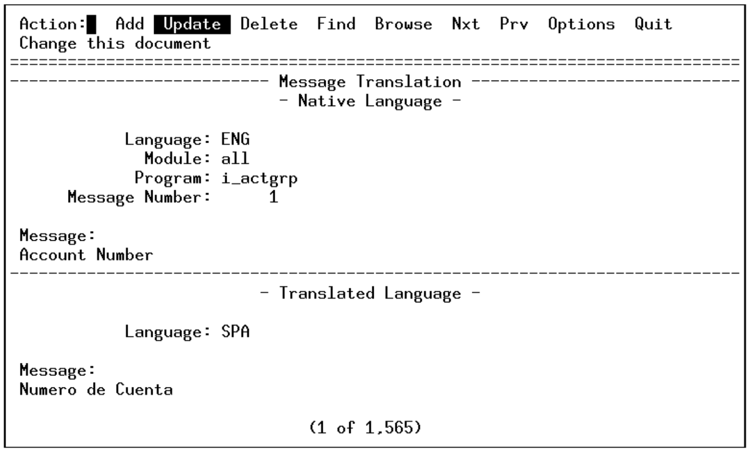


```
 Update: [ESC] to Store, [DEL] to Cancel                        Help:
 Enter changes into form                                        [CTRL]-[w]
============================================================================
------------------------------ Help Message -------------------------------
        Multilevel Tax Groups are used to allow the tracking
        not only of multiple levels of tax for a given document
        but also multiple taxes for any given line item.

        In order to use this feature, you must enter a Y into the
        Use Multilevel Tax Group field of the Company Information
        form. If you decide to use the Multilevel Tax Groups feature
--------------------------- Translated Message ----------------------------








----------------------------------------------------------------------------

Enter the translated error text
```

You can move between the two detail sections by pressing [TAB]. This allows you to use the arrow keys to scroll the native text. The bottom section contains the translated text. Pressing [ESC] saves the translated help text.

Message Translation

The `stxmssgr` table contains various strings of text that get displayed to the screen. To translate a message into different languages run the `fg.tools` program if you have Fitrix *Menus*, and select the Message option on the String Translation menu. If you do not have Fitrix *Menus* then you can run `$fg/codegen/utility.4gm/i_tmssgr.4gs/i_tmssgr.4g[i|e]`. Find the message you want to translate in whatever native language you choose. If the native language is English, then enter "ENG." After the string has been found, Update the form. Specify the language you want to translate into, then enter the translated text.



```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Options  Quit
Change this document
===========================================================================
-------------------------- Message Translation ----------------------------
                           - Native Language -

           Language: ENG
             Module: all
            Program: i_actgrp
     Message Number:      1

Message:
Account Number
---------------------------------------------------------------------------
                         - Translated Language -

           Language: SPA

Message:
Numero de Cuenta

                              (1 of 1,565)
```

The following steps describe how to translate messages.

1. **Find the native message.**

   The first step is to find the message you want to translate. Execute the Find command on the ring menu and then enter the key information that identifies the particular message you want to translate. All of the fields on the top part of the screen make up the key for messages (language, module, program, and message number.)

2. **Select Update.**

3. **Enter the foreign language to translate into.**

   After selecting Update, the cursor is put into the Translated Language field. In this field enter the language you want to translate into. If you want to create a Spanish translation, enter SPA.

   After entering a value into the translated language field for the first time, a global variable is set so that when you update the next message the value that you entered before automatically appears. This is so you do not have to keep entering the same information over and over if you are sitting down and trying to translate all of the messages at once. Once a language value appears in this field, press [ENTER] to get to the message line. You can change this value at any time by entering a new language. This is a 3 character field.

4. **Enter the translated text.**

   The Message field is a 132 character field that contains the translated message. If text exists for the translated language and key, it is automatically displayed here. Text can be modified by typing over it.

5. **Save the new record.**

18

Helpful Techniques

This section explains how to perform a variety of modifications to your programs including:

- Creating field-level help unique to a program
- Creating phony joins
- Centering a window
- Calling screen applications from a screen application
- Capturing the user’s name
- Disabling the "Add a Navigation" menu option
- Disabling the F1 and F2 keys in a screen detail section
- Enabling hot keys in scroll mode of a detail section
- Immediate Zoom without pressing [CTRL]-[z]
- Disabling Function Keys

Creating Field Level Help That is Unique to the Program

Field level help is unique to the table and column. This means that once you define help for a field, the same help appears in every program that uses the same table.column.

Here is one way to create field help unique to the program. Basically you need to add your custom help to the stxhelpd table with ISQL using the normal table and column key and also a new hlp_number other than one. The help number is what makes your custom help unique to a program.

You then need to add some calls to put_vararg to pass the unique values to the help function.

Here is a sample trigger that goes in the trigger file for your main form:

```
-------begin screen.trg----------
input 1 (if the field you want is in the header section of the form)
on_event help
    call put_vararg("hlp_number")
    call put_vararg("2")  # this is the unique hlp_number for this form
    call scr_help()
    ;
on_event help
     if scr_fld = "customer_num"
     then
        call put_vararg("hlp_number")
        call put_vararg("2")  # this is the unique hlp_number for this form
        call scr_help()
     else
     call scr_help()
     end if
     ;
-------end screen.trg----------
```

You then need to create your own help text using the following keys to tie the text to the field on the screen: table, column, and hlp_number. Changing the hlp_number to something other than one makes it unique to this particular program because you are calling it explicitly.

This maintains all of the regular methods of help, including the built in field level help.

For more information on creating help refer to "Creating Application Help" on page 15-22.

Creating Phony Joins

A problem occurs when you want to display information from two tables on the screen that have no real header/detail relationship. For example, the Company Information form common to all Fitrix accounting applications displays company information in the top portion of the screen and department codes in the lower half of the screen. The problem is that there is no real join between the company information and department codes tables.

The solution is to set up a phony join between two columns then to modify a few lines in the code. For the example mentioned above this involves setting up the following `join` line in `input 2` of the .per's FOURGEN section:

```
  join = stxinfor.src_key = stxcntrc.co_name
```

After the code is generated, three places in the logic need to be modified in `detail.4gl`: `lld_m_prep()`, `lld_read()`, and `lld_delete()`. The changes involve simply searching for the `where` clause that match the `join` line and replacing the second half of the clause. However, to maintain regenerability, the following steps need to be taken. First, the modified functions are copied into the `at_eof` section of the `input 2` section of the trigger file. Next, the `do_not_generate` trigger is added for the three above mentioned functions to the default section. The following is an example of a trigger file used to get around this.

```
######################################################################
# Copyright (C) 1991
# Your Company Name
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#) .../all.4gm/i_contrl.4gs/screen1.trg  1.12  Delta: 10/11/91
######################################################################
# Screen Generator version: 4.00.UC1

defaults
    do_not_generate
        lld_delete
        lld_m_prep
        lld_read;
...
    ######################################################################
    function lld_m_prep(n)
    ######################################################################
...
      let m_xinfor.src_type = "D"
...
    ######################################################################
    function lld_read()
    ######################################################################
...
              "where stxinfor.src_type = ’D’"
...
    ######################################################################
    function lld_delete()
    ######################################################################
...
          where stxinfor.src_type = "D"
...
```

Centering a Window

The following section explains an easy way to determine the starting location for a window in order to center it.

This method can be used for any windows other than full screen windows (main entry screen).

Starting screen coordinates in Fitrix *Screen* are given in y, x format, where y=row and x=column. The convention for all windows is to have a "fixed" y (row) at position 5. This way, all windows open up just below the menu line of the ring menu. The x (column) should usually be centered. Here’s a good formula to use to establish the starting location:

```
let x = ((80 - x1) / 2) + 1
```

x is the total column length of the window to be centered. For example, if you have a browse window that has 14 rows and 60 columns. The y is 5 (following convention). To find the conventional starting location for x:

```
let x = ((80 - 60) / 2) + 1   # the result of which is 11
```

Therefore, the proper starting location is 5, 11.

Calling Screen Applications from a Screen Application

This is one way you can modify `options.4gl` to run other programs from the Options ring-menu in your program. Remember, `options.4gl` does get regenerated, so your changes should be placed in a .ext file. The code is added after the menu command similar to what is shown below.

```
    menu "Options"
        command
           "Customer" "Reference the Customer database file"
           run "runcust"
           exit menu

  # runcust syntax: (runcust located in the program directory)
  # don’t forget to make the edit file executable (ex: chmod 770 runcust)
  # you can of course pass in filters to the fglgo command.
  #
  # cd ../i_custr.4gs
  # fglgo i_custr.4gi
```

Another way to perform this task is to define a char variable in `options.4gl`, set the variable to the command you want to run, then run that variable as in the example below:

```
xx char(40) # must be large enough to accept the string
let xx = "cd ../i_custr.4gs; fglgo i_custr.4gi"
    menu "Options"
        command "Customer" "Reference the Customer database file"
            run xx
            exit menu
```

Capturing the User’s Name

The function fg_username returns the user’s name. It is located in the $fg/lib/standard.4gs library directory. You can call it like this:

```
let usr_name = fg_username()
```

The fg_username function uses systables to obtain the user’s name. If no name is found, the value "UNKNOWN" is returned.

Disabling the "Add a Navigation" Menu Option

You can disable the "Add a Navigation Action" menu selection to disallow users from adding an event.

To make this work, you need to copy $fg/lib/standard.4gs/lib_text.4gl into your local directory. Make the modifications outlined below to lib_text.4gl, remove the .4go’s and recompile your program. Run the program, press [CTRL]-[g], and notice that neither an accept key, [ENTER] or [CTRL]-[z] allows you to add a navigation action. All you see is a series of asterisks.

```
# This is $fg/lib/standard.4gs/lib_text.4gl
#######################################################################
function textput(_text)
#######################################################################
# This function puts the passed text into the next element of the
# txt array.
#
 define
   _text char(74), # passed text
   n smallint # generic number

 # Don’t allow recursion
 if is_open = "Y" then return end if

 # Initialize the array
 if reset_cnt = 123
 then else
    call textinit()
 end if
 # Substitute the string to be blocked with a known string
 # to test for in textpick()

 # Circumvent select navigation event events
a->> if _text = "Add a navigation action"
a->> then
a->> let _text = "***********************"
a->> end if

 let arr_cnt = arr_cnt + 1
 let arr_cur = arr_cnt
 if arr_cnt > 150
    then
    call lib_error("lib_scr","textzm",1,"")
 return
 end if

 # Set text if passed in scratch
 if _text = "(see scratch)" then let _text = scratch end if

 # Bump up max_col if necessary
 let n = length(_text)
 if n > max_col then let max_col = n end if
 # Set text into array
 let txt[arr_cur]._text = _text

end function
# textput()

#######################################################################
function textpick(head)
# returning the line number of the picked element or 0 if [DEL] pressed.
# also returning picked data in scratch if selected.
# returns 0 and scratch[1,4]="zoom" and scratch[5,74] zoom line text
# (if you need the line # for zoom, call arr_curr())
#######################################################################
# This function places the user into display array on the txt[] array.
# It is designed as a ‘picker’ to select from a list.
#
 define
a->> j smallint, # arr_curr()
 head char(80) # screen heading (passed)

 # Don’t allow recursion
 if is_open = "Y"
 then
     call lib_error("lib_scr","recursiv",1,
     "Window: Generic text picker/editor")
     return false
 end if

 # Assign the static ‘heading’ variable
```

```
     let heading = head

     # Bump up max_col if heading is bigger than max_col
     if max_col < length(heading)
     then let max_col = length(heading)
     end if
     # Call textshow (no update - view only)
     if textshow(false) = false
     then return 0
     else
     # Test for known string and return 0 indicating nothing
     # was selected if a disabled item was selected
    a->> let j = arr_curr()
    a->> if txt[j]._text[1,4] = "****"
    a->> then
    a->>   return 0
     a->> end if
     return arr_curr()
     end if

    end function
    # textpick()
```

This modification must be made to disable the ability of a user to Zoom from "Add a Navigation Action" on the Navigate picker and put this map to a hot key. About line 408 in `lib_text.4gl`:

```
    # Send the zoom down to the picker’s calling function
    let arr_cur = arr_curr()
    a-->if txt[arr_cur]._text[1,4] = "****"
    a-->then
    a-->    let scratch = txt[arr_cur]._text
    a-->else
            let scratch = "zoom", txt[arr_cur]._text
    a-->end if
    return false
```

Disabling the F1 and F2 Keys in a Screen Detail Section

You can disable the F1 and F2 keys in the detail section of a Screen Generated application by using the Informix options command to re-map these two keys. Refer to the INFORMIX-4GL Reference Manuals for specific information on the options command. Place the options commands in a `before_input` trigger in the input 2 section of your .trg file.

Example:

```
before_input
    options insert key F36
    options delete key F36
    ;
```

Notice that in this example, insert and delete are re-mapped to the same function key, F36, which is not even accessible from most keyboards.

If you have problems re-mapping a key to F36, another safe alternative is to re-map insert and delete to "control-s". For example:

```
before_input
    options insert key control-s
    options delete key control-s
    ;
```

Using Triggers to Disable Function Keys

Function keys can be easily disabled using triggers. For example, you may have an input screen that you want to disable the [TAB] and [DELETE] keys for. All you have to do is create a .trg file with the same name as the .per file, such as: `screen1.trg` if the .per is named `screen1.per`.

Then add the following lines to the `input 1` section of the .trg file:

```
input 1
 on_event tab
    exit case
    ;
```

The final step is to regenerate your code.

You can also easily disable the [F1] and [F2] function keys by re-mapping them to something that is unlikely to be defined, such as F36. That is, most termcap entries do not have a definition for F36, so reassigning insert and delete to F36 (yes, both!) in `after_init` puts the disabling logic in main, and effectively disallows usage on a program level.

Example:

```
after_init
         options insert key f36
         options delete key f36;
```

Creating a Verification Prompt for Deletions

Here is an example of how to create a prompt for verification before executing the "Delete" ring menu option.

Let’s say that an `ok_delete` function is placed into the source code for a customer input screen. When the user tries to delete a customer with the "Delete" ring menu option, `ok_delete` pulls up a warning and asks the user to verify deletion of the record:

```
##################################################################
function ok_delete()
# returning true or false based upon ok to delete
##################################################################
#
    define
        prompt_response char(1)


    open window delete_rec at 14,16 with 9 rows, 51 columns
        attribute (border, blue, prompt line last)

    display STR.delete1 at 1,1 attribute (reverse)
    display "===================================================" at 2,1
    display STR.delete2 at 3,1
    display STR.delete3 at 4,1
    display STR.delete4 at 5,1
    display "===================================================" at 8,1


    let prompt_response = null
    while prompt_response matches STR.not_yes_no or
          prompt_response is null
        prompt STR.ok_continue for prompt_response
    end while
    if upshift(prompt_response) = STR.n_response
    then
      close window delete_rec
      return false
    end if

    close window delete_rec
    return true

end function
# ok_delete()
```

Immediate Zoom Without Pressing [CTRL]-[z]

It is possible for a user to Zoom immediately upon entering a particular field. This arrangement might be appropriate if, for example, the shipping address for a specific order must be selected from among several possible ship-to addresses.

Add a trigger similar to the following before the field with a zoom.

```
input 1
  before_field customer_num
    if p_orders.customer_num is null
    then
         call fgStack_push("")
         call socketManager("cust_zm","zoom", "default")
         let p_orders.customer_num = fgStack_pop()
         if p_orders.customer_num is null
           then let nxt_fld = "customer_num" end if
    end if
    ;
```

Adding Cursor Scrolling in Detail/Add-on Detail Screens

The following is an .ext you can plug into your detail or add-on detail screens to allow you to control your cursor scrolling.

```
######################################################################
# Copyright (C) 1992 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%
######################################################################
#
# The following logic will allow you to programmatically control
# field and row positioning within the standard detail array, and
# should also work for your add-on detail as well.
#
# What is required is that you have a ’hidden’ no-entry field as the
# -last- field in the detail line. While some see this need for a
# dummy field as a disadvantage, others would need this input array
# control will make the dummy field. However, if the current last
# field happens to already be a no-entry field, than you will not
# need to add the dummy field. The bottom is that the last field
# needs to be a no-entry field. Replace the name of this existing/
# new no-entry field with the "no_entry_field" string found below in
# the before_row block code. Then, all you need to add is the
# field/row validation that you need, setting goto_row and goto_fld
# to where you want the cursor to be placed:
#
#   let goto_row = 5
#   let goto_fld = "field_name"
#
# Remember to change the filename name in the 'start file' command.
######################################################################

start file "detail.4gl"

  static_define
    #_scroll_variables - are assigned the destination to scroll to
    goto_row smallint,        # The array row to go to
    goto_fld char(18)         # Field to go to on goto_row;

  before_input
    #_init - initialize variables
    let goto_row = null
    let goto_fld = null;

  before_row
    #_scroll_control - scrolling logic management
    if goto_row is not null
    then
        case
          when p_cur < goto_row # Go to the next row
            let nxt_fld = "no_entry_fld"
            return
          when p_cur > goto_row # Go to the top
            let nxt_fld = "goto top"
            return
          when p_cur = goto_row # We've made it.
            let nxt_fld  = goto_fld
            let goto_row = null
            let goto_fld = null
        end case
    end if;

  before block lld_a_row after_row
    #_scroll_fall_through -  don't validate row data if in 'scroll' mode
    if goto_row is not null
    then
        let nxt_fld = null
        return
    end if;
```

Part Four

Appendixes

A

Fitrix *Screen* Utilities

This appendix discusses several useful utilities included with Fitrix *Screen* along with other miscellaneous information such as:

- The `scr_demo` script
- The tags utility
- A script that lists all functions (with descriptions) in a program
- Adding stores demo tables to a database (`mkdemo`)
- Creating a demonstration database (`fg.demodb`)
- Viewing database table descriptions
- Using `imap`
- Adding Code Generator tables with `mktables`
- Using `fg.setshell`

The Demo Script (`scr_demo`)

A number of screen demo applications are included with the Code Generator. The screen demos are essential to learning Fitrix *Screen*. The screen demos:

- provide real working examples illustrating various capabilities of Fitrix *Screen*.
- show how browse forms, Zooms, and lookups work.
- show how an add-on header is hooked into your main form.
- show how triggers are automatically merged into your .4gl files and how any line of code can be modified via block commands.

Each demo application has a corresponding .bak directory, which contains the original .per form specification files.

There are two ways to run a screen demo.

1. You can use the `scr_demo` script. This script automatically sets up your environment and prepares a fresh directory with new .per forms.
2. Or you can do the work of the `scr_demo` script by hand. After your environment is set correctly `($fg, $PATH, $DBPATH, and $INFORMIXDIR)`, create a .4gs directory. Next, copy the files from the `$fg/codegen/demo.4gm/screen*.bak` directory to your .4gs directory (where * is the number of the demo you wish to run). Then generate, compile, and run the demo.

In order to use the `scr_demo` script, invoke it from a UNIX prompt as follows:

```
scr_demo {1|2|3|5|6|7|8|9}
```

The script must be invoked with one argument, to specify the particular demo to run. The following demos are available:

1. Header only screen demo.
2. Header/detail screen demo.
3. Header/detail demo with Zoom, lookup, math, etc.
4. Same as 3 but this directory is used with the Form Painter.

5. Header/detail complex demo with triggers.
6. Header/detail that demonstrates triggers and Version Control.
7. Extension screen demo.
8. Add-on detail demo.
9. View-only, view-detail, and query screen demo.

Once the scr_demo script carries out the preparatory steps, it displays the present working directory and opens a shell for the purpose of running the demo.

Note

Each time the scr_demo script is run, it removes everything in the $fg/codegen/demo.4gm/screen*.4gs directory, then copies the .per files from the parallel .bak directory. This action means that every time you run scr_demo, you wipe out all previous work. If you wish to leave a demo and then return to it later, simply cd directly into the screen*.4gs directory and continue your work. Do not run the scr_demo script if you want to continue a previous demo.

Note

The stores demo tables need to be present in any database you run the screen demos against. You can add these tables by running the mkdemo script discussed next.

Adding `stores` Demo Tables to a Database (`mkdemo`)

The screen demos require tables from Informix's `stores` demo database. Although installing the Code Generator adds these tables to the `standard` database, you can add the `stores` demo tables to any database by running the `mkdemo` script.

Syntax:

```
mkdemo -dbname database
```

Creating a Demonstration Database (`fg.demodb`)

The `fg.demodb` script allows you to create another `stores` database for demonstration purposes. You can create another `stores` database anywhere you want. To create a demo database first change directories to the location you want to create the `stores` database. Type:

```
$fg/bin/fg.demodb
```

You are prompted for a name for the database.

If you are running the OnLine engine, you are prompted for the dbspace to create the database in. For more information on dbspace refer to your INFORMIX-OnLine documentation.

If you are running the Standard engine, dbmerge is run and the database is created in your current directory.

Locating Functions/Displaying Function Descriptions

This section describes some utilities included with Fitrix *Screen* that help you quickly locate and display library functions.

The Tag Utility

During compilation of source code, a database of function calls is created and stored in files named tags. The files are found in the local module directory and at the `$fg/lib` level. These tags files constitute the paths for all local and library functions called by the executable program. These tags are created by two shell scripts:

- `$fg/bin/itags` creates an INFORMIX-4GL `tags` file in the local source directory.
- `$fg/bin/litags` creates a `tags` file for 4GL libraries and merges it into `../tags`

Tags files allow you to benefit from hypertext-style mobility. If you use vi as your text editor, you can set up your system to edit a function simply by pressing one pre-defined key. For example, if the cursor is currently positioned on a word that is a function call, your pre-designated hypertext key takes you to the file that defines the function. A separate key can be set up to take you back to the departure point. The tags feature helps you to examine the source code in a step-by-step fashion.

To benefit from the tags feature, make a few additions to your `$HOME/.exrc` file. Set tags as follows:

```
set tags=tags\ ../tags\ ($fg expanded)/all.4gm/tags\
($fg expanded)/lib/tags
```

The `$fg/lib/tags` points to the tags file in the directory containing the archived library functions. With the above line in your `$HOME/.exrc` file, you can take advantage of the power of hypertext-style mobility with source code.

To automate the process further, to map keys in your `$HOME/.exrc` file.

```
map ] ^]
map [ ^^
```

Note

The characters ^] represent [CTRL]-[ ]]. The characters ^^ represent [CTRL]-[^]. When mapping these keystroke combinations, press [CTRL]-[v] prior to typing in the "action" characters.

Thereafter, pressing] with the cursor positioned on the name of the desired function displays that function, wherever it happens to be defined within the current application. Likewise, the [key returns you to the file from which you began.

Note

Some versions of vi do not recognize the [CTRL]-[]] shortcut.

It is not necessary to map keys in your `$HOME/.exrc` file to take advantage of the tags feature. You can use the tag command within vi to instantly edit or view any function defined in the program source code. For example, while using vi to edit a source code file (*.4gl), the command:

```
:tag func_name
```

loads the source code file containing function `func_name()`, with the cursor positioned on the first line of code defining `func_name()`. To return to the file from which the last tag command was given, type:

```
:e#
```

The tags utility can also be used to start vi from the command line. The command is:

```
vi -t func_name
```

In such cases, map the] keystroke as follows:

```
map ] :tag
```

Upon editing a file with vi, the] key begins the command:

:tag *func_name*

Simply complete the command with the name of the function you wish to edit, press [ENTER], and the tags feature loads the file containing the appropriate source code.

Sometimes tagging to a function might not give you the function you want. The `$fg/tags` file is built using a binary sort. If you try to tag on a function that appears in more than one library, you are placed into the function and library that appears first in the tags file. This means that the tags utility always takes you to the function that appears first alphabetically. For example, if you tag on the `init()` function while examining screen code generated programs, you are placed into the `$fg/lib/report.4gs/init.4gl` file. This obviously is not the `init()` you want to see.

The following lines are from the `$fg/lib/tags` file and illustrate how the first `init.4gl` is tagged to rather than the second:

```
init    $fg/lib/report.4gs/init.4gl     /^function init
init    $fg/lib/scr.4gs/init.4gl        /^function init
```

Displaying Functions Within Programs

The following shell script displays a list of all the function names and comments found within those functions in a specified program. Thus by running this script and specifying the name of a program, you can view a description of every function found within that program.

```
######################################################################
:
# awk script to display function name and comments.  specify filename(s)
# on command line.You can use wildcards and also redirect the output to
# a file.

awk '
BEGIN {
    TRUE=1
    FALSE=0
    inflag=FALSE
}

{
    if ( inflag == TRUE && $0 ~ /^#/ ) {
            # substitute space for pound sign(s)
            gsub( /#/, " ", $0 )
            print $0
        }
        else
            inflag = FALSE
    }

    /^function/ {
        gsub( /^function */, "", $0 )
        inflag=TRUE
        printf ("\n%s\n", $0)
    }' $@

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

Here is sample output.

```
lld_input()

returning -1 if tab pressed (next window), 0 otherwise

lld_b_field(field_name)

This function is called from the input function before every field. The
'prv_fld' variable contains the field we came from. The 'scr_fld' variable
contains the field we're going into. Set 'nxt_fld' if you want to skip this
field or exit input.

lld_a_field()

This function is called after every field.

lld_a_input()

This function is called whenever the input statement exits (except due to an
interrupt). If you don't want the input session to end, set the nxt_fld
variable to contain the field to be placed back into.

lld_event()

This function is called whenever the user presses an event key. The event is
mapped to the 'scr_funct' variable and processed here.
```

This script is extremely useful when trying to learn new programs. By running this script in a particular library, you can produce a listing of all functions and their descriptions in that library.

Viewing Database Table Layouts (`imap`)

Fitrix *Screen* includes the script `imap`, which can be used to provide information on individual database tables in a specified database. The script can also be used to provide a listing of all tables in a particular database.

The syntax for the `imap` script is as follows:

`imap [-dbname` ***`database`*** **`] [`*****`tablename`*****`]`**

If you pass a `-dbname` *`database`* argument, the `imap` script uses that database. Otherwise, `imap` searches for the value of the `$DBNAME` system variable (if it has been specified). If `$DBNAME` is not set, `imap` defaults to the `standard` database.

If you pass a table name, then the `imap` script returns the schema of that table. If no table name is specified, `imap` produces a list of tables in the database.

Invocation examples:

```
imap -dbname stores customer
imap -dbname stores
imap items
imap
```

Adding Code Generator Tables (`mktables`)

The UNIX script `mktables` adds tables to your database that are needed by generated programs and may also be used to create new databases in `$fg/codegen/data`. The `mktables` script also builds all of the tables required by *any* program using the `scr.a, user_ctl.a, or standard.a` libraries.

The syntax for the `mktables` script follows:

```
$fg/bin/mktables -dbname database
```

This script adds a number of cg* and stx* tables to your database. If space is a concern, you may remove the cg* tables from your application database. Only the stx* tables need to be present to run a program that was generated with the *Screen* Code Generator. For a complete list of tables refer to "Code Generator Tables" on page D-5.

Adding Tables Required by Fitrix Security (`mksecuri`)

In order for your generated applications to useFitrix Security, you need to add the security tables to your database. The following script adds the tables needed by Fitrix Security.

```
$fg/bin/mksecuri -dbname database
```

Adding Upper Level Library Tables (`mklib`)

Applications created with Fitrix *Screen* require a number of tables to be present in any database they are run against. The `mklib` script adds these tables.

```
$fg/bin/mklib -dbname database
```

Setting up the Shell (`fg.setshell`)

The `fg.setshell` script forces a program to be run in the bourne shell. The purpose of this shell script is to re-boot the current program using a shell that's known to work on this platform. Most platforms pass the QA suite using the regular bourne shell located in `/bin/sh`, but some platforms have shells that work better than the `/bin/sh` shell.

This program is usually called from within other shell scripts using the following format:

```
. fg.setshell
```

The `fg.setshell` script is located in `$fg/bin`.

Cleaning Your Database (`fg.delfrm`)

At times you may need to clean up old screen information from your database. For instance, if you delete a directory, the form images in that directory are still retained in the database.

Sometimes when a form is parsed, corrupt data may get into your database. The Code Generator usually cleans itself out when parsing a form but if bad data gets in, the Code Generator may be unable to clean itself out. If you run into strange errors when generating code you should try running this script and specify your problem directory and forms.

To run this script `$fg/bin` must be in your `$PATH`.

Syntax:

```
fg.delfrm { [-m module]|[-p program]|[-s scr_id]|
[-k cust_key] } [-dbname dbname] [-c] [u]
```

-m *module*	one module or a list of modules.
-p *program*	program.
-s *scr_id*	the name of the screen.
-k *cust_key*	specify `cust_key` if using version control.
-dbname *dbname*	database name. default standard.
-c	cron mode. no display output. default no cron mode.
-u	update statistics. default no update.

B

The .per Specification File

This appendix explains the .per specification file, which serves as input to the Code Generator. Although the Form Painter automatically creates these .per specification files, understanding the make up of the .per file gives you a better comprehension of the overall code generation process. This section covers:

- The .per specification file
- preventing code generation on a .per form
- converting INFORMIX-SQL perform files

The .per Specification File

The code generated by the Code Generator is based on information supplied by the .per file(s) present in the application directory. That is, the resulting code depends on the prior specification of the .per file(s). Even Zoom (lookup) logic can be automatically generated by the Code Generator provided there is a zoom .per form present prior to generating the application. The .per file(s) in any given directory can be created with any text editor or with the Fitrix *Screen* Form Painter.

The .per form specification file contains six basic sections: `DATABASE`, `SCREEN`, `TABLES`, `ATTRIBUTES`, and `INSTRUCTIONS`. An optional section titled FOURGEN is specified below the `INSTRUCTIONS` section. The sections of the .per file must appear in this order.

The `FOURGEN` section provides an additional source of information for the Code Generator. Since it is enclosed in braces "{}", this section is ignored by form4gl, the 4gl form preprocessor. Code can be generated without having a `FOURGEN` section specified in the .per file for header screens only. All other screen types must have a `FOURGEN` section to work with the Code Generator.

Note

All .per files must be less than 200 lines.

Note

.per file naming convention: .per file names must be seven characters or less not including the ".per" extension in order to be read into the Form Painter. For example: invoice.per is the maximum length of a .per name.

DATABASE Section

The DATABASE section specifies the database on which the form is based. The example below indicates that the stores database is the basis for the display fields on this form.

An example:

```
DATABASE stores
```

SCREEN Section

The SCREEN section of the .per file defines the image of the data-entry screen. Although 4GL accepts screens that are up to 600 rows by 600 columns, 4GL Forms limits the size of the SCREEN section to 74 characters in width by 18 lines in length (to allow for the border around the form). The section begins with the SCREEN keyword, and continues with the screen image enclosed by curly braces. Fields in the SCREEN section must also be defined in the ATTRIBUTES section. Field tags must be limited to six characters in order to work properly with the Form Painter.

Note

A { must precede the screen image and be on a line by itself.

Fields are generally delineated by square brackets. Sometimes you may need to locate two fields next to each other, and you only have one space for a delimiter so two square brackets do not work ( ][ ). When this occurs, you can use the pipe (|) to delineate between two fields.

An example:

```
   Contact Name: [f001                |f002                ]
```

An example SCREEN section:

```
SCREEN
{
----------------------------- Order Form ------------------------------
 Customer No.:[f000   ]    Contact Name:[f001           ][f002           ]
 Company Name:[f003                ]
      Address:[f004                ][f005                ]
  City/St/Zip:[f006           ][a0] [f007 ] Telephone:[f008              ]

   Order Date:[f010    ]     PO Number:[f011      ]    Order No:[f009    ]

 Shipping Instructions: [f012                                    ]
-----------------------------------------------------------------------
Item Description      Manufacturer            Qty.  Price       Extension
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
[f14][f15            ][f16][f17            ]  [f18 ][f19       ][f20       ]
                                                                 ===========
                              Order weight:[f30    ]    Freight:[f31       ]
                                                    Order Total:[f32       ]
}
```

TABLES Section

The TABLES section of the .per file lists the tables containing the columns specified in the ATTRIBUTES section. All tables listed must be part of the database in the DATABASE section. The first table listed in the TABLES section is the default main table for the program.

An example:

```
Tables
   orders
   items
   customer
   stock
   manufact
```

ATTRIBUTES Section

The `ATTRIBUTES` section found in the .per file coordinates the fields defined in the `SCREEN` section and optionally provides other attributes on a field-by-field basis. Field tags must be limited to six characters to work properly with the Form Painter.

An example:

```
ATTRIBUTES
f000 = orders.customer_num, comments =
  " Enter the customer code.";
f001 = customer.fname, noentry;
f002 = customer.lname, noentry;
f003 = customer.company, noentry;
f004 = customer.address1, noentry;
f005 = customer.address2, noentry;
f006 = customer.city, noentry;
a0 = customer.state, noentry;
f007 = customer.zipcode, noentry;
f008 = customer.phone, noentry;
f009 = orders.order_num, noentry;
f010 = orders.order_date, default = today, comments =
  " Enter the order date.";
f011 = orders.po_num, comments =
  " Enter the customer’s purchase order number.";
f012 = orders.ship_instruct, comments =
  " Enter any special shipping instructions to show on the invoice.";
f14 = items.stock_num, comments =
  " Enter the stock number for this line item.";
f15 = stock.description, noentry;
f16 = items.manu_code, comments =
  " Enter the manufacturers code for this stock number.", UPSHIFT;
f17 = manufact.manu_name, noentry;
f18 = items.quantity, comments =
  " Enter the number of units sold for this item.";
f19 = stock.unit_price, noentry;
f20 = items.total_price, noentry;
f30 = orders.ship_weight, comments =
  " Enter the total shipping weight for this order.";
f31 = orders.ship_charge, comments =
  " Enter the total shipping charge for this order.";
f32 = formonly.t_price type money, noentry;
```

Note

It is recommended that you do *not* use the `include` statement for data validation for the following reason: If bad data is entered into a field causing an error message, that data can be saved if the [TAB] key is pressed to move out of that field followed by an [ESC].

Formonly Fields

Formonly fields are used to display values of variables that are not associated with columns of the database. For more information on formonly fields refer to your Informix documentation.

INSTRUCTIONS Section

The `INSTRUCTIONS` section establishes the screen records used with the display columns on the data-entry form. Screen records can include some or all of the display fields established in the `ATTRIBUTES` section. When defining screen records, you must provide the full name of each display field to be included. With the Code Generator, you cannot use formats like `field1 THRU field13`, or `stores.*`.

The `INSTRUCTIONS` section should also contain the following `delimiters` statement changing the default display field delimiter to blanks (to accommodate the new field highlighting characteristics).

An example:

```
INSTRUCTIONS

screen record abc (table1.column, table2.column, ...)
screen record def[4] (table3.column, table4.column, ..)

delimiters " "
```

INSTRUCTIONS Section—Points to Observe

The following is a list of points to follow when building the `INSTRUCTIONS` section of the .per file.

- The first screen record listed in a header screen is the main record.
- The first screen record listed in a header/detail screen is the record for the header.
- The second screen record listed in a header/detail screen must be an array type and is the record for the detail lines. The number of elements of the array corresponds to the number of lines *on the screen*, not the number of program elements in the internal program array.
- The first table listed in the `TABLES` section of the screen determines the DEFAULT main table name for the header.
- The first ELEMENT of the detail screen record (table3.column in defaults above) determines the main table name for the detail lines.
- The table = tabname in the `FOURGEN` section (below) overrides the above defaults (and is the preferred method of determination).
- All fields in the .per file must appear in the screen record.
- If the table name of an element in the screen record is the same as the main table for that screen record (defined above), then the Code Generator provides data input/output for that column. The following example uses "customer" as the main table:

  ```
  screen record s_cust (customer.fname, customer.lname, orders.ord_num)
  ```

- If the table name of an element in the screen record is *not* the same as the main table for that screen record, then the field must either be a destination field of a lookup, or a math field. Otherwise the Code Generator doesn’t generate code for that field.

- If the table name isn't the main tablename for the screen record, and it's not a lookup destination or math result column, the programmer must fill that field in the appropriate `p_prep()` function (`llh_` for header, `lld_` for detail) manually.
- It is recommended to have the following delimiters statement in the instructions area of the file (with two spaces between the quotes) because the new cursor highlights the entire data field.

```
delimiters "  "
```

- If you do not have the delimiters statement (like above), then the field highlights *and* displays the specified `[ ]` delimiters (most users do not like both highlighting and the `[ ]` delimiters).

FOURGEN Section

The `FOURGEN` section of the .per form specification file provides a wealth of specific information used by the Code Generator to generate the program. While it is not required for other types of perform files, it is the method by which you control characteristics of the code generated by Fitrix *Screen* . If the `FOURGEN` section is not specified prior to code generation, default values are used.

The `FOURGEN` section is required when a header-detail screen has an `input 2` join line or when you want to use math, zoom, lookup,
or any other Code Generator function. The keyword `FOURGEN` must appear in the .per file for the above mentioned functions to work or when any `FOURGEN` specifications are used.

The following is a list of points to keep in mind when building the `FOURGEN` section of the .per form specification file.

- The entire `FOURGEN` section must be surrounded by braces `{ }` (form4gl treats this as a comment section).
- There is no case sensitivity in this section (`FOURGEN` doesn't need to be capitalized).
- The lines of #### above and below the word `FOURGEN` are not required.

Sample `FOURGEN` section:

```
{
#####################################################################
FOURGEN
#####################################################################

defaults
    type       = header/detail
    init       = order_num > 1000
    attributes = border, blue
    location   = 2, 3

input 1
    table   = orders
    key     = order_num
    filter  = order_date > "12/31/80"
    order   = order_num
    math    = t_price = sum(total_price) + ship_charge
    lookup  = key=customer_num, table=customer,
              filter=customer_num = $customer_num
    zoom    = key=customer_num, screen=cust_zm, table=customer

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

The `FOURGEN` section can contain 3 sections: `defaults`, `input 1`, and `input 2`. An explanation of the attributes in the `FOURGEN` section follows.

defaults section

The FOURGEN section of the .per form specification file contains a defaults section that defines the characteristics of the generated code. It is not mandatory that this information be provided as part of the .per file. Code Generator defaults are used in the absence of defaults section. The following section provides an overview of the components of this section.

type

(up to 15 chars)

```
    type = zoom
```

The scr_type can be only one of the following:

- **header**—this is a header only form or flat type. It contains one input area and one main table.
- **header/detail**—this is a header with another scrolling (detail) section joined to the header.
- **add-on header**–this is a flat type like a header form, only it is used in conjunction with a header/detail form to provide multiple screens.
- **add-on detail**—this is an addition scrolling detail form that can be attached and used with the main screen.
- **extension**—this is an additional screen that serves as an extension of the main screen.
- **view-header**—this is a view-only header form that can be attached to a main screen.
- **view-detail**—this is a view-only detail form that can be attached to a main screen.
- **query**—this is a form that can be used to build SQL query.
- **browse**—this is a scrolling type screen whose main table is the same as the header section main table. It allows you to view one row of the header table per line rather than one row per screen.

- **zoom**—this is a special type of screen that allows you to scroll through data from another table (or set of tables joined).

Default: If there is only one input area (and that input area is not an array type) then the screen type is assumed to be header only. If there are two input areas (a non-array type followed by an array type) then the screen type is assumed to be header/detail. If there is only one input area, and it is a scrolling type, then the screen type is assumed to be zoom *unless* the filename is `browse.per`. In that case, the default screen type is browse.

init

(up to 200 chars)

```
init = customer_num is not null and customer_type = "A"
```

This is the initial filter that is used when the application is first run. You can override it by invoking the program with a `filter = "filter clause"` on the command line.

Default: 1=0 (select no rows upon program load). You may also specify 1=1 to select all rows, or [as above] you may specify an SQL filter.

attributes

(up to 30 chars)

```
attributes = blue, border
```

This overrides the default window attributes. To retain consistency throughout your applications, we recommend not using the attributes keyword.

Default: white, border

location

(two integers separated by a comma)

```
location = 5,20
```

This specifies the y (row), x (column) location to place the window.

Default: 2, 3 (minimum upper left coordinates)

returning

(field name)

```
returning = customer_num
```

This keyword is for zoom screens only. It identifies the name of the field in this screen that you wish to return to the function that called the zoom. You may only specify one field to return to the calling function. This keyword is *not* required for zoom screens. If it is omitted, however, no data can be returned to functions calling this zoom screen.

input section

The `FOURGEN` section of the .per form specification file continues with the `input` section(s), which define(s) the characteristics of the generated code. The input designator must be followed by a number designating which input area (sequentially) you are defining. The only screen type that has more than one input area is header/detail. `input 1` corresponds to header sections while `input 2` corresponds to detail sections. The following section provides an overview of the components of this section, and serves as an example of how it can be used.

order

(up to 100 chars)

```
order = company, lname
```

This determines the order of each Find. You may specify `desc` after any column to impose a descending sort on that column. The order defaults to ascending order.

In .per terms `desc` is the string following the `order =` within the `FOURGEN` section of a .per file. When you have an `order by` in the select statement, the columns you are ordering by must also be in the `select` clause of the select statement. If these are not present, you get a syntax error. Basically, if there is no `order by` requested in the .per file, the only thing you have in your `select` clause is "rowid". If an `order =` is specified in the .per file, then the `order by` requested is put into a char variable called `sql_order`. At the time of the SQL statement creation, the contents of the `sql_order` variable need to be checked.

table

(default main table is specified in TABLES section)

```
table = customer
```

This defines the main table for this input area. There can be only one main table for an input area.

key

(unlimited list of fields)

```
key = customer_num
```

This defines the list of fields that build the unique key for the main table. The system uses this information to key secondary data to the main table. This secondary data includes Freeform Notes and User Defined Fields. If the key is not defined, you do not get Freeform Notes or User Defined Fields. The fields in the key must not exceed 30 characters in length, and the total combined key length may not exceed 80 characters.

Duplicate checking code is automatically generated on the fields listed on the key line of the `input 1` header region in the FOURGEN section. Duplicate checking logic is only generated if at least one of the fields in the key is enterable. A duplicate error message is displayed after a row is accepted and the user is placed back in the first enterable field in the key.

Notes:

1. Duplicate checking is not performed on detail rows.
2. Duplicate checking is only performed if all the fields in the key are non-null.
3. All entry fields in the key should be made required fields using the `nonull` keyword.
4. Duplicate checking is performed in the `llh_a_input()` function. The actual test for a duplicate value is done in `llh_dupchk()`.
5. If a duplicate has been created, the message, "This value already exists" appears in a box at the bottom of the screen when the user presses the accept [ESC] key. The user is placed back in the first enterable field in the key.
6. If you do not want to utilize the default duplicate checking logic:

- create a trigger that has a `do_not_generate llh_dupchk`.
- create your own `llh_dupchk` that always returns true.

join

(up to 200 chars)

```
join = customer.customer_num = orders.customer_num
```

This defines the join criteria for detail lines joined to the header input area and also defines the join criteria for zoom screens (if there is more than one table listed in the `TABLES` section, this defines the join for those tables).

filter

(up to 200 chars)

```
filter = items.item_num is not null  (for detail)
```

This is the hardcoded filter that is used in every query. This filter is combined using `AND` with the user's query by example filter, and the filter passed via the command line. You can also use the `matches`, `not matches`, `like`, and `not like` operators in the filter.

Default: 1=1 (no hardfilter) Example: If you *only* wanted to see customers with a `customer_num` greater than 1000 in this program, you would specify:

```
filter = customer_num > 1000
```

arr_max

(integer)

```
arr_max = 200
```

This is the number of internal program array elements you wish to provide space for in the scrolling input area. It only is used for detail and zoom type input areas.

Default: 100

autonum

(auto sequencing of detail lines)

```
autonum = line_no
```

This is for detail input areas only. It sequences the detail lines by specifying a unique line number field that the system maintains. Autonum maintains the order that detail lines are entered. If you do not specify autonumber for your detail lines, the detail lines are not displayed in the order that they were originally entered. This line number field is not recommended to be listed in the form image because it is not maintained during input. It is maintained only upon disk writes.

math

(math statement)

```
math = t_price = sum(total_price) + ship_charge
```

This statement tells the Code Generator how to perform math on the screen. The system knows about addition, subtraction, multiplication, division, modulus, and summation of detail fields. The first element must be the destination field, followed by an equal sign and the mathematical formula required to fill the destination field.

blobdef

(blob definition)

```
blobdef = blobtext, vi, y
blobdef = blobbyte, Wingz, y
```

The `blobdef` statement allows you to link Informix BLOBs (Binary Large Objects) to fields in your form. BLOBs may be of type byte or text. BLOB field types are only supported in the `input 1` (header) section of header and header/detail forms. For more information on blobs refer to "Creating BLOB Fields" on page 7-18 and "Creating BLOBs" on page 15-31.

```
blobdef = column_type program {y|n}
```

The `blobdef` is the keyword. The `column_type` is the datatype. The `program` is the program that invokes the blob. The `y/n` flag determines whether the blob can be edited.

lookup

```
lookup = name=stock_manu, key=manu_code, table=stock,
          filter=stock_num = $stock_num and manu_code = $manu_code,
         into=unit_price
```

This statement defines a system lookup. Lookups are keyed from a field in the main table for this input, and they look up information from another table to place into destination fields. The destination fields should be noentry type.

The Code Generator attempts to find a column in the table by the name of the `into` field. In the example above, code is generated to select the `unit_price` column from the `stock` table; so it is important that the field names you select in the `screen.per` match the actual names of columns in tables. If the field on the screen has a different name than the column in the table being looked up then you must use the `from_into` statement.

The `into` statement is needed when there is more than one lookup to the same table. Otherwise the default destination is all fields in the input area that share the same table name.

Note

Lookups must appear in the .per file in the order they are needed. If a lookup depends upon another, you need to list the lookups in the .per form in the order that they are performed.

The following describes three examples of lookup usage:

1. If there is no `into` statement, the generator searches the screen record for definitions of the same table as the table name of `table=tablename`.

   ```
   screen record s_pvendr (stpvendr.vend_code, stpvendr.bus_name,
                           stpvendr.terms_code, stptermr.terms_desc)
        ...
        ...
        lookup = name=term_lookup,key=terms_code,table=stptermr,
        filter= stptermr.terms_code = $terms_code
   ```

   The generator finds `stptermr.terms_desc` in the screen record therefore defaulting the `into=terms_desc`. If the generator cannot find an associated table, then the lookup is defined as a validation only lookup (a lookup that returns no data).

2. If you use the `into` statement, all `into's` must be specific. You cannot use the `into` statement for some fields and expect the generator to default the other ones.

   The `into=column` must be a column in the lookup table. It does not have to be a screen record field. If your screen record field has the same name as the column then the lookup returns data into that field otherwise it puts that data into a parallel record.

   ```
   screen record s_acct (stpinvce.acct_no, formonly.acct_desc)
        ...
        ...
        lookup = name=acct_lookup, key=acct_no, table=stxchrtr,
        into=acct_desc, into=incr_with_crdt,
        filter= stxchrtr.acct_no = $acct_no
   ```

   The generator puts the `acct_desc` into `p_pinvce.acct_desc` and `incr_with_crdt` into `q_pince.incr_with_crdt`.

   The `p_` record is associated with the screen and the `q_` records are parallel to the `p_` records.

3. If you want to assign a lookup where the column selected is not the same name as the field you want to put it into, you can use the `from_into` syntax.

   ```
   screen record s_acct (stpinvce.acct_no, formonly.james_desc)
       ...
       ...
       lookup = name=acct_lookup, key=acct_no, table=stxchrtr,
       from_into=acct_desc james_desc, from_into=incr_with_crdt
                 is_it_a_credit,
       filter= stxchrtr.acct_no = $acct_no
   ```

   The generator puts the `acct_desc` `into p_pinvce.james_desc` and `incr_with_crdt` into `q_pince.is_it_a_credit`.

4. You can specify a lookup that does not return information to a field for data validation purposes by not using the `into` statement. Information is returned in this case as long as your screen record does *not* contain any fields that reference the table that the lookup is made into.

   A lookup can also be used for data validation as in the following example (that appears in `header.4gl`):

   ```
   if llh_lookup("cust_lk",true) = false
   ```

   In this example, when the function is passed a value of true, then the field must be filled with a valid code *and* the lookup takes place. If the function is passed a value of false, the lookup occurs, but data validation does not. However, it is up to the programmer to take care of the situation where the field is left blank. See the following example:

   ```
   after field cust_code
     if auto_zoom("custz","strcustr","")
       then let p_rshipr.cust_code = scratch end if
     call lib_after()
     if data_changed
     then
         # Perform lookups
         if llh_lookup("cust_lk",true) = false
           then next field cust_code end if
     end if
   ```

The lookup statement looks for the following elements separated by commas: `name`, `key`, `table`, `filter`, `into`, `from_into`.

lookup - name

(unique lookup name - up to 18 chars)

```
 name = custlkup2
```

This defines a unique name for the lookup. The default name is the name of the table. You would only need to define a name if you have more than one lookup in an input area that looks up from the same table.

lookup - key

(fieldname)

```
key = customer_num
```

This defines the field that the lookup is triggered on. In this case, when you change the data in the `customer_num` field a new lookup is performed. The key attribute is required.

lookup - table

(tablename)

```
table = customer
```

This defines the table to be looked into. There can only be one table per lookup. This is required information.

lookup - filter

(SQL filter)

```
filter = customer.customer_num = $customer_num
```

This is the `where` clause that is built when the lookup is performed. If you have a $ preceding a field, that tells the system to use the data in program variables as that part of the filter. The filter clause must be expressed in one line. The above example places the current `customer_num` (say, `104`) into the following `where` clause:

```
where customer.customer_num = 104
```

The filter statement is required.

lookup - into

(screen field in this input area)

```
into=fname, into=lname, into = company...
```

This tells the lookup which field to place the data into. You may have any number of `into` statements, but you may only specify one field per `into`. Normally, you do not need to specifically tell the system which fields to input into. The default `into` list is determined by all fields defined in the screen record that have a table-name that matches the table for this lookup. You may want to override that default list if you do not wish to have all elements from that table filled on this lookup, or if you wish to lookup into columns of the main table for the input area.

lookup - from_into

(screen field in this input area)

```
from_into=acct_desc james_desc
```

The `from_into` statement allows you to place information that is retrieved from one column into a field with a different column name. In the preceding example, whatever is found in the `acct_desc` column is placed into the `james_desc` field on this form. The column must be a column in the lookup table and the `into` field must be a field on the screen.

zoom

(definition)

```
zoom=key=manu_code, screen=stk_mnu, table=stock,
     filter=stock.stock_num = $stock_num
```

This statement defines the relationship between this screen and a zoom screen. It has multiple sections like the lookup statement described previously.

zoom - key

(field to zoom from)

```
key = customer_num
```

This defines the field you wish to zoom from, and the field that the zoom screen returns data into.

zoom - screen

(zoom screen name - up to 7 chars)

```
screen = stk_mnu
```

This specifies the name of the zoom screen to call when the user gives the zoom command in the key field.

zoom - table

(tablename)

```
table = stock
```

This defines the main table for the zoom. It is required only if you wish to have AutoZoom functionality. The system needs it to build the matches clause for autozoom.

```
where {tablename}.{key} matches {data typed into key field}
```

When running the application, if you are in an AutoZoom type field and an asterisk is entered anywhere in the field, any data in that field is used to build the matches clause, and the zoom screen is automatically called.

zoom - filter

(SQL filter)

```
filter=stock.stock_num = $stock_num
```

This is the `where` clause that is built when the Zoom is performed. If you have a $ preceding a field, it tells the system to use the data in the system as that part of the filter. In the above example, the current `stock_num` (say, 4) is placed into the following where clause:

```
where stock_num = 4
```

If a zoom filter is specified, it uses that filter instead of placing the user into the query by example screen. If no documents are selected based on this filter, the user *is* put into the query by example screen. In either case, if the user executes Find from this query by example screen, the filter is *not* used in the subsequent select.

For example, if you use the following filter:

```
filter=1=1
```

then the query screen is not called when the zoom screen is first displayed.

zoom - from

(column name differs from screen field)

```
from=column_name
```

This statement lets you specify the name of the column being zoomed into if it differs from the name of the column on the screen. For example, if the field on your screen is named `customer_num` and the field being zoomed into is called `cus_num`, you would specify `from=cus_num`.

The `from` keyword is required for a zoom definition in the `FOURGEN` section if all of the following are true:

1. The screen field which is the key to your zoom has a different name than the corresponding column name in the table you are perusing with your zoom.

2. The screen field you are zooming from is a character field.

3. The zoom has autozoom enabled. Zooms with the `noautozoom` keyword or a filter do not require the `from` keyword.

```
zoom= key=state, screen=stat_zm, table=state, from=code
```

noautozoom

(do not generate autozoom logic)

```
zoom    = key=trd_ds_code, screen=discz, noautozoom,
          table=stxinfor,filter=stxinfor.src_type = "I"
```

This keyword requires no arguments. It tells the Code Generator not to generate autozoom code for this field.

Specifying `noautozoom` eliminates the following two lines for the specific field (in this case, `trd_ds_code`).

```
if auto_zoom("discz","stxinfor","(see scratch)")
  then let p_rcustr.trd_ds_code = scratch end if
```

You would specify `noautozoom` if you *needed* to have an asterisk as a piece of data in a zoom key field, or if the zoom key field is a numeric type field that cannot use the matches clause in an SQL query.

defaults

```
default = customer_num = 105, address1 = "805 Westham
          PKWY",order_date = today, po_num = "3K5100"
```

Use the `default` keyword in either `input 1` or `input 2` of the `FOURGEN` section of the perform screen to place default values in fields on the screen.

Numeric fields:

```
default = field-name1 = numeric-value, field-name2 = numeric-value
```

Character fields:

```
default = field-name1 ="char. string", field-name2 = "char. string"
```

Variable defaults:

```
default = field-name1 = variable-name, field-name2 = variable-name
```

Explanation:

- `default` is the required keyword.

- `field-name` is the name of a column in the screen record without the table name prefix.

```
Correct:  customer_num
Incorrect: orders.customer_num
```

- `numeric value` is an integer or other numeric constant.
- `char. string` is a character string enclosed in quotes with a length less then or equal to 30 characters.

Notes:

1. If any defaults are present in the `ATTRIBUTES` section, the Code Generator creates a default entry with the `default` keyword in the appropriate input area of the `FOURGEN` section.
2. Defaults are limited to 30 characters in length. The default line can contain many default values for fields with each default value having a maximum length of 30 characters.
3. Defaulting is not performed in `input 1`, header, unless all of the variables in the `input 1` program, `p_` record are null.
4. Defaulting is not performed for a specific row in the detail input array unless all the program, `p_` record variables for a given row are null.

nonull

(entry required)

```
nonull = po_num, customer_num, ship_charge
```

The `nonull` keyword is used to require entry in fields even when data is changed or a field is entered multiple times.

Notes:

1. The `nonull` keyword is only available for the header input 1 region.
2. If the `REQUIRED` keyword is found in the `ATTRIBUTES` section for a field in the input 1 area the Code Generator and the Form Painter remove it and create a `nonull` entry in the `FOURGEN` section.

3. The REQUIRED keyword may be used in the attribute section for fields in the input 2, detail, region. It is not stripped by the tools. However, this technique should not be used.
4. Nonull logic is written in llh_a_input().
5. When the accept key is pressed with nonull fields left null an error message, "A required field is null," appears in a box at the bottom of the screen. The user is placed back in the first null nonull field based upon cursor path.

Preventing Code Generation on a .per Form

Each input screen in your application can only have one source form. A source form is a .per form used to generate code. If you have two similar header or header/detail .per forms in one directory, and you try to generate code, the Code Generator generates code for the first form, then for the second form. The code generated off of the second form overwrites the code for the first form.

Sometimes you might have a situation where you want to generate code from one form, but then you want to physically display a different form. In this instance you would specify the display only form as a non_source_form. This statement tells the Code Generator to skip this form and not generate off of it.

If you have a .per in a local directory that is not to be used for generating code, the first line following the copyright information of the .per should contain the statement non_source_form. An example:

```
{
###########################################################
# Sccsid:  %Z%  %M%  %I%  Delta: %G%
############################################################

non_source_form
}
```

The non_source_form statement should appear following copyright information *within* the braces. Anything contained within braces is ignored by form4gl.

When you run the Code Generator on all forms in a local directory, it does not generate code for those .pers that contain this line. The `non_source_form` statement allows you to have multiple screens in the local program directory and eliminates any chance of running the Code Generator on the wrong screen.

Converting INFORMIX-SQL Perform Files

Perform applications written in INFORMIX-SQL may be easily converted to INFORMIX-4GL by running the Code Generator on the perform screens. The code created by Fitrix *Screen* effectively replaces the perform screen interface with the Code Generator ring menu interface. Many Perform commands are not recognized by INFORMIX-4GL, however, if you have defined many additional instructions in the perform form, you have to add functions to the Code Generator code to achieve the same effects.

- All statements of the `INSTRUCTIONS` section of a perform screen, with the exception of the `delimiters` command, are ignored.
- 4GL does not accept form definitions containing more than one screen. If your perform file contains multiple screen definitions these must be removed.
- Joins defined in the screen form are ignored.
- The `LOOKUP, NOUPDATE, QUERYCLEAR, RIGHT`, and `ZEROFILL` attributes of perform screens have no meaning in 4GL.

C

Program Migration

This section discusses moving your generated applications onto production platforms.

- Program migration

Moving Applications to Other Systems

To successfully run programs generated with Fitrix *Screen* on systems other than the development system, a few steps must be taken. These steps will ensure that the tables, data, and forms 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`
   - `$fg/lib/data/library`
3. Change your `PATH` on the target system to include the `$fg/bin` directory.
4. Be sure that each database to be converted is in the `$DBPATH`.
5. Run `mklib -d` *`database`* on each database that needs to be converted.

   The `mklib` script adds a number of tables required by Fitrix generated applications.
6. If you are also installing Fitrix *Screen* at the customers site then you need to run `mktables -d` *`database`*, which adds a number of cg* and stx* tables to your database.

7. If your customerwill be using Fitrix Security, which is included with the Enhancement Toolkit, you need to run `mksecuri -d` *`database`* to add security tables.

These steps are required to make the appropriate forms available to the application:

8. Create a `$fg/lib/forms` directory on the target system.

9. Copy the files in the `$fg/lib/forms` directory on the development system into the `$fg/lib/forms` directory on the target system.

10. 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 and/or custom runners

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

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

D

Screen Tables

This appendix covers the database tables used by *Screen*.

- Adding Code Generator tables with `mktables`
- A list of the Code Generator tables

Required Tables

The following table lists the database tables required for *Screen* to run, as well as what tables are needed to run the generated application.

Table	Description	*Screen*	Generated Applications
`cgdcolmr`	data dictionary for database columns	X	X
`cgdtablr`	data dictionary for database tables	X	X
`cgmcmdd`	menu item definition detail	X	X
`cgmcmndr`	menu item definition header	X	X
`cgmmenud`	program menu definition	X	X
`cgmmposr`	menu position	X	X
`cgrrimgd`	line image for reports	X	X
`cgsblobr`	contains blob information	X	
`cgsclipr`	clipboard detail	X	
`cgscmdsd`	featurizer detail table	X	
`cgscmdsr`	featurizer header table	X	
`cgsdpndd`	field dependency list	X	
`cgsifldd`	input field definition	X	
`cgsimged`	storage for screen image	X	
`cgsinptr`	input area definitions	X	
`cgsscrnr`	main screen definitions for a .per	X	X
`cgsstypr`	screen defaults	X	

Table	**Description**	***Screen***	**Generated Applications**
`cgstrigd`	stores triggers code	X	
`cgstrigr`	available triggers	X	
`cgszoomr`	zooms (from fields)	X	
`cgxfnamr`	screen type local function name	X	
`cgxfsetd`	functions generated for screen type	X	
`cgxlkupr`	lookups	X	
`cgxlntod`	lookup from/into detail	X	
`cgxmathr`	math	X	
`cgxsorcd`	pathname of perform and triggers files	X	X
`pcdtablr`	contains list of module tables	X	X
`stxacknd`	software acknowledgement detail	X	X
`stxactnr`	navigation event reference	X	X
`stxaddld`	user defined fields detail	X	X
`stxaddlr`	user defined fields header	X	X
`stxcompr`	list of valid companies for mz	X	X
`stxerord`	base files error text detail	X	X
`stxerorh`	base files error text header	X	X
`stxfiler`	operating system information	X	X
`stxfiltr`	scheduling for reports	X	X

Table	Description	*Screen*	Generated Applications
`stxfuncr`	security events	X	X
`stxgropd`	group permission security detail	X	X
`stxgropr`	group permission security header	X	X
`stxhelpd`	base files error text header	X	X
`stxhotkd`	hot key definitions detail	X	X
`stxkeysr`	hot key definitions reference	X	X
`stxlangr`	language translation table	X	X
`stxmssgr`	stores program comments	X	X
`stxnoted`	freeform notes detail	X	X
`stxnvgtd`	navigation events detail	X	X
`stxparmd`	parameter detail	X	X
`stxparmh`	parameter header	X	X
`stxprogr`	list of programs	X	X
`stxsecud`	security permissions detail	X	X
`stxsecur`	security permissions header	X	X
`stxtodod`	todo list detail	X	X
`stxtxtdd`	default text	X	X
`stxuniqc`	unique serial control	X	X

Note

If you have *Report* Writer, the `cgdtablr`, `cgdcolmr`, and `pcdtablr` tables must be present.

Code Generator Tables

Screen automatically builds in a wide range of features to expand the power and versatility of your data-entry interface. In order to accommodate these features, the generated code must be able to rely on the existence of specialized tables to maintain the information these features support.

As an application is being generated, the Code Generator searches the specified database (named in the .per file) for these tables. If they do not exist, they are created. This section lists the tables referenced by features of generated code (in alphabetic order).

`cgdcolmr`—the data dictionary for columns

```
tabname char(18),
colname char(18),
language char(3),
description char(30),
col_label char(30),
mssg_line char(74)
```

`cgdtablr`—the data dictionary for tables

```
tabname char(18),
language char(3),
description char(30),
uniq_list char(60),
tab_order char(60)
```

`cgmcmndd`—menu item definition detail

```
m_name char(8),
m_order smallint,
m_lang char(3),
m_label char(20),
m_help char(50)
```

cgmcmndr—menu item definition header

```
m_name char(8),
m_desc char(20),
m_order smallint,
m_type char(1),
m_event char(8),
m_class char(12),
m_style char(1),
m_rowon char(1),
m_curon char(1),
m_toton char(1),
m_deton char(1),
m_enter char(1)
```

cgmmenud—program menu definition

```
module char(8),
prog char(8),
scrid char(8),
m_local smallint,
m_name char(8),
m_desc char(20),
m_order smallint,
m_type char(1),
m_event char(8),
m_class char(12),
m_style char(1),
m_rowon char(1),
m_curon char(1),
m_toton char(1),
m_deton char(1),
m_enter char(1)
```

cgmmposr—menu position

```
m_name char(8),
m_type char(1),
x_pos smallint,
y_pos smallint,
width smallint,
hold char(1)
```

cgsblobr—the blob description table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
fldname char(18),
runprog char(60),
progedit char(1)
```

cgsclipr—the clipboard detail table

```
username char(12),
clipname char(8),
clip_title char(30),
seqno serial not null,
srow smallint,
erow smallint,
scol smallint,
ecol smallint,
clip_status char(1)
```

cgscmdsd—featurizer detail table

```
cmd_id integer,
line_no smallint,
txt char(70),
indent smallint,
cont_line char(1),
whitespace smallint
```

cgscmdsr—featurizer header table

```
cmd_id serial not null,
cmd_type char(1),
cmd_order smallint,
trig_code smallint,
module char(14),
prog char(14),
cust_key char(3),
filename char(14),
load_time integer,
src_file char(14),
funct_id char(18),
block_id char(20),
from_after char(1),
from_str char(50),
to_thru char(1),
to_str char(50)
```

cgsdpndd—the field dependency list table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
src_type smallint,
src_field char(18),
dpnd_field char(18)
```

cgsifldd—the input field definitions table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
seqno smallint,
field_tag char(6),
tabname char(18),
fldname char(18),
fldtype char(42),
f_autonext char(1),
f_comments char(74),
f_default char(30),
f_display_like char(42),
f_downshift char(1),
f_format char(50),
f_include char(50),
f_picture char(50),
f_noentry char(1),
f_required char(1),
f_upshift char(1),
f_valid_like char(42),
f_verify char(1)
```

cgsimged—the screen image storage table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
line_no smallint,
image_data char(132)
```

cgsinptr—the input area definitions table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
scr_rec char(12),
maintab char(18),
sel_join char(200),
sel_filter char(200),
sel_order char(100),
unique_key char(80),
ok_delete char(1),
auto_number char(18),
scr_siz smallint,
arr_max smallint
```

cgsscrnr—the main screen definitions table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
scr_type char(15),
maintab char(18),
init_filter char(200),
win_x smallint,
win_y smallint,
delimiters char(2),
_returning char(18),
scr_attr char(30),
load_time integer,
trig_time integer,
non_src_frm char(1),
eng_ver char(10),
fgl_ver char(10)
```

cgsstypr—the default screen type table

```
set_key char(20),
dflt_arr_max integer,
userdef char(1)
```

`cgstrigd`—the trigger code table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
trig_code smallint,
arg_one char(18),
trig_order smallint,
trig_text char(74)
```

`cgstrigr`—the triggers table

```
trig_def char(30),
trig_code smallint
```

`cgszoomr`—the zoom table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
key_field char(18),
zoom_scrid char(7),
zoom_table char(18),
noautozoom char(1),
sel_filter char(200),
zoom_from char(18)
```

`cgxfnamr`—screen type local function name

```
set_key char(20),
func_key char(10),
func_name char(18)
```

`cgxfsetd`—functions generated for screen type

```
set_key char(20),
func_key char(10),
userdef char(1)
```

cgxlkupr—the lookups table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
lkup_name char(18),
key_field char(18),
lkup_table char(18),
sel_filter char(200)
```

cgxlntod—the lookup from/into detail table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
manual char(1),
lkup_name char(18),
lkup_into char(18),
lkup_from char(18)
```

cgxmathr—the math table

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
input_num smallint,
key_field char(18),
math_text char(100)
```

cgxsorcd—stores the pathname of perform and trigger files

```
module char(8),
prog char(8),
scrid char(7),
cust_key char(12),
src_code char(3),
src_name char(200)
```

pcdtablr—the list of modules table

```
prodid char(8),
line_no smallint,
tabname char(18)
```

stxacknd—the software acknowledgement detail table

```
ack_module char(8),
ack_program char(8),
line_no smallint,
ack_text char(60)
```

stxactnr—the navigation event reference table

```
language char(3),
act_key char(15),
description char(30),
os_command char(74),
press_enter char(1)
```

stxaddld—the user-defined fields detail table

```
filename char(18) not null,
record_key char(30),
line_no smallint,
data char(30)
```

stxaddlr—the user-defined fields header table

```
filename char(18) not null,
line_no smallint,
field_label char(20)
```

stxcompr—list of valid companies for use with mz

```
comp_id char(8),
db_name char(14),
logfile char(150),
line_no smallint,
consolidate smallint
```

stxerord—the base files error text detail table

```
language char(3),
userdef char(1),
err_module char(8),
err_program char(8),
err_number smallint,
a_b char(1),
line_no smallint,
err_text char(60)
```

stxerorh—the base files error text header table

```
language char(3),
userdef char(1),
err_module char(8),
err_program char(8),
err_number smallint,
err_line char(40)
```

stxfiler—the operating system information table

```
uniq_num integer,
line_no serial not null,
line_text char(248)
```

stxfiltr—schedule for reports

```
unique_id char(15),
seq_no smallint,
```

stxfuncr—security events

```
module char(8),
progname char(8),
funcname char(20),
description char(30),
allow_flag char(1),
userdef char(1)
```

stxgropd—the group permissions detail table

```
group_id char(8),
user_id char(8)
```

stxgropr—the group permissions header table

```
group_id char(8),
description char(30)
```

stxhelpd—the base files help text detail table

```
language char(3),
userdef char(1),
hlp_module char(8),
hlp_program char(8),
hlp_number smallint,
line_no smallint,
hlp_text char(60)
```

stxhotkd—the hot key definitions detail table

```
hot_key smallint,
act_key char(15),
hot_module char(8),
hot_program char(8),
hot_user char(10)
```

stxkeysr—the hot key definitions reference table

```
key_code smallint,
key_desc char(15)
```

stxlangr—language translation table

```
language char(3),
tr_tab_col char(37),
tr_context char(10),
native char(50),
non_native char(50)
```

stxmssgr—stores program comments

```
language char(3),
mssg_module char(8),
mssg_program char(8),
mssg_number smallint,
message char(132)
```

stxnoted—the freeform notes detail table

```
filename char(18) not null,
record_key char(30),
line_no smallint,
data char(60)
```

stxnvgtd—the navigation events detail table

```
act_key char(15),
line_no smallint,
nav_module char(8),
nav_program char(8),
nav_user char(10)
```

stxparmd—the user permissions table

```
language char(3),
module char(8),
user_id char(8),
access_key char(30),
line_no integer,
userdef char(1),
sbd_flag char(1),
parm_desc char(30),
is_rule char(1),
is_fatal char(1),
help_num smallint,
val_table char(30),
val_column char(60),
val_filter char(60),
val_join char(60),
val_switchbox char(8),
val_description char(18),
zoom_filter char(60),
zoom_switchbox char(8),
parm_value char(32)
```

stxparmh—the program permissions table

```
language char(3),
module char(8),
heading char(76)
```

stxprogr—the list of programs table

```
module char(8),
progname char(8),
description char(30),
use_trx smallint,
userdef char(1)
```

stxsecud—security permissions detail

```
user_id char(8),
module char(8),
progname char(8),
funcname char(20),
allow_flag char(1)
```

stxsecur—security permissions header

```
user_id char(8),
lname char(20),
fname char(20),
minitial char(1),
company char(30),
dept char(15),
responsibility char(30),
phone char(15)
```

stxtodod—the to do list detail table

```
todo_user char(10),
line_no smallint,
todo_text char(60)
```

stxtxtdd—the default text table

```
txt_key char(30),
line_no smallint,
dflt_text char(74)
```

stxuniqc—the unique serial control table

```
uniq serial not null
```

E

Control Key Defaults

This appendix provides a list of control key defaults and a cross reference for engine compatibility:

- Control key defaults
- Engine compatibility

Control Key Defaults

	Control Key Defaults	Trapped During Input
^a	Toggle between insert and overstrike modes (Informix edit key)	
^b	Back Tab	yes
^c	Operating system key (on DOS systems)	
^d	Delete to end of line (Informix edit key)	
^e	Edit hot keys	yes
^f		yes
^g	Navigate (go)	yes
^h	Same as left arrow (Informix edit key)	
^i	Tab	yes
^j	Same as down arrow (Informix edit key)	
^k	Same as up arrow (Informix edit key)	
^l	Same as right arrow (Informix edit key)	
^m	Enter	
^n	Toggle input areas	yes
^o	Operating system exit	yes
^p	Paste	yes
^q	Reserved for O/S (continue screen output)	
^r	Redraw the screen	
^s	Reserved for O/S (stop screen output)	

	Control Key Defaults	Trapped During Input
^t		yes
^u	Undo	yes
^v	Mark/Copy	yes
^w	Help	yes
^x	Delete character (Informix edit key)	
^y	Display Program Information menu	yes
^z	Zoom	yes

	Function Keys Trapped During Input
F1-F4	At menu level: User-definable menu hot keys
F1-F4	During input: Informix movement keys F1=Insert line, F2=Delete line, F3=Page down, F4=Page up (during input of a non-scrolling section, F1-F4 aren’t used)
F5-F16	User definable hot keys
F17-F30	User definable hot keys (must add logic to trap)
F34	Hardmapped to "^B" (Back Tab"
F35	Hardmapped to "ESC" (accept)
F36	Hardmapped to "DEL" (cancel)

Engine/4GL Compatibility

The following table lists 4gl data types and the engines on which they run.

Data Type	4GL	Engine	Comments
byte (BLOB)*	>=4.10	Online	header forms only
char	all	all	
date	all	all	
datetime	>=4.10	all	
decimal	all	all	
float	all	all	
integer	all	all	
interval	>=4.10	all	
money	all	all	
serial	all	all	
smallint	all	all	
smallfloat	all	all	
text (BLOB)*	>=4.10	Online	
varchar	>=4.10	Online	and formonly on SE

*Binary Large OBject

F

Reserved Terms and Style Guide

This appendix contains the following information:

- A list of reserved terms
- Table naming conventions
- Screen form style guide

Reserved Terms

There are a number of terms that can be considered reserved, for their inadvertent use may cause unexpected problems with code generated by Fitrix *Screen*. When modifying code created by the Code Generator, it is helpful to check against this list to ensure that duplication of names does not occur.

Many terms have special significance with Informix products, and cannot be used when modifying generated code. Please check the *INFORMIX-4GL User Guide* for the list of INFORMIX-4GL reserved words.

The list of reserved variable names is restricted to the variables found in the `globals.4gl` file. The list is reprinted here for convenience.

```
scr1_max       rec1_max      rec1_cnt       progid
scr_id         menu_item     scr_funct      sql_filter
sql_order      input_num     p_cur          s_cur
scr_fld        prev_data     this_data      data_changed
hotkey         scratch       nxt_fld
```

The list of reserved function names can be easily identified with the help of the tags feature. For information on the Tags feature refer to "The Tag Utility" on page A-5. The tags file, which is re-created every time the relevant source code is recompiled, contains a list of function names and locations called by the application.

The list provided in this manual is based on the demo application (orders), and the library source generated as part of the application.

```
cust_zm        stk_mnu       stockzm        brw_close
brw_display    brw_hook      brw_open       lld_add
lld_delete     lld_display   lld_high       lld_input
lld_lookup     lld_m_prep    lld_math       lld_p_prep
lld_read       lld_setdata   lld_showline   llh_add
llh_delete     llh_display   llh_high       llh_input
lh_lookup      llh_m_prep    llh_math       llh_p_prep
llh_read       llh_setdata   llh_update     mld_arr_count
mld_clear      mld_scroll    mlh_clear      llh_construct
mlh_cursor     mlh_init      mlh_key        ring_options
switchbox
```

Table Naming Conventions

The Code Generator imposes an additional restriction on table names. In addition to the database engine restriction that the first 8 characters of the table name be unique, the Code Generator requires that the last 6 characters be unique. When naming internal program record arrays and records the Code Generator uses the last 6 of the unique name to coin the internal program record and arrays. For example:

```
    strinvce becomes p_rinvce and m_rinvce records in globals.
```

The trouble starts when you have table names like "herinvce" and "derinvce" as a header and a detail table. The generator generates two records one that is not an array and one that is an array for the header and detail tables respectively with the same name.

A program generated with a header and detail record with the same name does not compile. This would create an array defined in globals for `m_rinvce` for the detail and also a `m_rinvce` record for the header.

Screen Form Style Guide

The following conventions have evolved from the development of standardized applications. Use this guide as a source of information for consistent and organized display of fields and sections on a screen form.

- The Screen or Form title should appear in the center of the first line of the screen. It should start with a capital letter, but should not be all capital letters.
- The title should not contain the word Screen or Form.
- The title should be a noun describing the document or the contents of the file.
- The title should not contain a verb unless the menu item that calls the screen only allows one possible action and other menu items allow different actions on the same file.
- Screens and menu items are named for the contents of the file or table they access, not a report or the file itself. For example, the screen should say, "Ledger Accounts" rather than "Chart of Accounts."
- When a file contains documents, they should be called by their most common name. For example: invoices, orders, checks, etc.
- Do not use the term "Document" as part of the name of a document. For example: use "Tax Definitions" or "Tax Codes," not "Tax Documents".
- If there is one document in a file, the file’s title should be singular. If there are more than one, it should be plural. For example, title multiple document files "Company Information," "Account Groups," and "Customers," not "Customer."
- If the file contains multiple items, but a description of its contents cannot be made plural, add the word, Items. For example: "Inventory Items," not "Inventory" or "Inventories."
- If the screen contains rows of detail, the detail should appear in the lower portion of the screen, separated from the "header" information by a line.
- If the screen contains totals from the detail, that information should appear below the detail in a special footer section separated by a line.

- Headings at the top of a screen form should be centered within a line of dashes which extends to the ends of the form, for example:

```
|                                                                             |
|------------------------------ Heading Here ---------------------------------|
|                                                                             |
```

not

```
|                                                                             |
|                          Heading Here                                       |
|-----------------------------------------------------------------------------|
|                                                                             |
```

- Sub-headings should be within a line that extends to three spaces from either side, for example:

```
|                                                                              |
|   --------------------------- Sub Heading -------------------------------    |
|                                                                              |
```

- Detail line areas should always have headings within such a line, for example.

```
|                                                                            |
| - Column - Column -------- Column Head ------ Column Head ---------------- |
|                                                                            |
```

- Detail heading lines should extend to one character from the form edge.
- Detail line sections should end with a solid line of dashes that extends to within one character space of the form’s edge.
- Subsection lines should never extend beyond the heading line, and should have a space under the first and last dash of the heading line.

For example, format this way:

```
-------------------------- Subsection -----------------------
  Cash Account Number: [    ]   Expense Account Number [    ]
```

However, avoid this:

```
------------------------ Subsection -----------------------
Cash Account Number: [    ]     Expense Account Number [    ]
```

- If a form has no detail section and has room for another line, there should be a solid line of dashes from left to right just above the comment line at the bottom of the form.
- Whenever possible, fields on a document header should appear one to a screen row.
- Fields read in from a detail table should appear in rows and columns.
- Use a "field descriptor" to indicate the contents of a field whenever possible. Descriptors for fields in a heading should appear before the field. Descriptors for fields arranged in columns and rows should appear at the top of the column.
- Descriptors should begin with a capital letter, but not appear in all capital letters.
- Descriptors for data entry fields in which the field follows to the right of the descriptor should be followed by a colon.
- Descriptors and headings should not be abbreviated unless this is required for spacing. Abbreviations should be followed by a period.
- Do not use # to indicate numbers if No. fits. Do not use No. if Number fits.
- Data entry fields should be stacked, each starting in the same column. Their headings should be placed so that their ending colon is one character to the left of the beginning of the field.
- Display-only fields should not start in the same column as data entry fields. Display-only fields should form their own column, preferably to the right of the data entry fields.

- Row descriptors start, if the area is large enough, over the first character of the value in a text field. In a numeric column, they end over last character of data (right justified). If the column is narrower than the heading, the heading is centered. They are not followed by colons or other punctuation.
- Screens should not include technical information or terms of no interest to the users, such as the technical name of the program or of any files.

G

Termcaps

This section covers:

- Terminal options
- Termcaps
- Suggestions for debugging termcap problems

Terminal Options

Programs created with Fitrix *Screen* use key combinations which may conflict with the terminal options currently set on your system. These options can be remapped to other key combinations with the Unix `stty` command.

The most common terminal options that must be remapped are:

Terminal Option	Typical Default Key Combination
susp	[CTRL-Z]
dsusp	[CTRL-Y]

These options are remapped by the startup shell scripts for Fitrix *Screen*. They also may need to be remapped for programs created by the Fitrix *Screen*.

These terminal options can be remapped on most systems with the following command:

```
stty susp <SOME KEY>  dsusp <SOME KEY>
```

In the first shell script shown below, the susp and dsusp terminal options are both remapped to [CTRL] - [-] or control "-" with the command:

```
stty susp "^-" dsusp "^-"
```

The two shell scripts shown below are examples of how you might want to bootstrap your programs in a manner which will avoid `stty` terminal option conflicts.

Shell script example 1:

```
:
######################################################################
# Copyright (C) 1992 Your Company Name
# All rights reserved.
# Use, modification, duplication, and/or distribution of this
# software is limited by the software license agreement.
# Sccsid:  @(#)  /u/fourgen/bin/run  1.5  Delta: 4/16/92
######################################################################

# Usage: run dirname [args]

. fg.setshell

# Check for an argument
if test "$1" = ""
then
    echo "Syntax: run dirname [args]"
    exit 1
fi

cd $1
shift

progname=‘sed -n "s/^NAME *= *//p" Makefile‘

# Run executable.
if test -x $progname
then
    exec ./$progname "$@"
fi

# Run 4gi (RDS) if existing.
progname=‘echo $progname | sed -e s’,Ž4ge,Ž4gi,‘
if test -f $progname
then
    exec fglgo $progname "$@"
fi

# Can’t find the program to run
echo "Cannot find the program: $progname"
sleep 2
exit 100
```

Shell script example 2:

```
:
######################################################################
# Copyright (C) 1991
# Use, modification, duplication, and/or distribution of this
# software is limited to the terms of the software agreement.
# Sccsid:  @(#)  /usr/fourgen/work/bin/fg.setshell  1.14  Delta: 4/10/92
######################################################################

# The purpose of this shell script is to re-bootstrap the current
# program using a shell that’s known to work on this platform.
# Most platforms pass the QA suite using the regular bourne shell
# located in /bin/sh, but some platforms have shells that work
# better than the /bin/sh shell.

# This program is usually called from within other shell scripts using
# the following format:
#
#   # Make sure we’re using the correct shell
#       . fg.setshell
#

if test "$fg_shell" = ""
then
    # Get the unix version information
    unix_version=‘uname -a‘ 2>/dev/null

    # Some platforms require special shells
    case $unix_version in
        *HP-UX*)  fg_shell=/bin/ksh
                  {
                  stty susp ""
                  stty dsusp ""
                  }>/dev/null 2>/dev/null;;
        *ULTRIX*) fg_shell=/bin/sh5
                  stty old;;
        *AIX*)    fg_shell=/bin/sh
                  stty susp ""
                  stty dsusp ""
                  stty quit "";;
        *Sun*)  fg_shell=/bin/sh
                  {
                  stty susp ""
                  stty dsusp ""
                  stty lnext ""
                  stty rprnt ""
                  stty werase ""
                  stty flush ""
                  } 2>/dev/null;;
        *)        fg_shell=/bin/sh
                  {
                  stty susp ""
                  stty dsusp ""
                  }>/dev/null 2>/dev/null;;
    esac
    SHELL=$fg_shell
    export SHELL fg_shell

    # this followng line is here ONLY for 4.1 executable core-dump prob
    test "$TERM" || TERM=vt100;export TERM

    # this followng code is here ONLY for 4.1 executable core-dump prob
    if test "$TERMCAP" = ""
    then
        if test -f /etc/termcap
        then
            TERMCAP=/etc/termcap;export TERMCAP
        else
            TERMCAP=$INFORMIXDIR/etc/termcap;export TERMCAP
        fi
    fi

    # Re-boot ourselves in a shell that’s known to work
    thisprog=‘type $0 |
      sed -e  ’s,[()],,g’ -e ’s,Ž$,,’ -e ’s,..* /,/,’ -e ’s, .*,,’‘
    if test ! -f "$thisprog" -o "$thisprog" = ""
    then
        echo "ERROR in fg.setshell"
        exit 1
    fi
    exec $fg_shell $thisprog "$@"
else
    fg_shell=""
fi
```

Writing Termcap Entries

This section will help you decipher your termcap files and can help you debug your own termcaps. Should you need more information on termcaps and terminfo, refer to the appendix of your *INFORMIX-4GL Reference Manual.*

Termcap is short for "terminal capabilities," which are descriptions of the various features of a terminal, and instructions on how to use these features, all written in a rather cryptic language in a termcap file. The language which describes the terminal capabilities is interpreted by programs that use terminal I/O in order for the program to correctly control the terminal and interpret input from the keyboard.

The Termcap File

The termcap file, `/etc/termcap`, usually consists of several termcap entries, each one corresponding to a particular terminal or to a particular emulation mode on some terminal, or to a terminal being used in some special fashion. The rest of the termcap file, about 20%, consists of lines beginning with a #. These are comment lines and are generally less intelligible than the rest of the termcap file. These lines are to be mostly ignored. One termcap entry can be separated from another once you understand what an entry itself looks like.

The Termcap Entry

Each entry has the form `label[|label][:capability]:`. This means there are a string of labels by which the entry can be referred to, each separated from the next by a "|" symbol followed by a string of terminal capability codes each separated from the next by a ":". If the termcap entry is longer than a single line (almost all of the time it is) then the symbol "\" is used on the end of a line to indicate that the entry continues to the next line. An easy example of what two termcap entries might look like follows (easy because whoever edited them tried to make them easy to distinguish as entries):

```
n2|7901|NCR 7901:co#80:li#24:bs:am:cl=^L:\
  ti=\E0P^X^L:\te=^O^X\E0@:cm=\EY%+%+:ce=\
  EK:cd=\Ek:kh=^A:kl=^U:bc=^U:kr=^F:nd=^F:\
  ku=^Z:up=^Z:\kd=^J:do=^J:kb=^H:kc=^M:so=^N:\
  se=^O:sg#0:ul:us=\E0‘^N:ue=\E0@^O:ug#0:\
  NM=^O^X\E0@:NB=\E0B^N:NR=\E0P^N:\
  NS=\E0R^N:AL=\E0A^N:AB=\E0C^N:\
  AR=\E0Q^N:

n3|vwpt|viewpoint|ADDS Viewpoint:co#80:li#24:\
  bs:am:cl=^L:ti=\E0P^X^L:\te=^O^X\E0@:\
  cm=\EY%+ %:ce=\EK:cd=\Ek:kh=^A:kl=^U:\
  bc=^U:kr=^F:nd=^F:\ku=^Z:up=^Z:kd=^J:do=^J:\
  kb=^H:kc=^M:so=^N:se=^O:sg#0:ul:us=\E0‘^N:\
  ue=\E0@^O:ug#0:NM=^O^X\E0@:NB=\E0B^N:\
  NR=\E0P^N:NS=\E0R^N:AL=\E0A^N:\
  NR=\E0P^N:NS=\E0R^N:AL=\E0A^N:\
  AB=\E0C^N:AR=\E0Q^N:AS=\E0S^N:OV#0:\
  k1=\E1:k2=\E2:k3=\E3:k4=\E4:k5=\E5:\
  k6=\E6:k7=\E7:k8=\E8:k9=\E9:MP=\E0P^X^L:\
  MR=\E0@^X:NU=^N:EN=^V:CN=^X:CF=^W:
```

Not all termcap entries appear like the examples above. Sometimes they might look like the following:

```
n2|7901|NCR 7901:co#80:li#24:bs:am:cl=^L:ti=\
  E0P^X^L:te=^O^X\E0@:\cm=\EY%+ %+:ce=\
  EK:cd=\Ek:kh=^A:kl=^U:bc=^U:kr=^F:nd=^F:\
  ku=^Z:up=^Z:\kd=^J:do=^J:kb=^H:kc=^M:so=^N:\
  se=^O:sg#0:ul:us=\E0‘^N:ue=\E0@^O:ug#0:\
  NM=^O^X\E0@:NB=\E0B^N:NR=\E0P^N:NS=\
  E0R^N:AL=\E0A^N:AB=\E0C^N:AR=\E0Q^N:
n3|vwpt|viewpoint|ADDS Viewpoint:co#80:li#24:\
  bs:am:cl=^L:ti=\E0P^X^L:\   te=^O^X\E0@:cm=\
  EY%+ %+ :ce=\EK:cd=\Ek:kh=^A:kl=^U:bc=^U:\
  kr=^F:nd=^F:\ku=^Z:up=^Z:kd=^J:do=^J:kb=^H:\
  kc=^M:so=^N:se=^O:sg#0:ul:us=\E0‘^N:\ue=\
  E0@^O:ug#0:NM=^O^X\E0@:NB=\E0B^N:NR=\
  E0P^N:NS=\E0R^N:AL=\E0A^N:\AB=\E0C^N:\
  AR=\E0Q^N:AS=\E0S^N:OV#0:k1=\E1:k2=\E2:\
  k3=\E3:k4=\E4:k5=\E5:\k6=\E6:k7=\E7:k8=\E8:\
  k9=\E9:MP=\E0P^X^L:MR=\E0@^X:NU=^N:\
  EN=^V:CN=^X:CF=^W:
```

Obviously, it is harder to distinguish one entry from the next in the second example especially when you consider that there may be hundreds of entries formatted together like this.

Notice that the last line of the entry for `n2` does not end with a "\" but that every other line of the entry does end with "\". If the last line of the entry ended in "\" like the others then the entry for `n2` would continue into the entry for `n3`. Correct interpretation of the termcap entries relies heavily upon these *very* important "\" characters.

The Labels

The label part of the termcap entry is the mechanism by which a program can find the entry in the termcap file. Usually it consists of a two letter code, a short name or two, and a brief description of the terminal. The termcap entry can be identified by any of the labels in the label section of the entry and the identification is usually based upon the value of the system variable `$TERM` (use `echo $TERM` to see its current value). It is highly advisable to use one of the short names rather than the two character code for the value of `$TERM` since the two character code may not be unique and programs find the first occurrence of the label whether or not there is another—yours—further down in the file.

The Capability Codes

The terminal capabilities directly follow the labels in each termcap entry and each code is separated from the next by a ":" and has a two letter "name." There are three different sorts of codes used to identify a capability; a boolean type either the code is there or not with no specific value associated with it; a numeric type an integer value is assigned to the code; a string type a string of characters is assigned to the code.

The **boolean** type codes are used to identify the existence or lack of a certain terminal characteristic such as whether the cursor automatically wraps around the margins of the terminal. In the termcap entry they can be identified because they consist solely of the capability name (`am` for automatic margins).

The **numeric** type code is used to identify countable parameters associated with the terminal such as number of columns and number of rows on the screen. These codes have the form `codename#value` (i.e. number of columns would be `co#80` for an eighty column screen).

The **string** type is used to identify strings of characters sent by certain keys on the keyboard and strings needed by the terminal to perform certain actions such as positioning the cursor in a particular location on the screen. For example, to identify the character string sent from the keyboard by the up arrow key the code might read `ku=\E[1` and would indicate that the keyboard sends the character sequence: `ESC (octal 33) [ 1` when you press the up arrow key.

Special Characters

The character strings for a terminal capability often use such characters as \E, ^R, or ^A which represent ASCII [ESC], [CTRL]-[R], and [CTRL]-[A] respectively. In the termcap file however, these characters are never given in their literal form because they are generally non-printing characters. Therefore, in order to represent them in text they have a special form.

Ascii [ESC] is represented with \ and E (looks like "\E").

Ascii [CTRL] characters are represented with a ^ followed by the character in upper case, hence ^R and ^A.

Characters can also be represented by their octal (base eight) value in cases such as the ":" which is used to interpret the termcap file by separating arguments and can’t be included directly as part of a string (it would be interpreted as the end of the string). The octal code for a colon is \072.

The Codes

Unfortunately, there are too many terminal capability codes to list all of them here and many programs use special sets of codes in addition to the more or less "standard" set. Therefore, you should look in the system documentation for a thorough list of the various termcap codes and their functions. You must refer to your program documentation to find the codes for any special functions used by the program. Here is a list of some of the more common codes:

Code	Function
cm	control code for cursor positioning by row and column
ku	character sequence sent by the cursor up key
kd	character sequence sent by the cursor down key
kr	character sequence sent by the cursor right key
kl	character sequence sent by the cursor left key
kh	character sequence sent by the home key
k0-k9	character sequences sent by the function keys
ho	control sequence used to position the cursor at 0,0
do	control sequence used to move the cursor down a row
cr	character sequence sent by the enter/return key
nd	control sequence used to move the cursor back a column
up	control sequence used to move the cursor up a row
bt	control sequence used to back tab
bs	boolean code which indicates that backspace is ^H
am	boolean code which indicates margins are handled automatically
co	number of columns on the display
li	number of lines on the display
so	control sequence used to turn standout mode on
se	control sequence used to turn standout mode off
sg	number of characters of display required by the ’so’ string

Code	Function
cl	control sequence used to clear the display
ce	control sequence used to clear to the end of the line

The biggest problem with terminal capabilities is not how to read them but what the various codes mean for the various programs that use them. Unfortunately, the answer to that question often remains in the head of the author of that program and does not reach the users of the program nearly often enough, or in an intelligible form. The other complication is that terminal manufacturers seldom produce readable reference material for their own terminal's characteristics.

Ideally, with a combination of knowing how the program uses termcap and how the terminal behaves, one should always be able to fix or write a termcap entry for any program and terminal (or discover that the program cannot run at all on the terminal). The implementation of the capabilities on various terminals is anything but standard, but the interpretation and use of the codes by a program usually follows certain guidelines.

If a program does any full screen display and entry, if it highlights anything, ever, or if it just clears the screen, then it almost certainly uses termcap to decide how to do its various terminal-oriented tasks. other uses for the termcap entries are: to generate graphics, position the cursor on the screen, and to identify special input from the keyboard (keys with special meaning).

Interpretation and Action

Of the three types of terminal capabilities, the most heavily used and the most complicated are the string type. These, in turn, can be grouped into two classes of applications. The first class, which includes codes for cursor movement (ku, kd, kr, kl, kh, etc.) and many special program codes, is used only to identify keystrokes from the keyboard. For example, when you enter an "a" from the keyboard, you press the "a" key and only one character is sent to the computer; but other keys, such as the cursor keys often will send a sequence of characters to the computer. In order for programs that use special keys to correctly recognize keyboard entry, the program needs to know how to interpret the characters it is receiving from the keyboard.

With the aid of the termcap file, a program can recognize complex input and behave accordingly. The group of codes that is used for identifying input is one of the two classes of string type terminal capability codes.

The other class of string type codes is used for directly controlling the terminal screen. These codes include character sequences that invoke a graphics character set, or start a block of highlighted screen, or turn off the highlight or graphics, just to list a few. Others indicate the correct codes to send to the terminal, to move the cursor about the screen, or to enable and disable the terminal's auxiliary port. With a combination of these two classes of codes, a program can both interpret input from the keyboard and perform actions with the terminal such as complex graphics display.

Now you know why programs need termcap files and how they use them, as well as how to read them. The only steps left are testing, modifying, and writing these files. Testing a termcap is not simple because the termcap is only part of the terminal I/O system, any element of which can be to blame for weird or incorrect displays. However, once your terminal is working on the most primitive level (you get a login and can run most system commands without any problem) then specific program misbehavior usually can be attributed to problems with the termcap.

Most of the time, termcap difficulties are related to only a couple of errors in an existing termcap entry for the terminal. It is unusual to not be able to find a termcap entry that provides most of the features for your terminal simply by trying various values for the variable `$TERM` and using the different emulation modes available with many terminals. Even when you cannot find such a termcap entry, you can get a substantial head start on developing a new termcap entry by using an existing termcap entry for a similar terminal type.

When you are testing and modifying a termcap entry, it is usually best to make a temporary file that contains only that entry so that there is no danger of corrupting the other termcap entries. Then, in order to direct the system to use that file you can set the system variable `$TERMCAP` equal to the full path name of the temporary file and then export `$TERMCAP`. Once you have this special file setup you need the proper documentation in order to identify and correct problems within the termcap entry.

You will need the system documentation on the various termcap codes and program documentation if the programs you will be running require any special termcap entries. Also you must have technical documentation for the terminal and be sitting at the terminal ready to go. It helps to have a second terminal available which already has a functioning termcap so that you can edit files without having to rely on the terminal for which you are writing the termcap.

Finally, you must have the UNIX editor `vi` and/or the program `od` on your system in order to read the various character codes sent by the keyboard. With these tools (and some time) at hand you are ready to go.

Testing the Keys

The most reliable way to find what characters are being sent by the special keys on your keyboard is to directly collect and view the output of those keys in an uninterpreted form. `vi` and `od` both provide excellent ways to see what a key is sending from the keyboard (`vi` is better if your termcap is already good enough to support it).

To use `od` (od stands for octal dump) you simply type `od -bc [RETURN]` at the command line. At this point the program `od` is waiting for input from the keyboard. Now when you press a key followed by a [RETURN] and [CTRL]-[d] the character and octal representation of the characters sent by the key will be displayed in two rows. The first row is the character representation (if any) for each character and the second row is the octal value for each character (ignore the first string of digits on the first row).

By comparing the octal values with a table of ASCII characters, you can determine exactly which characters are being issued by the key. (`od` will be waiting for your next input followed by [RETURN] and [CTRL]-[d] but can be terminated by an additional [CTRL]-[d].)

To use `vi` instead of `od`, you can invoke `vi` without a file name, type i to get into insert mode, then type [CTRL]-[v] followed by the key you want to test followed by a return. The characters displayed on the screen are the ASCII representation for the characters sent by the key - ^[is [ESC], ^A is [CTRL]-[a], etc. To exit `vi` type `[ESC] :q!`.

Action Codes

Certain terminal functions require a control code sent by the program to the terminal which causes the terminal to perform the function desired. An example of this type of function is highlighting. In order for the terminal to begin highlighting a certain block of text, first the cursor must be positioned at the beginning of the block and then a code needs to be sent to the terminal to begin highlighting. Then a code needs to be sent after all the characters to be highlighted have been sent in order to stop highlighting. The program looks in the termcap for the string code cm to use for positioning the cursor on the screen at a certain row and column. Then it gets the string code so to turn highlighting (standout) on. Then it looks for the code se to turn highlighting off. If any of these codes are incorrect, the highlighting action will fail and may also wipe out the rest of the display.

There are many other control codes that are sent to the terminal which can disturb the display seriously if they are incorrect. These codes can only be found in the technical manual for the terminal. For example, so must be set to the manual entry for "start standout mode" and cm must be set to the manual entry for "direct cursor addressing (cursor movement)." (for cm only, there are additional characters explained in the system documentation that refer to the format for cursor and row numbers required as a variable part of the cm string.)

By using the terminal manual for the control strings and the system documentation for the termcap codes, you should be able to fix and add needed controls (not all terminals will have all of the possible capabilities). Often, but not always, there will be an appendix in the technical documentation for the terminal which gives the control codes for the available emulation modes for each terminal function. This is often the only place in the documentation where the codes are explicitly given in an understandable form.

The Other Codes

The two classes of string type codes have been discussed so far. By comparison the other codes are simple to understand and work with. The boolean type are either in the termcap or not and indicate the existence of a particular terminal characteristic. There are only two common numeric type entries, `li` for number of rows and `co` for number of columns.

Observations

Things to keep in mind when working with termcaps:

1. It takes time to eliminate all of the possible bugs from a termcap but often the solutions are simple. The only way to effectively work with termcaps is with a substantial dose of patience.
2. Usually the best way to start solving a termcap problem is not with the termcap. Make sure that `$TERM` is set to a label in the label string and that the label is unique.
3. Always double check your terminal setup and emulation mode before working on a termcap for that terminal.
4. Before changing a termcap make a copy, and then use a fine tooth comb for syntax errors. All labels should be separated by "|", all entries separated by ":", all but the last line ends in "\".
5. A lot of work can be saved by using at least part of an existing termcap.
6. No termcap entry (all of the labels and codes combined) can be over 1024 characters long. Usually the entries after the 1024th character will be ignored.

Index