Fitrix

Visual Development Tool (VDT)
Screens And Menus
Course Workbook
4.12

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS252 227-7013 Fourth Generation Software Solutions, 2814 Spring Rd , Suite 300, Atlanta, GA 30039

Copyright

Software License Notice

Your license agreement with Fourth Generation Software Solutions, which is included with the product, specifies the permitted and prohibited uses of the product Any unauthorized duplication or use of Fitrix, in whole or in part, in print, or in any other storage and retrieval system is forbidden

Licenses and Trademarks

Fitrix is a registered trademark of Fourth Generation Software Solutions
Informix is a registered trademark of Informix Software, Inc
UNIX is a registered trademark of AT&T

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

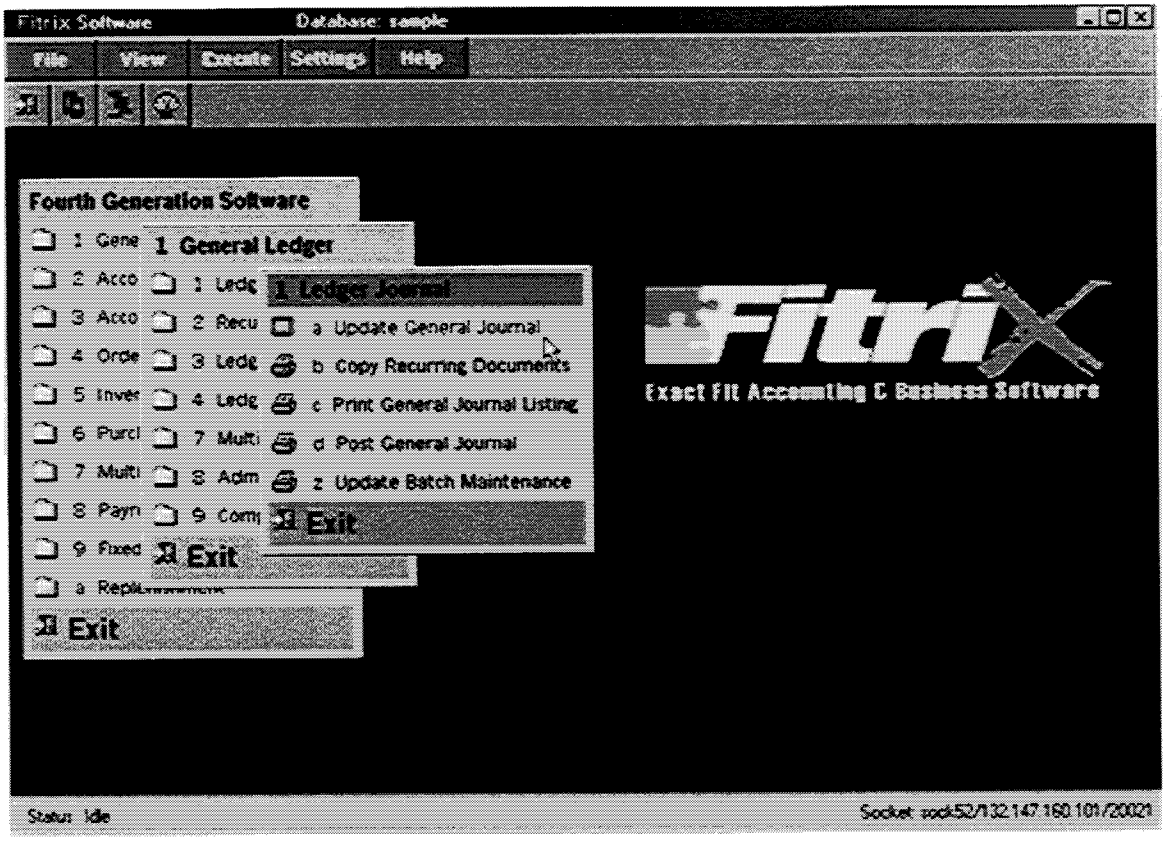
Corporate: (770) 432-7623
Fax: (770) 432-3448
E-mail: info@fitrix com

Copyright

Welcome to the Fitrix , Visual Development Tool (VDT) Screen and Menu Course Workbook. This manual is designed for use in the Fitrix VDT Training class. We hope that you find all of this information clear and useful. Although the pictures in this manual are all of character based screens, please keep in mind that any program created by the Visual Development Tool offers the option of being viewed in a graphic based Windows screen.

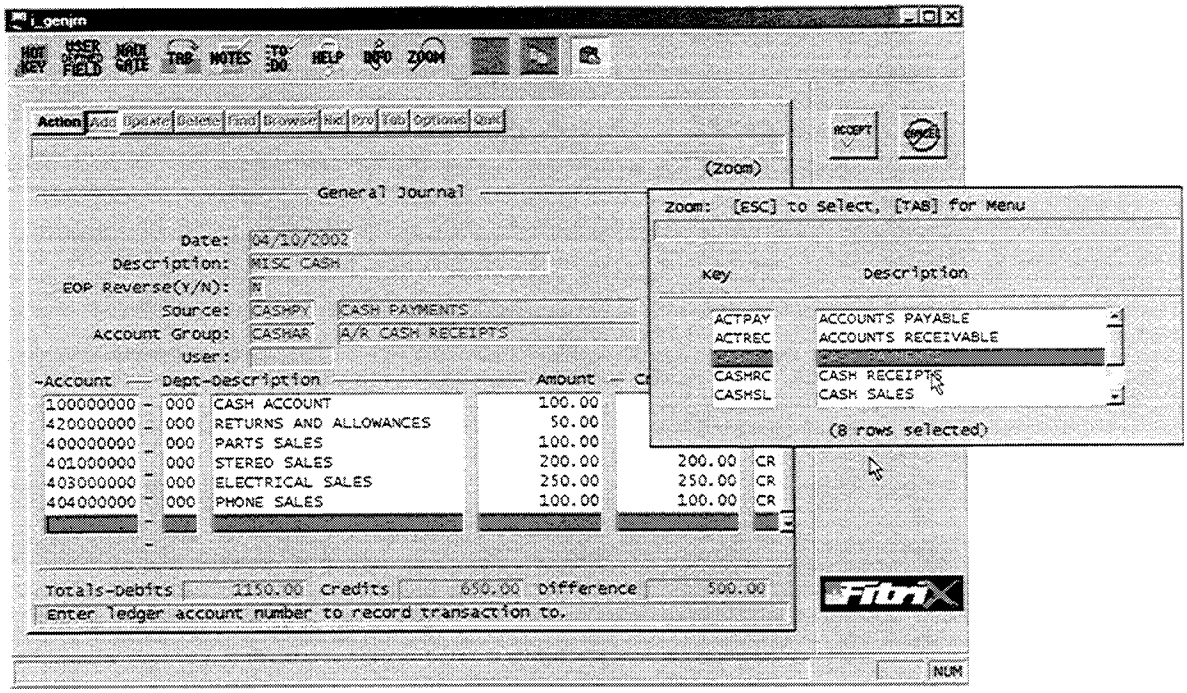
The Visual Development Tool itself runs only in character mode, but any program created with VDT can be viewed in graphical mode using MS windows as well as character.

Examples of graphic based product viewing modes are shown below in Example 1 and Example 2.



Example 1· Menu Graphical Windows Mode

Here is another example.



Example 2: Data Entry Screen in Graphical Windows Mode

Displaying our products in graphic mode, as shown in Example 1 and Example 2, is customary for many Fourth Generation product users. However, your viewing mode is a user preference. Changing from character based to graphical based is a user 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 help desk 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, your Fitrix serial number and your exact question.

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

Table of Contents

Section 1: Using a Generated Input Program

Section 2: Getting Started with the Form Painter

Section 3: Working with the Database

Section 4: Creating Zooms

Section 5: Creating Lookups

Section 6: Input Areas and Specification Files

Section 7: Working with the User Control Libraries

Section 8: Using the Screen Code Generator

Section 9: Creating Triggers

Section 10: Managing Screen to Table Flow

Section 11: Screen Handling and Add-on Headers

Section 12: Working with Switchboxes

Section 13: Working with Program Events

Section 14: Creating Pop-Up Menus

Section 15: Creating Extension Screens

Section 16: Version Control and Conventions

Section 17: Compiling Generated Code

Section 18: Using the Featurizer

Section 19: Getting Started with Menus

Section 20: Building a Menuing System

Section 21: Security

1

Using a Generated Input Program

Main topics:

- Setting Environment Variables
- Using a Generated Input Program
- Using the Ring Menu Commands
- Accessing Zooms
- Using AutoZooms
- Using Lookups

Setting Environment Variables

In order to create and run programs with *Screen*, you must set certain UNIX environment variables and export them.

1. The `$fg` variable should point to the directory where your *Screen* product is installed. For example, the following command sets `$fg` to the `/usr/` directory:

   ```
   fg=/usr/ ; export fg
   ```

2. The `$INFORMIXDIR` variable should point to your `informix` directory. For example, the following command sets `$INFORMIXDIR` to the `/usr/informix` directory:

   ```
   INFORMIXDIR=/usr/informix; export INFORMIXDIR
   ```

3. The `$PATH` variable should include both `$fg/bin` and `$INFORMIXDIR/bin` directories:

   ```
   $fg/bin

   $INFORMIXDIR/bin
   ```

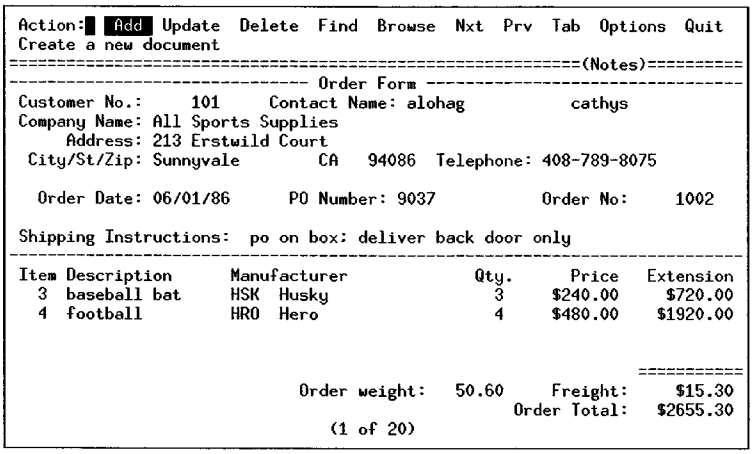
4. The `$DBPATH` variable must include two additional `$fg` directories:

   ```
   $fg/lib/forms

   $fg/codegen/data
   ```

Using a Generated Input Program

FourGen *Screen* lets you create sophisticated input programs. The following figure illustrates an input program built by FourGen *Screen*.

You can create sophisticated input programs with FourGen *Screen*.

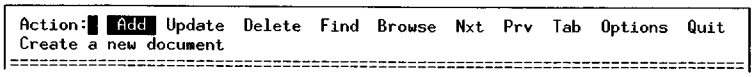


All input programs contain a ring menu interface located at the top of the screen.

Using the Ring Menu Commands

The ring menu consists of 10 commands. You can activate a command by highlighting the command and pressing [ENTER] or by typing the first letter of the command name.

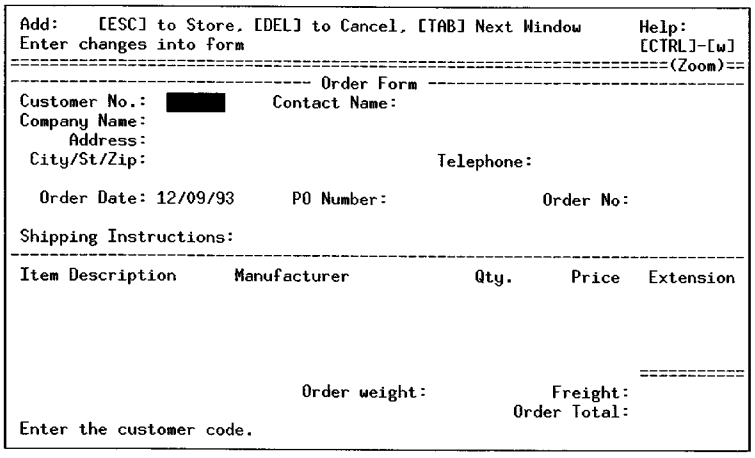
The ring menu consists of 10 ring menu commands.



The Add Command

The Add command lets you add a record (or *document*). Use Add when you want to create a new entry in your input program. When you select Add, your cursor moves to the first input field on the form.

The Add command lets you add a record.



The Update Command

The Update command lets you change a value in an existing document. Use Update when you want to alter or correct an input field value. Before you can update a document, you must use the Find command to select it (see "The Find Command" on page 1-6).

The Update command lets you change a value in an existing document.

```
Update: [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                        [CTRL]-[w]
=============================================================(Notes)==(Zoom)==
------------------------------ Order Form -------------------------------
Customer No.:  101     Contact Name: alohag          cathys
Company Name: All Sports Supplies
     Address: 213 Erstwild Court
 City/St/Zip: Sunnyvale        CA   94086  Telephone: 408-789-8075

  Order Date: 06/01/86      PO Number: 9037          Order No:     1002

Shipping Instructions:  po on box: deliver back door only
-------------------------------------------------------------------------
Item Description      Manufacturer             Qty.      Price   Extension
  3  baseball bat     HSK  Husky                 3     $240.00    $720.00
  4  football         HRO  Hero                 4     $480.00   $1920.00


                                                                ==========
                            Order weight:   50.60     Freight:     $15.30
                                                  Order Total:   $2655.30
Enter the customer code.
```

The Delete Command

The Delete command lets you remove an existing document. Use Delete to erase a document. Before a document is deleted, a prompt appears to confirm the deletion.

The Delete command lets you remove an existing document.

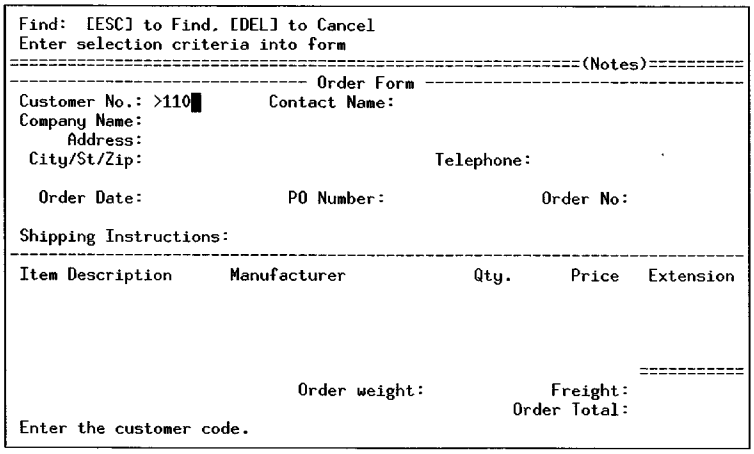
```
Delete:  Verify document deletion
Erase this document? (Y/N) ▌
============================================================(Notes)==========
------------------------------ Order Form ----------------------------------
Customer No.:     101     Contact Name: alohag          cathys
Company Name: All Sports Supplies
     Address: 213 Erstwild Court
 City/St/Zip: Sunnyvale        CA   94086  Telephone: 408-789-8075

  Order Date: 06/01/86      PO Number: 9037          Order No:     1002

Shipping Instructions:  po on box; deliver back door only
-------------------------------------------------------------------------------
Item Description      Manufacturer            Qty.      Price   Extension
  3  baseball bat     HSK  Husky                3     $240.00     $720.00
  4  football         HRO  Hero                 4     $480.00    $1920.00


                                                                ===========
                           Order weight:   50.60     Freight:      $15.30
                                                 Order Total:    $2655.30
                              (1 of 1)
```

The Find Command

The Find command lets you select a single document or a group of documents. Use Find to retrieve a document that you want to update or delete. When you select Find, the cursor moves to the first field of a blank form. To specify which document you want to select, you can enter *selection criteria* into the fields on the blank form. This ability is known as Query-By-Example (QBE).

For example, to select all the documents with values in the Customer No. field greater than 110, enter:

The Find command lets you select a single document or a group of documents. When you select Find, you initiate a Query-By-Example search.

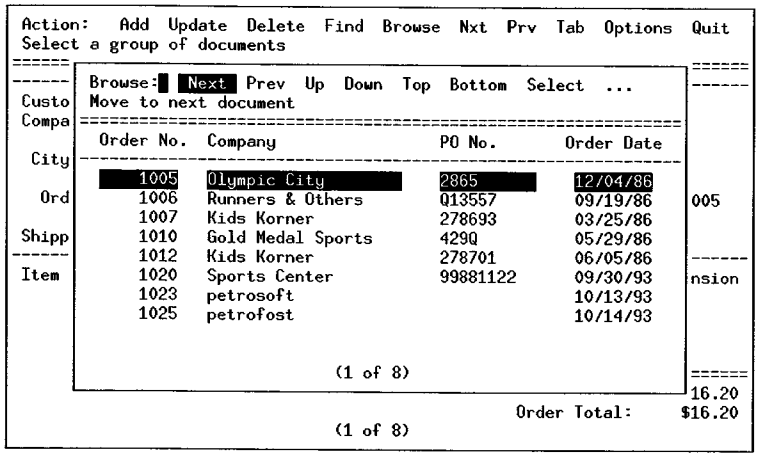


The Browse Command

The Browse command lets you view selected documents in a line-by-line format. Use Browse to get an overall view of the documents you have selected with the Find command (see "The Find Command" on page 1-6). Browse is useful because, although Find lets you select a group of documents, only one document is visible (or current) on the form at a time. Browse lets you see all the selected documents.

When you select Browse, a pop-up window appears showing all the selected documents in a line-by-line format.

The Browse command lets you view selected documents in a line-by-line format.



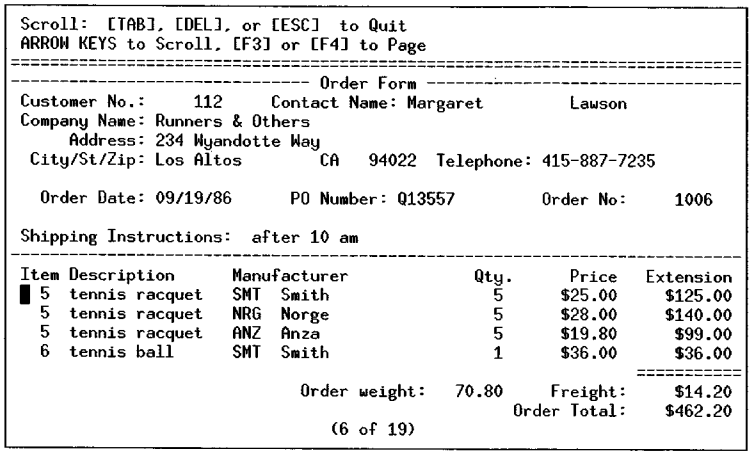
The Nxt and Prv Commands

The Nxt and Prv commands let you page through a group of selected documents. Before you can use Nxt or Prv, you must use Find to select a group of documents (see "The Find Command" on page 1-6).

The Tab Command

The Tab command lets you view lines on the scrolling portion of a form. Tab is for viewing only—you cannot add or update lines. When you select Tab, your cursor moves to the first line of the scrolling section. You must use Find to select a document before you can use the Tab command (see "The Find Command" on page 1-6).

The Tab command lets you view lines on the scrolling portion of the form.



The scrolling section is also called the *detail* section of the form. It represents the *many* side of a *one-to-many* table relationship. You can use the arrow keys or [F3] and [F4] to scroll through the detail lines.

The non-scrolling section of the screen is known as the *header* section. It represents the *one* side of a one-to-many table relationship. Typically, the header section is on the upper half of the input program and the detail section is on the lower half.

The Options Command

The Options command gives you a place to add your own custom ring menu commands. Use Options for custom ring menu items. For example, under Options, you could add a command that initiates your E-mail program. When you select Options, the ring menu clears and displays your custom ring menu items.

The Options command gives you a place to add your own custom ring menu commands.

```
Options: Quit
Return to the main menu
================================================================================
------------------------------ Order Form --------------------------------------
Customer No.:     112     Contact Name: Margaret         Lawson
Company Name: Runners & Others
     Address: 234 Wyandotte Way
 City/St/Zip: Los Altos        CA   94022  Telephone: 415-887-7235

  Order Date: 09/19/86      PO Number: Q13557         Order No:     1006

Shipping Instructions:  after 10 am
--------------------------------------------------------------------------------
Item Description      Manufacturer              Qty.      Price   Extension
  5  tennis racquet   SMT  Smith                 5      $25.00     $125.00
  5  tennis racquet   NRG  Norge                 5      $28.00     $140.00
  5  tennis racquet   ANZ  Anza                  5      $19.80      $99.00
  6  tennis ball      SMT  Smith                 1      $36.00      $36.00
                                                                ==========
                            Order weight:   70.80     Freight:     $14.20
                                                  Order Total:   $462.20
                               (6 of 19)
```

By default, Options always contains a Quit command that returns you to the main ring menu.

The Quit Command

The Quit command exits the program. Use Quit when you are finished using the input program. When you select Quit, the program stops and you are returned to the point at which you began the program.

The Quit command exits the program.

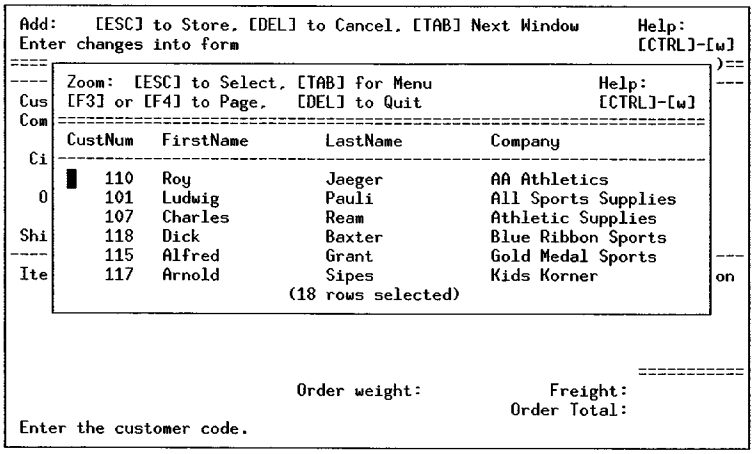
```
Action: Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
End the program
================================================================================
```

Accessing Zooms

Zoom screens help the user enter data. When entering values in fields, the user can Zoom into a list of valid values for that field and select one. Users invoke Zooms by pressing [CTRL]-[z] in a field. Not all fields have Zooms attached to them.

Zoom screens help users enter valid values.

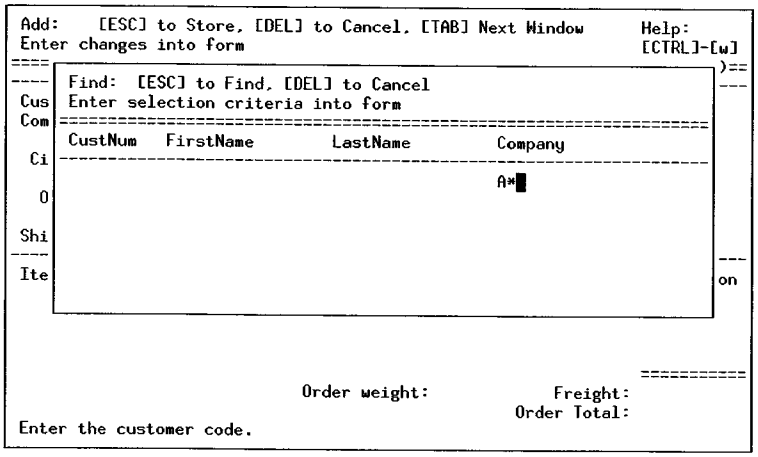
In this example, the user initiates a Zoom from the Customer No. field.



Zooms also use filters before returning values. If there are many valid values that can go into a field, Zooms, by default, first display a selection criteria screen. The selection criteria screen allows users to limit which values the Zoom returns.

Zooms can filter values before they are returned.

In this example, the user wants to see a list of companies that begin with the letter A.

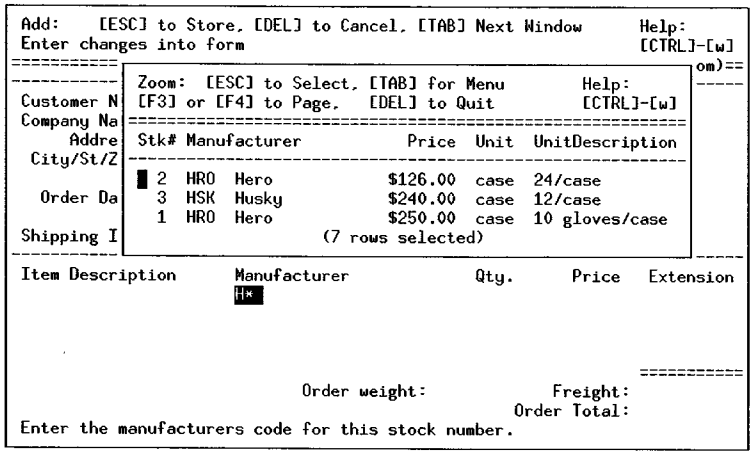


Using AutoZooms

You can also invoke a Zoom without pressing [CTRL]-[z]. If you place an asterisk in a field and press [ENTER], the Zoom is performed for you. You can combine the asterisk with letters to filter the Zoom.

AutoZooms let you enter selection criteria directly into a field.

In this example, the AutoZoom returns values that begin with H.



Using Lookups

When a user enters a value in a field, Lookups can be defined to pull related data into adjacent fields. Lookups also ensure that the user only enters valid values into a field.

Lookups pull related data into adjacent fields.

In this example, the Lookup pulled in values for the Contact Name, Company Name, Address, City/St./Zip, and Telephone fields. This Lookup is based on the Customer No. field.

```
Add:    [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                         [CTRL]-[w]
================================================================================
------------------------------ Order Form --------------------------------------
Customer No.:     102     Contact Name: Carole          Sadler
Company Name: Sports Spot
     Address: 785 Geary St
 City/St/Zip: San Francisco    CA   94117  Telephone: 415-822-1289

  Order Date: 11/20/92      PO Number:               Order No:

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




                                                                ===========
                           Order weight:           Freight:
                                                Order Total:
Enter the order date.
```

Section Summary

- In order to create and run *Screen* programs, you must set certain UNIX environment variables and export them.
- Using *Screen*, you can create sophisticated input programs.
- The topmost portion of an input program is known as the ring menu. The ring menu contains 10 ring menu commands.
- Most input programs use zoom screens to assist in data entry. Zooms perform data selection and validation tasks.
- You can access a Zoom by pressing [CTRL]-[z].
- AutoZooms let you place selection criteria directly into an input field.
- Lookups pull related data into adjacent fields. For example, when a user enters a number into the Customer No. field, the Contact Name, Company Name, Address, City/St./Zip, and Telephone fields get filled automatically.

Exercise 1A

Objective: To set up your development environment.

Use the Bourne Shell

For all the exercises in this book, you should be using the UNIX Bourne shell. Other shells, such as the C and Korn shells, use a different method for setting variables. If you are not using the Bourne shell, you should switch to it now.

- **At the UNIX prompt, enter:**

  ```
  /bin/sh
  ```

  A dollar sign ($) prompt appears. This prompt indicates that you are in the Bourne shell.

Check Your Current Environment Variable Settings

The `env` command displays current environment variable settings.

- **At the UNIX prompt, enter:**

  ```
  env
  ```

Use the `env` command to see what values the following environment variables contain:

```
fg
INFORMIXDIR
PATH
DBPATH
```

You must set the above environment variables to the appropriate values prior to using *Screen*.

To show the value in a single environment variable, you can use the `echo` command.

- **At the UNIX prompt, type:**

  ```
  echo $fg
  ```

Set Your Environment

Each of the environment variables shown on the previous page must point to a specific directory, depending upon how your system is set up. Here is a rundown of the correct variable settings:

fg	This variable should point to the directory where the Screen product is installed. For example, `$fg=/usr/fourgen`.
INFORMIXDIR	This variable should point to the directory where your Informix product is installed. For example, `$INFORMIXDIR=/usr/informix`.
PATH	This variable should contain both the `$fg/bin` and `$INFORMIXDIR/bin` directories.
DBPATH	This variable should contain `$fg/lib/forms` and `$fg/codegen/data`.

Note The dollar sign ($) before the environment variable indicates that you want to display the value contained within the variable.

You must issue two commands to set an environment variable. First enter the variable name followed by an equals sign and the value the variable should contain. Second, "export" the variable. For example, to set the `$fg` variable.

- **At the UNIX prompt, type:**

```
fg=/usr/        ; export fg
```

Use the echo command again to check the variable:

- **At the prompt, type:**

```
echo $fg
```

Exercise 1B

Objective: To become familiar with the screen demo programs.

List the Screen Demonstration Programs

- **At the UNIX prompt, type:**

  `scr_demo`

  The following list appears:

```
syntax: /usr/       /work/bin/scr_demo [12356789]
 1 - Header only screen demo
 2 - Header/Detail screen demo
 3 - Header/Detail demo with zoom, lookup, math, etc
 5 - Header/Detail demo with Add-On Header
 6 - Featurizer demo with Add-On Header
 7 - Header/Detail demo with Extension Screens
 8 - Header with Add-On Detail
 9 - Header with View-Detail, View-Header, and Query
```

Start scr_demo 5

- **At the UNIX prompt, type:**

  `scr_demo 5`

  When you start `scr_demo 5`, the following message appears:

```
Please wait...preparing Screen Demo 5

You have been placed into:
/usr2/       /codegen/demo.4gm/screen5.4gs.

Directory listing:
browse.per   cust.trg     order.per    screen5.bak  stockzm.per
cust.per     cust_zm.per  order.trg    stk_mnu.per

A new shell has been opened.
To exit the demo, type [CTRL]-[d]
```

  In addition, your prompt changes to reflect the demo program:

```
Screen Demo 5 ->
```

The screen demonstration programs give you a fresh set of form specification (*.per), trigger (*.trg) extension (*.ext), and feature set (*.set) files. These files supply the *Screen* Code Generator with instructions for building an input program.

Note

Some screen demonstrations contain all of these files while others only contain form specification (*.per) files. At this point, you do not have to know or understand what these files do. Just realize that they are used by FourGen Screen to create an input program.

Also realize that each time you run a screen demonstration program, you receive a fresh set of files. Because of this fact, do not be afraid to "break" the screen demonstration. If a file is corrupted, just start over.

Once you receive the Screen Demo prompt, you can use *Screen* to build and run an input program. In general, the following steps are required:

1. Run the *Screen* Code Generator to create source code.
2. Run `fg.make`, the compilation program to compile the source code and build a runnable program file.
3. Run the resulting program file.

Exercise 1C

Objective: To convert the initial `scr_demo 5` files into a program.

Start the Screen Code Generator

- **At the Screen Demo prompt, enter:**

  `fg.screen`

This command starts the *Screen* Code Generator, which reads the form specification (*.per) files in the demo directory and creates 4GL source code based on these files. As the *Screen* Code Generator works, multiple lines of code scroll past your screen.

List the Generated Files

When the *Screen* Code Generator finishes creating code, the Screen Demo prompt reappears. You can use the `ls` command to see a listing of the files the *Screen* Code Generator creates.

- **At the Screen Demo prompt, type:**

  `ls -C`

  The following list of files appears:

```
Makefile      cust.per      errlog        midlevel.4gl  stk_mnu.per
Makefile.org  cust.trg      globals.4gl   midlevel.org  stockzm.4gl
browse.4gl    cust_zm.4gl   globals.org   order.per     stockzm.org
browse.org    cust_zm.org   header.4gl    order.trg     stockzm.per
browse.per    cust_zm.per   header.org    screen5.bak
cust.4gl      detail.4gl    main.4gl      stk_mnu.4gl
cust.org      detail.org    main.org      stk_mnu.org
```

As you can see the *Screen* Code Generator creates several source code (*.4gl) files. From these files, the compilation utility (`fg.make`) builds a runnable program file.

After you use the *Screen* Code Generator to create 4GL source code, you can use the `fg.make` command to compile the source code and build a runnable program file. The `fg.make` command automatically determines the type of Informix development system you are using (either the 4GL or RD) and creates the appropriate program file.

Note If you are using the INFORMIX-4GL, `fg.make` creates a program file with a *.4ge extension. If you are using INFORMIX-RDS, `fg.make` creates a pseudo-code file with a *.4gi extension is created.

Start the Compilation Utility

- **At the Screen Demo prompt, enter:**

  `fg.make`

  This command performs several tasks, most of which are described in later chapters. For now, you should simply realize that it builds a runnable program file.

List Your Program File

When the `fg.make` command finishes, the Screen Demo prompt reappears. Again, you can use the `ls` command to display the files created by `fg.make`. Depending on your development system, you should see either a *.4ge or *.4gi program file.

- **At the Screen Demo prompt, type:**

  `ls -c`

  In the file listing, you should either have a `screen5.4ge` or a `screen5.4gi` file.

Start the Input Program

There are two methods for starting an input program. Once again you must choose the method appropriate for your development system.

- **Start the program:**

```
fglgo screen5.4gi
```

Once you issue the appropriate command, the screen demo 5 input program begins:

Once you issue the appropriate command, the demonstration program begins.

This figure shows the main screen of scr_demo 5.

```
Action:[ Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Create a new document
=============================================================(Notes)==========
------------------------------ 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  HSK  Husky                 2     $800.00    $1600.00
  2  baseball         HRO  Hero                   3     $126.00     $378.00
  3  baseball bat     HSK  Husky                 3     $240.00     $720.00
                                                  3                 $0.00
                                                                  ==========
                              Order weight:   20.60     Freight:     $20.00
                                                    Order Total:   $2718.00
                                 (1 of 16)
```

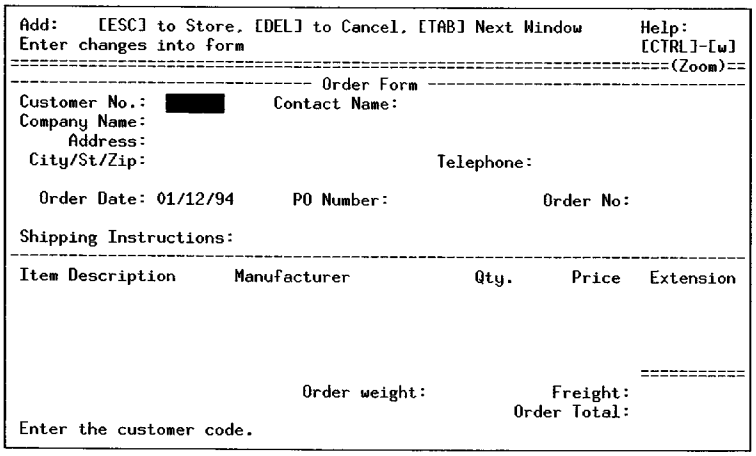
Exercise 1D

Objective: To become familiar with the input program functionality.

Add a Record

1. **Select Add from the ring menu.**

   A new record is created and your cursor moves to the first field:



   On some fields, the (Zoom) lamp appears. It indicates that a reference table exists for the field. You can press [CTRL]-[z] to open the reference table and select a value for the field.

2. **Fill in the input fields.**

3. **Press [ESC] to store the record.**

Find a Record

The Find ring menu command lets you select a single record, a group of related records, or all the available records.

1. **Select Find from the ring menu.**

   A blank record appears and your cursor moves to the first field:

```
Find:  [ESC] to Find, [DEL] to Cancel
Enter selection criteria into form
=============================================================(Notes)==========
------------------------------ 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:
                                                Order Total:
Enter the customer code.
```

2. **Press [ESC].**

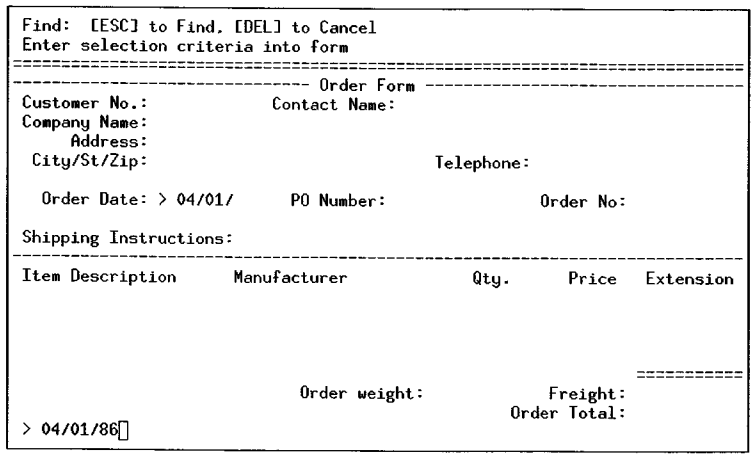
   All the records in your database table get returned. The first record appears in your main screen. You can use Nxt and Prv commands to scroll through the entire list.

To limit a Find to a single record or a group of related records, you can enter selection criteria in the fields. This ability is know as Query-By-Example (QBE). For instance, to select all the records that have order dates greater than 04/1/86:

1. **Select Find from the ring menu.**
2. **Move your cursor to the Order Date field and enter:**

   ```
   > 04/01/86
   ```

Since this value is larger than the input field, the selection criteria is displayed at the bottom of the screen:



3. **Press [ESC].**

All the records older than 04/01/86 are returned. Again you can use Nxt and Prv to scroll through the list of records.

Update a Record

The Update command lets you alter the values in a record.

1. **Use Find to select the record you want to update.**
2. **Select Update.**

   Your cursor moves to the first input field.

```
Update: [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                         [CTRL]-[w]
=============================================================(Notes)==(Zoom)==
------------------------------ Order Form ---------------------------------
Customer No.:  101      Contact Name:
Company Name:
     Address:
 City/St/Zip:                              Telephone:

  Order Date: 06/01/86      PO Number: 9270          Order No:     1002

Shipping Instructions:  po on box; deliver back door only
-------------------------------------------------------------------------------
Item Description      Manufacturer              Qty.      Price   Extension
  4  football         HSK  Husky                  1     $960.00     $960.00
  3  baseball bat     HSK  Husky                  1     $240.00     $240.00


                                                                  ==========
                              Order weight:   50.60     Freight:     $15.30
                                                    Order Total:   $1215.30
Enter the customer code.
```

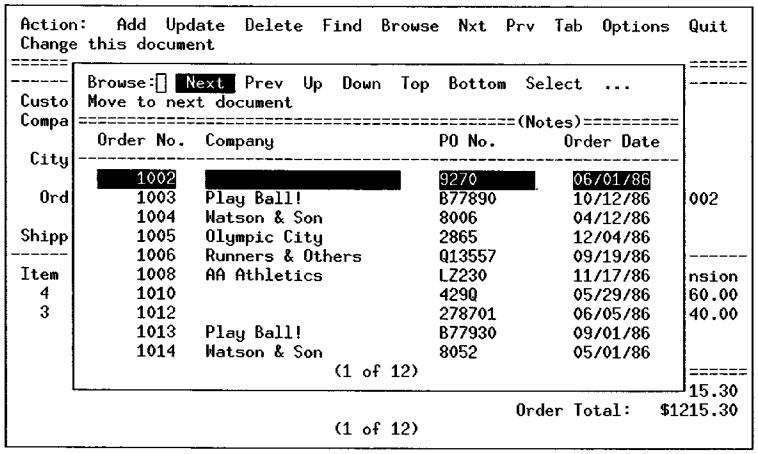
3. **Move to the field that you want to change and change its value.**
4. **Press [ESC] to store your change.**

Browse a List of Records

The Browse command lets you view a list of selected records in line-by-line format.

1. **Use the Find command to select a group of records.**
2. **Select the Browse command from the ring menu.**

   A secondary window appears showing the selected records in a line-by-line format:



3. **Use the ring menu commands on the browse window to scroll and select a record.**

Quit the Input Program and Screen Demo

Once you are done exploring the input program, select Quit from the ring menu. The Quit command returns you to the Screen Demo prompt.

- **At the Screen Demo prompt, type [CTRL]-[d] or enter:**

  `exit`

2

Getting Started with the Form Painter

Main topics:

- Form Painter Overview
- Starting the Form Painter
- Using the Form Painter Pull-Down Menus
- Creating a Form Image
- Converting Forms into Input Programs

Form Painter Overview

The Form Painter lets you develop complete data-entry programs written in INFORMIX-4GL. It is an interactive visual front end featuring a full screen editor, a database administration facility, and a screen enhancement builder. The Form Painter acts as the control center for running the *Screen* Code Generator and compilation utility. From within the Form Painter you can:

- Paint a form image, which can be directly converted into an input program.
- Access the database to add, delete, and update tables and columns.
- Store form image information in ASCII files (form specification *.per files), which are compliant with Informix's Perform format and easily moved to other systems.
- Create custom program events that are called from logical *trigger* points within the generated code.
- Copy and move any element of the form image.
- Store form image blocks on a Clipboard.
- Define data-entry areas and how they join with other data entry areas.
- Define how forms work with other forms.
- Specify the order in which input fields are processed on the form.
- Generate default form images with the AutoForm feature.
- Access other programs and tools on the system without leaving the Form Painter.

Starting the Form Painter

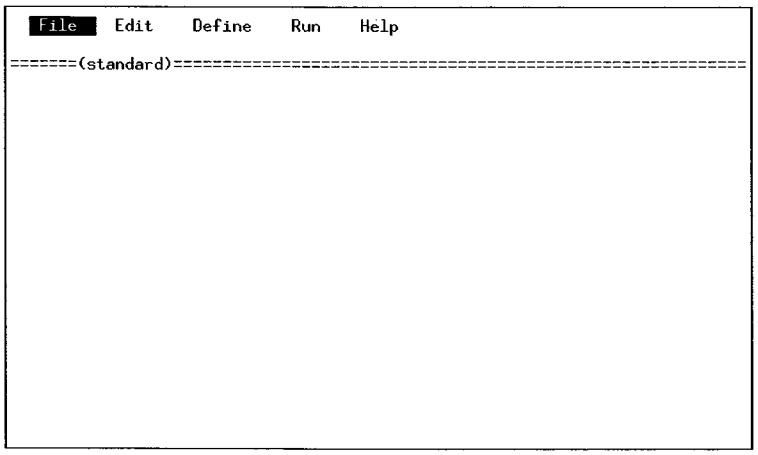
You can start the Form Painter using the fg.form command. This command has the following syntax:

fg.form -dbname *database*

Where *database* is the name of the database you want to use.

After you type this command, the Form Painter loads and displays the following window to your screen:

The Form Painter consists of two sections: the pull-down menus and the Form Editor.



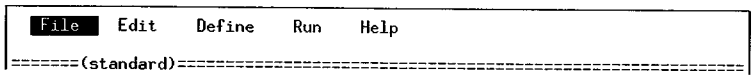
You should always start the Form Painter from the directory in which you want to read and write form specification (*.per) files.

Analogous to generated input programs, the Form Painter consists of two sections: the pull-down menus and the Form Editor.

Using the Form Painter Pull-Down Menus

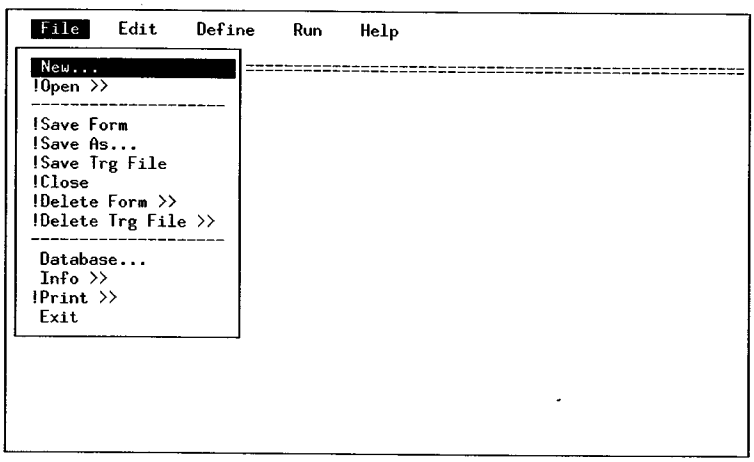
The Form Painter contains five pull-down menus.

The Form Painter contains five pull-down menus.



You can open a pull-down menu by highlighting it and pressing [ENTER]. You can also open a pull-down menu by typing the first character of the menu name (e.g., type F to open the File pull-down menu). Each pull-down menu contains a number of menu options. You select a menu option by highlighting it and pressing [ENTER]

You can open a pull-down menu by highlighting it and pressing [ENTER]. You can also open a pull-down menu by typing the first character of the menu name (e.g., type F to open the File pull-down menu).



Options preceded by an exclamation point (!) are not available. Options followed by greater-than signs (>>) open another menu with additional options. Options followed by an ellipsis (...) open a subsequent window.

Creating a Form Image

The Form Painter lets you paint form images. You can use the Form Painter to create a new form image or you can open existing form images. A form image graphically represents how your form will look and work once it is built. You paint and edit form images from within the Form Editor. The general steps for creating a new form image are as follows:

1. Select New from the File pull-down menu.
2. Enter a name for the new form.
3. Select the screen type you want to use.

In all there are ten screen types you can build. Your main screen is either a header or header/detail screen. The other screens act as secondary screens, some of which you can connect to the main screen (see "Using Different Screen Types" on page 11-2).

Screen Type	Function
`header`	Writes to a single database table.
`header/detail`	Writes to a header table and a detail table.
`add-on header`	Writes to a peripheral table from the main screen.
`add-on detail`	Writes to an additional scrolling detail table from the main screen.
`extension`	Writes to additional columns within the main header table.
`zoom`	Selects valid values for an input field.
`browse`	Lists documents in a line-by-line format.
`query`	Generates a selection prompt for use with report programs.
`view-header`	Allows you to view data from a peripheral header table.
`view-detail`	Allows you to view data from a subsequent scrolling detail table.

Painting the Form Image

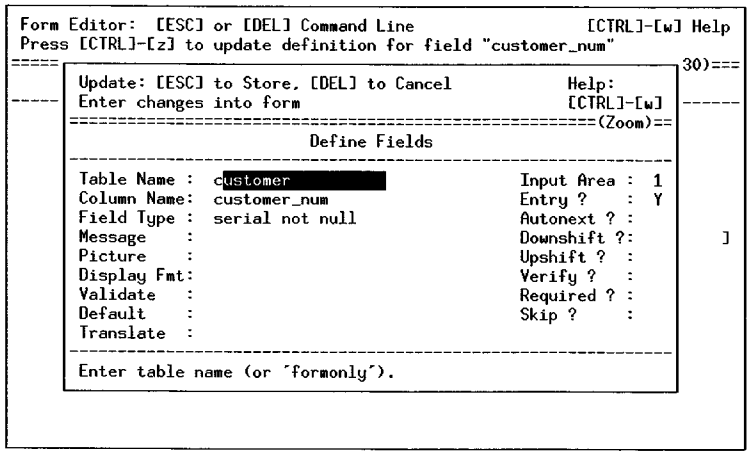
Once you load a form into the Form Editor, you can start painting the form image. Form images contain both text and input field definitions. The Form Editor provides several editing keys.

Keystroke	Use
[CTRL]-[a]	Toggles between insert and overstrike mode.
[CTRL]-[x]	Deletes a character.
[CTRL]-[d]	Deletes to the end of a line.
[CTRL]-[u]	Undoes an edit.
[CTRL]-[v]	Marks and cuts a text block to the Clipboard (see "Using the Clipboard" on page 2-10).
[CTRL]-[t]	Cuts a text block and places it on the Clipboard.
[CTRL]-[p]	Pastes a text block.
[F1]	Inserts a blank line above current line.
[F2]	Deletes current line.
[ENTER]	Moves cursor to start of next line.
[HOME]	Moves cursor to top left corner of form.
[	Defines a new field.
]	Lengthens an existing field.
[ESC]	Toggles between pull-down menus and Form Editor.
[DEL]	Returns to pull-down menus.

Defining Fields

When painting the form image, you enter field labels and field attributes. You define a field in the Form Editor by pressing the left bracket ([) key. This causes the Define Fields window to appear.

You define fields and set field attributes in the Define Fields window. When you press the left bracket key ([) from within the Form Editor, the Define Fields window appears.



In the Define Fields window you specify the attributes of the field. The attributes are arranged in the window so that the most important and least modified values are supplied first.

Most important are the Table Name and Column Name fields. You can enter values into these two fields directly or use Zoom to select from a list of available values.

The Field Type column is automatically filled in when you enter a valid column name in the Column Name field. You cannot modify the Field Type field because it relates to the column as defined in the database. If you specify Table Name as formonly, you are able to specify a value in the Field Type column.

The Input Area field specifies whether the field is on the header (1) or detail (2) part of the form.

The Entry? field is a Y/N field that determines whether the field is for display purposes only or if it accepts input from the user.

The Message field stores a descriptive line that is displayed when the user positions the cursor in the field.

In the Picture field, you can add a character pattern for displaying the data. For example, area code and phone number fields might display use (###) ###-#### as their character pattern.

The Display Fmt field serves as a hybrid attribute for Informix FORMAT and DISPLAY LIKE attributes, which are mutually exclusive. Refer to your Informix reference manuals for more information on these attributes.

The Validate field is similar to Display Fmt. It covers the INCLUDE and VALIDATE LIKE Informix attributes. These attributes are also mutually exclusive. Again, refer to your Informix manuals for more information on these attributes.

The Default field lets you set a default value to appear in the field. The user can change default field values.

The Translate field lets you indicate which language you want to use to display data for this field. If specified, translation logic is generated for this field.

The remaining fields are Y/N fields. You can experiment with these fields to see how they affect your input field.

Marking, Copying, and Pasting

When painting your form image, you can cut and paste fields and text. Copying consists of marking a block of text using the arrow keys and selecting the Copy option from the Edit pull-down menu. Once copied, you can paste the text block anywhere in your form image.

To mark and copy a text block:

1. **Position the cursor at one corner of the block of text you want to cut.**
2. **Press [CTRL]-[v] to start the Mark feature.**
3. **Use the arrow keys to highlight the entire block of text you want to mark.**

   As you move the cursor, the text you mark appears in reverse video.
4. **When you finish marking the entire block, press [CTRL]-[v] to copy the text block to the Clipboard.**

To paste a text block back onto your form image:

1. **Position the cursor on the form image where you want the block to appear.**
2. **Press [CTRL]-[p] to paste the block from the Clipboard to the form image.**
3. **Use the arrow keys to adjust where you want the block to *stick*.**

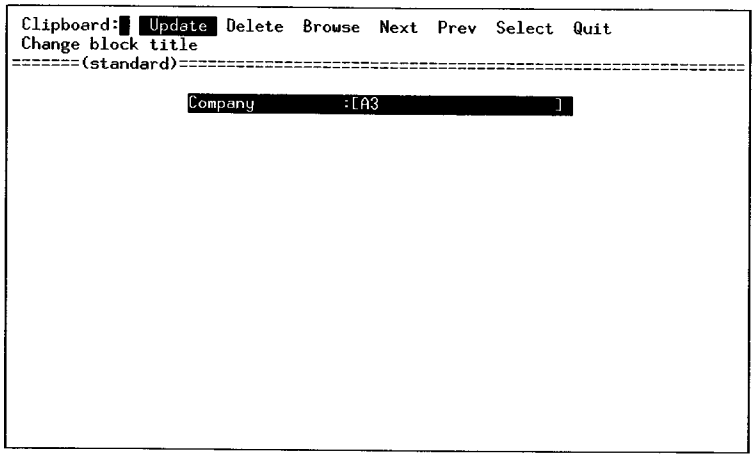
   You can move the entire text block to any location on your form image before you stick it to the image.
4. **Press [ESC] to stick the block to your form image.**

In a similar fashion, you can cut a block of text from your form image. Mark the block you want to cut as described above. Once you mark the text block, press [CTRL]-[t] to cut it. You can also paste a cut block back onto your form image in the same manner as described above.

Using the Clipboard

The Clipboard acts as a temporary storage place for text blocks. You can place anything onto the Clipboard and retrieve it. All the text you cut or copy gets placed on the Clipboard. Any text that you overwrite when you paste a block onto your form image gets stored to the Clipboard. You can access the Clipboard from the Edit pull-down menu.

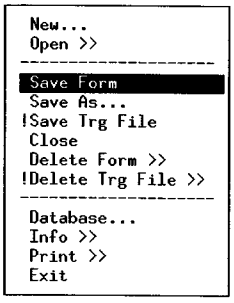
The Clipboard acts as a temporary storage place for text blocks.



Saving a Form Image

After you paint your form image, you must save it with the Save Form option on the File pull-down menu.

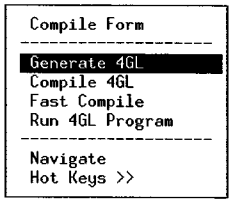
Use the Save Form option to save a form image.



Converting Forms into Input Programs

Once you create a form image and save it, you can run the *Screen* Code Generator and compilation utility from within the Form Painter. The Run pull-down menu contains all the options necessary to convert your form into an input program.

The Run pull-down menu contains all the options necessary to convert your form into an input program.



In general, you can use the following Run pull-down menu options to convert your form into an input program:

1. Generate 4GL - this option creates the INFORMIX-4GL source code.
2. Compile 4GL - this option compiles the 4GL code and links in library functions.
3. Run 4GL Program - this option runs the input program in the same manner a user would see it.

Note

You can also run the *Screen* Code Generator and compilation utility from outside the Form Painter (see "Starting the Tools from the Command Line" on page 8-8).

Section Summary

- The Form Painter is a front-end tool that lets you develop complete data-entry programs written in INFORMIX-4GL.
- There are two commands that start the Form Painter: fg.start and fg.form.
- The Form Painter contains five pull-down menus. You can open a pull-down menu by highlighting it and pressing [ENTER].
- The Form Painter lets you paint form images. You can use the Form Painter to create a new form image or modify an existing form image.
- Once you load a form into the Form Editor, you can start painting the form image. Form images contain both text and input field definitions.
- When painting the form image, you enter field labels and field attributes. You define a field in the Form Editor by pressing the left bracket ([) key.
- When painting your form image, you can cut and paste fields and text. Copying consists of marking a block of text using the arrow keys and selecting the Copy option from the Edit pull-down menu.
- The Clipboard acts as a temporary storage place for text blocks.
- After you paint your form image, you must save it with the Save Form option on the File pull-down menu.
- Once you create a form image and save it, you can run the *Screen* Code Generator and compilation utility from within the Form Painter.

Exercise 2A

Objective: To create a practice directory in which you will build your own input program.

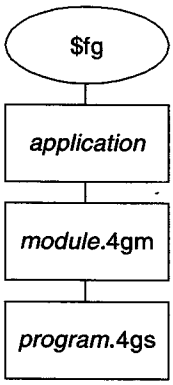
Create a Practice Directory Structure

In Exercise 1, you used the `scr_demo 5` to build an input program. The `scr_demo` command created a new shell for you to work in and placed you in the screen demo "program" directory. When you create input programs—without using the screen demos—you must create your own directory structure.

-generated input programs use a four-tiered directory structure. The first tier is your fourgen directory or the directory specified by the `$fg` variable. For example:

```
$fg=/usr/
```

The second tier is the application directory followed by the module directory and finally the program directory. The module and program directories use special naming extensions: *.4gm for the module directory and *.4gs for the program directory.



Before you build an input program with the Form Painter, it helps to duplicate this directory structure.

1. **Move to your home directory:**

   ```
   cd $HOME
   ```

2. **Create an application directory called `labs`:**

   ```
   mkdir labs
   ```

3. **Move to your `labs` directory and create a module directory called `aw.4gm` for Application Workbench:**

   ```
   cd labs; mkdir aw.4gm
   ```

   The semicolon delimits two UNIX commands.

4. **Move to your `aw.4gm` directory and create a program directory called `i_cust.4gs`:**

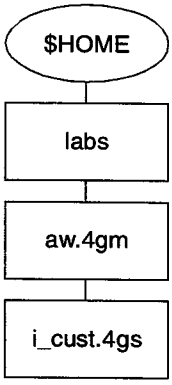
   ```
   cd aw.4gm ; mkdir i_cust.4gs
   ```

   Program directories reflect the type of programs they contain. Input program directories start with `i_`, which stands for input.

5. **Finally, move to the i_cust.4gs directory:**

   ```
   cd i_cust.4gs
   ```

   Once complete, you should be in the `i_cust.4gs` directory and have the following directory structure:



Exercise 2B

Objective: To start and become familiar with the Form Painter.

Start the Form Painter

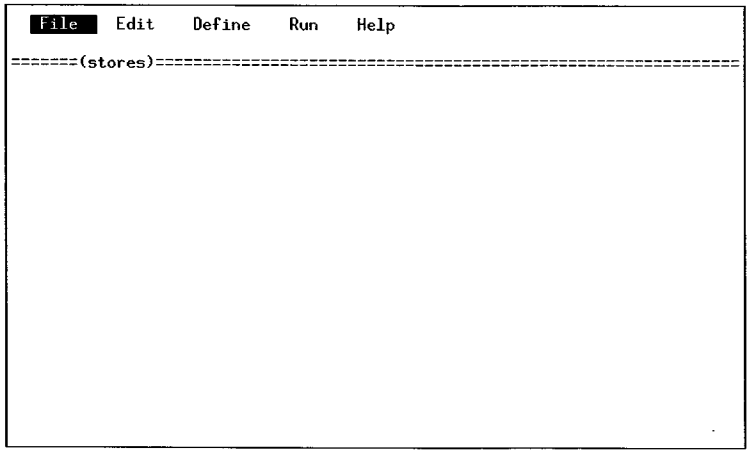
From within the `i_cust.4gs` directory, you can use the Form Painter to build an input program.

To start the Form Painter, enter:

```
fg.form
```

Note The `-dbname` flag specifies the database you want to use with the Form Painter. If you have been set up to use a different database, specify it in place of `stores`.

After you enter the `fg.form` command, the Form Painter appears:



The Form Painter lets you design input forms. In the next section you will build a Customer Entry program.

Exercise 2C

Objective: To use the Form Painter to design a Customer Entry form.

There are several steps involved in designing a Customer Entry form. In general you should use the following sequence:

1. Create a new form.
2. Add field labels.
3. Define which table and columns are used.
4. Save the form.

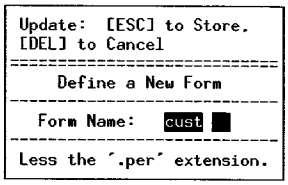
Create Your New Form

The New option on the File pull-down menu lets you create a new form. For this exercise, you will make a header form called `cust`.

1. **Select New from the File pull-down menu.**

   The Define a New Form box appears.

2. **Enter `cust` into the Form Name field.**



   The "Select the screen type" box appears.

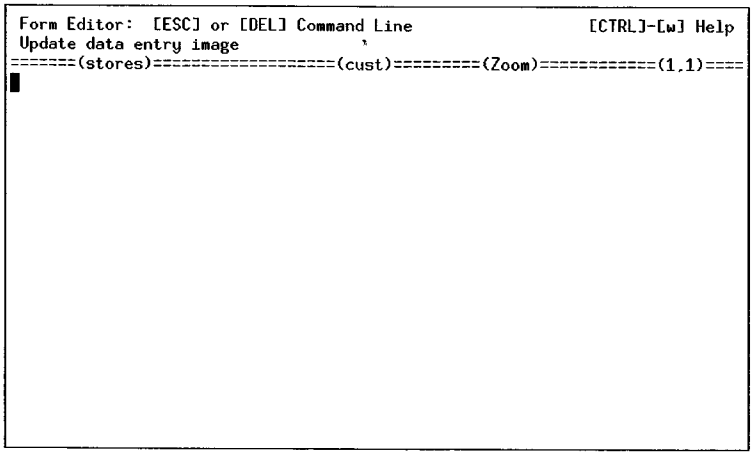
3. **Choose header from the "Select the screen type" box.**

   A new form is created and the cursor is placed on the upper left corner of the form (at this point, the form is empty).

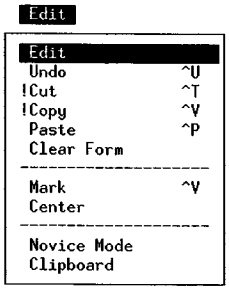
Add Field Labels

Once you create a new form, you can use the Form Editor to add input field labels. If you have just created a new form, your cursor is placed within the Form Editor automatically. The Form Editor lets you enter text and define input fields:

When you create a new form, your cursor is placed within the Form Editor automatically.



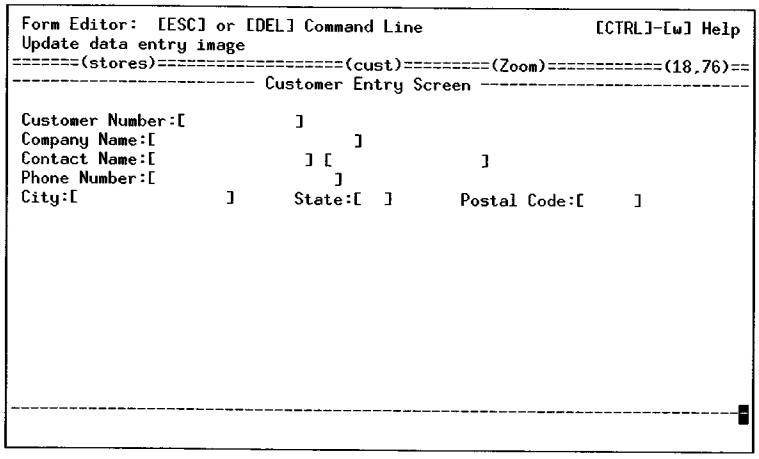
The [ESC] key lets you toggle between the Form Editor and the pull-down menus. You can also move to the Form Editor by selecting Edit from the Edit pull-down menu.



The Form Editor provides a number of useful editing keys and keystrokes to help you design your input form. The following list contains a few of them:

[F1]	Inserts a line.
[F2]	Deletes a line.
[ENTER]	Moves cursor to the start of the next line.
[HOME]	Moves cursor to the upper left corner.
[CTRL] - [a]	Toggles between insert and overstrike mode.
[CTRL] - [x]	Deletes a character.
[CTRL] - [d]	Deletes to the end of a line.
[CTRL] - [u]	Undoes an edit.

For this exercise, use the Form Editor to add input field labels that resemble the following form:



Make sure to add a dashed line to the bottom of your form. This line will separate your the message line from your input form. After you create all the labels, you can define the actual input fields themselves.

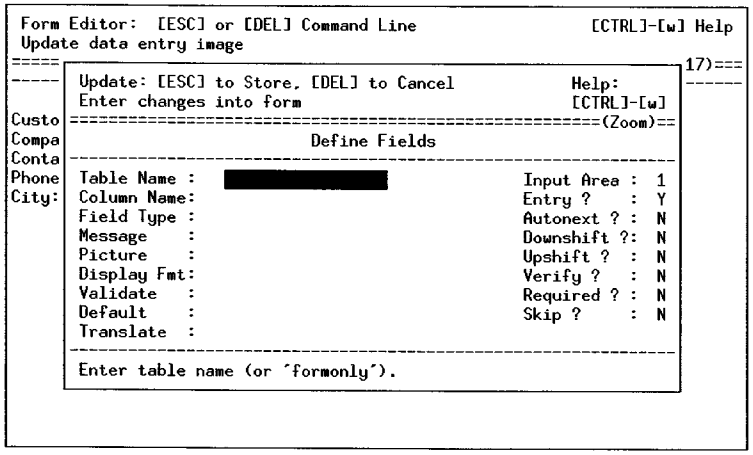
Define Input Fields

At this point, you need to define a corresponding field for each field label on your form. The Form Editor gives you a special key, the left bracket ([) key, for defining input fields.

1. **Position your cursor to the right of the Customer Number field label you created.**

2. **Press the left bracket ([) key.**

   The Define Fields dialog window appears.



Input fields are associated with columns in a database table. They accept data from the user and insert it into a column. In this exercise, each field that you define will correspond to a column in the `customer` table.

Note If you see a simplified version of this window, you are in "Novice mode." For all exercises in this training material, you must be in "Expert mode." The Edit pull-down menu contains an option that toggles between Expert and Novice mode. When Novice Mode is showing, it means you are in Expert mode and vice versa.

3. **Enter `customer` in the Table Name field.**

4. **In the Column Name field, press [CTRL]-[z].**

   A list of all the columns in the `customer` table appears.

5. **Highlight `customer_num` and press [ESC] to select it.**

   Data entered by the user into the Customer Number input field will go directly into this column in the `customer` database.

6. **Press [ENTER] to move to the Input Area field.**

   Notice that when you press [ENTER] the Field Type field gets filled in automatically with a `serial not null` value.

7. **Verify that the Input Area field contains a 1 and press [ENTER].**

   For now, all fields will have an Input Area of 1 (see "Input Areas and Specification Files" on page 6-1). Place a 1 in this field.

8. **Accept the Y value for the Entry? field and press [ENTER].**

   A Y value lets the user enter data into this field. An N specifies a no-entry field (i.e., a field in which the user cannot enter data).

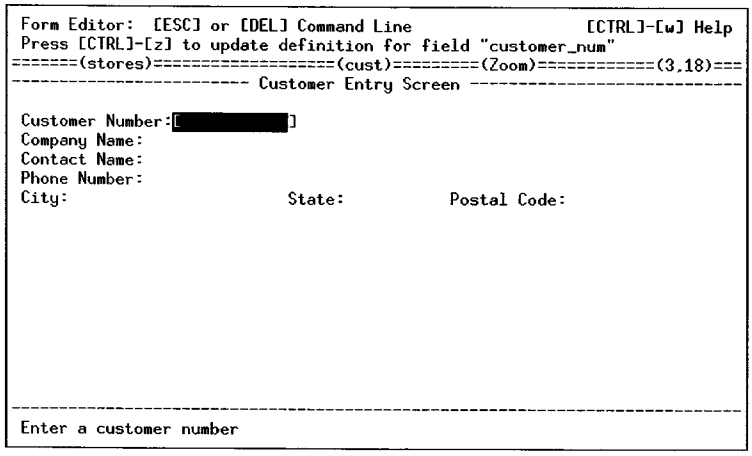
9. **Type a message in the Message field and press [ESC].**

   This message will appear at the bottom of the form when the cursor is in the Customer Number field.

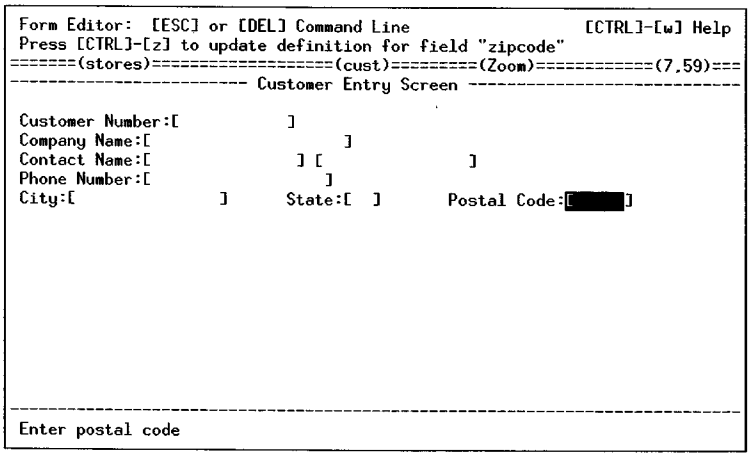
   For now, you can leave the other fields on the Define Fields window as is. The finished window should appear as follows:

```
Update: [ESC] to Store, [DEL] to Cancel          Help:
Enter changes into form                          [CTRL]-[w]
================================================================
                        Define Fields
----------------------------------------------------------------
Table Name :  customer                    Input Area :  1
Column Name:  customer_num                Entry ?    :  Y
Field Type :  serial not null             Autonext ? :  N
Message    :  Enter a customer number     Downshift ?:  N
Picture    :                              Upshift ?  :  N
Display Fmt:                              Verify ?   :  N
Validate   :                              Required ? :  N
Default    :                              Skip ?     :  N
Translate  :
----------------------------------------------------------------
Enter the input mask (picture) for this field. (no quotes)
```

Once you save the Customer Number field definition, the field appears in the Form Editor as two brackets with a highlight between them. Notice also how the field is automatically sized and the field message appears at the bottom of the screen:



Follow the same sequence of steps to define the rest of the input fields on your form. For the Contact Name field, define two fields (`fname` and `lname`). When you finish, your form should look as follows:



After you create a field definition, you might need to re-edit it at some point.

To re-edit a field definition:

1. **Place your cursor in the field and press [CTRL]-[z].**

   A pop-up menu appears.

2. **Select Field from the pop-up menu.**

   The Define Fields window appears.

3. **Edit the field definition using the Define Fields window and press [ESC] to save your changes.**

Save the Form

When you are satisfied with your input form, save it using the Save Form option under the File pull-down menu.

To save a form:

- **Select Save Form from the File pull-down menu.**

  The Form Painter reads your form image and generates instructions in a form specification (*.per) file. This file gets used by the *Screen* Code Generator to create source code, which is discussed next.

Exercise 2D

Objective: To use the Form Painter to generate, compile, and run your Customer Entry program.

Recall that you built a demonstration input program from the UNIX command line using `fg.screen`, `fg.make`, and `fglgo`. The Form Painter gives you the same ability, but you simply select these commands from the Form Painter's Run pull-down menu.

Generate Source Code

1. **Select Generate 4GL from the Run pull-down menu.**

   A pop-up menu appears asking you which forms to generate code for.

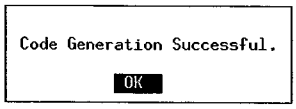
2. **Select All Forms from the pop-up menu.**

   A message box appears asking you if you want to only generate code for local forms.

3. **Select YES on the "Local forms only" message box.**

   The *Screen* Code Generator is run and code scrolls past your screen as it creates code based on your `cust` form. You might see a message indicating that your `cust` form is not current. If this happens, simply select YES from the message box.

   When the *Screen* Code Generator finishes, the following message appears:



Compile the Code

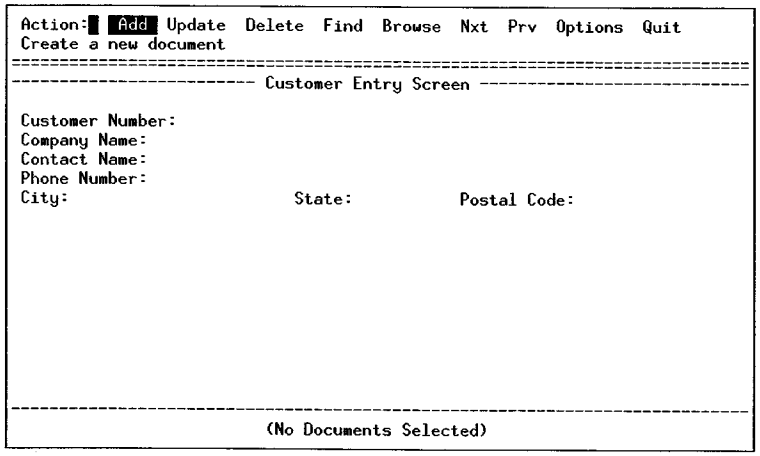
- **Select Compile 4GL from the Run pull-down menu.**

The Form Painter calls the compilation utility and creates a program file. When done, the following message appears:

Run Your Customer Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**

   The Form Painter runs your Customer Entry program.



2. **Use the ring menu to "test drive" your input program. When you finish, select Quit to return to the Form Painter.**

Exercise 2E

Objective: To make a slight change to your Customer Entry program and then rebuild it.

At times, you may want to make changes to your form and incorporate those changes into your generated-input program. For example you may want to move a field label and definition to a different location. The Form Painter makes this task easy.

In this exercise you will use the Form Painter's Mark, Cut, and Copy options to move the Phone Number field to a new location on the `cust` form. Once finished, you will save `cust` and rebuild an input program from it. The resulting input program will reflect the change you made.

Note This exercise picks up where the Exercise 2C left off. You should be in the Form Painter and have your `cust` form visible in the Form Editor. If you are not at this point, use the steps in the previous sections to catch up.

In general there are three steps to moving a portion of your form:

1. Mark the portion you want to move.
2. Cut the marked portion.
3. Paste the cut portion back onto the form in the appropriate spot.

Mark the Phone Number Field

Before you can move a portion of your `cust` form, you must mark it. You can mark anything that appears on your form: field labels, field definitions, or both.

1. **If you are not in it already, move to the Form Editor: Select the Edit option under the Edit pull-down menu.**
2. **Move the cursor to the start of the Phone Number field.**
3. **Press [CTRL]-[v].**

This keystroke places you into "Mark" mode.

4. **Use the arrow keys to highlight the Phone Number field.**

```
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
=======(stores)==================(cust)=========(Zoom)============(6,35)===
------------------------ Customer Entry Screen ----------------------------

Customer Number:[          ]
Company Name:[                  ]
Contact Name:[               ] [              ]
Phone Number:[                  ]
City:[              ]     State:[  ]      Postal Code:[     ]
```

This is the area that you will cut.

Cut the Phone Number Field

Once you mark (i.e., highlight) the Phone Number field, you can cut it from your form (once cut, you can paste it back into your form at any location).

- **Press [CTRL]-[t].**

  The Phone Number field disappears.

```
Form Editor:  [ESC] or [DEL] Command Line                 [CTRL]-[w] Help
Update data entry image
=======(stores)==================(cust)=========(Zoom)============(6,35)===
------------------------ Customer Entry Screen ----------------------------

Customer Number:[          ]
Company Name:[                  ]
Contact Name:[               ] [              ]

City:[              ]     State:[  ]      Postal Code:[     ]
```

Now you can use the Paste option to stick this field below the City, State, and Postal Code line.

Paste the Phone Number Field Back into Your Form

1. **Move your cursor below the City field.**

```
Form Editor:  [ESC] or [DEL] Command Line                    [CTRL]-[w] Help
Update data entry image
=======(standard)=================(cusent)=======(Zoom)============(8,1)====
---------------------- Customer Entry Screen -------------------------------

Customer Number:[           ]
Company Name:[                    ]
Contact Name:[               ] [              ]

City:[              ]    State:[  ]     Postal Code:[     ]
```

2. **Press [CTRL]-[p].**

   The Phone Number field reappears. You can use the arrow keys to "slide" the field around, but for now, leave it where it is.

Note Be sure to never paste on top of existing form objects. You must paste on a blank space or line.

3. **Press [ESC] to "stick" the field to the form.**

Use the Clipboard

If you make a mistake during cutting and pasting, you can select the Clipboard option from the Edit pull-down menu. Everything you cut gets placed on its own page in the Clipboard. You can use the Clipboard's ring menu to scroll through all the objects you have cut and select the one you want.

When you select an object from the Clipboard, it gets pasted into your form (just like the Phone Number field). Once again, you can reposition the object with the arrow keys before pressing [ESC] to "stick" it to the form.

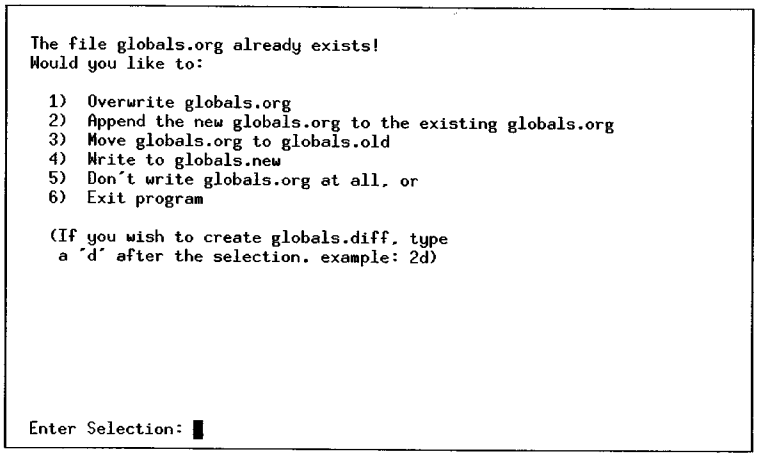
Save Your Changes

Now that you have moved the Phone Number field, you can save your form and rebuild it. Once rebuilt, the resulting input program will reflect the new location of the Phone Number field.

1. **Save your `cust` form with the Save Form option under the File pull-down menu.**

2. **Select Generate 4GL from the Run pull-down menu.**

   During code generation, the "Overwrite" message might appear:



   This message lets you know that you are creating a "new" source code file on top of a file that already exists in your `i_cust.4gs` directory. For this exercise—and in most cases for that matter—you want to overwrite this file. Depending on the number of changes you have made, you might see this message several times.

   When it appears, simply select option one to overwrite the file.

3. **Select Compile 4GL from the Run pull-down menu.**

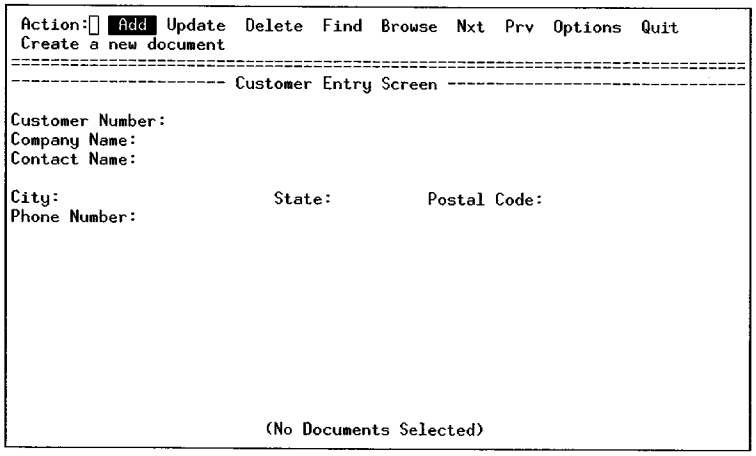
   Once compiled, your program is ready for you to run.

Run the Customer Entry Program Again

Now you can see your changes in the resulting input program.

- **Select Run 4GL Program from the Run pull-down menu.**

  The Form Painter initiates your Customer Entry program.



Notice that the Phone Number field appears in its new location. Once again, spend some time experimenting with this program. Add a new document and see if the cursor path through your input fields has changed.

When you are done, select Quit from the ring menu to return to the Form Painter. Exit out of the Form Painter as well (select Exit from the File pull-down menu).

3

Working with the Database

Main topics:

- Displaying the Table Information Window
- Changing Database Values
- Using the AutoForm option

Displaying the Table Information Window

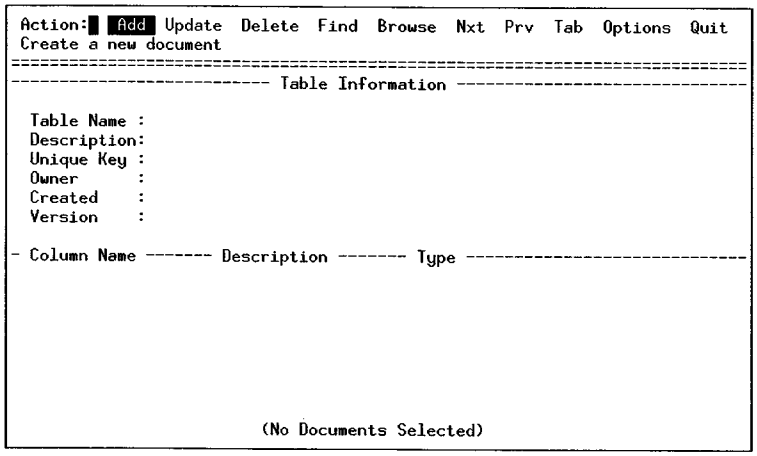
The Form Painter gives you direct access to the database through the Table Information window. This window lets you manage tables and columns in the database.

To initiate the Table Information window:

- **Select Database from the File pull-down menu.**

  The Table Information window appears.

Use the Database option on the File pull-down menu to initiate the Table Information window.



The Table Information window lets you do the following:

- Alter the structure of your database
- Add and drop database tables
- Add, modify, and drop columns from tables

Changing Database Values

The Table Information window looks and functions like other generated input programs because it was created with the *Screen* tools.

The Table Information window looks and functions like other generated input programs.

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

  Table Name : customer
  Description: Customer Information
  Unique Key : customer_num
  Owner      : seanb
  Created    : 10/18/93
  Version    :  57

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

  customer_num        Customer Number     serial not null
  fname               First Name          char(15) not null
  lname               Last Name           char(15)
  company             Company Name        char(20)
  address1            Address Line #1     char(20)
  address2            Address Line #2     char(20)
  city                City                char(15)
                                  (1 of 1)
```

You can add tables to the database and give them descriptive names. It is very important to fill in the Unique Key field. This field identifies to an input program the columns that uniquely identify a row.

When you use the Table Information window to alter a table (for example, you delete a column), a pop-up window appears and displays the SQL statement that it will run on the table.

The Form Painter stores all the changes you make in a file called `dbadmin.sql`. All changes are also time stamped, and this file remains in your local program directory.

Using the AutoForm Option

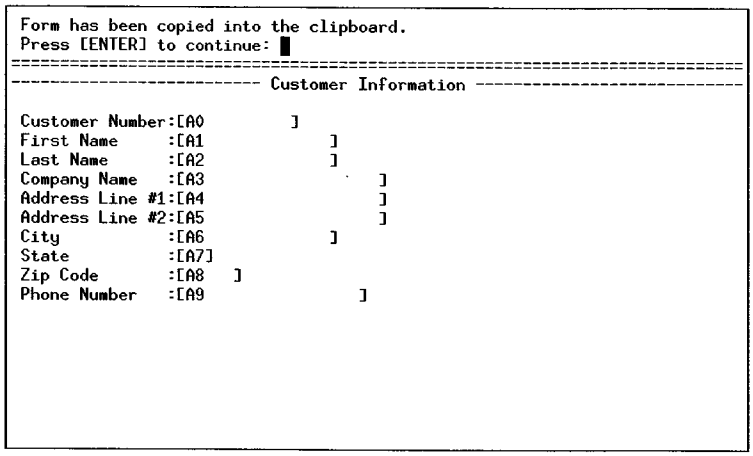
The Table Information window also lets you generate a default form image from a table; in other words you can create an AutoForm. The AutoForm command is located under the Options command on the ring menu. When you create an AutoForm, the AutoForm image gets stored to the Clipboard. You can then quit from the Table Information window and paste the AutoForm image into your form image using the Form Painter.

To create an AutoForm:

1. **Use Find to select the table you want to generate an AutoForm from (see "The Find Command" on page 1-6).**
2. **Select the Options command then AutoForm.**

   An AutoForm gets built and its image is stored to the Clipboard.

Use the AutoForm command to generate a default image of a table.



Once you create an AutoForm, you can go back to the Form Painter and retrieve the AutoForm from the Clipboard. Once retrieved, the AutoForm is placed into the Form Editor, and you can edit it any way you want.

Section Summary

- The Form Painter gives you direct access to the database through the Table Information window. This window lets you manage tables and columns in the database.
- With the Table Information window you can alter the structure of your database; add and drop database tables; and add, modify, and drop columns from tables.
- The Table Information window looks and functions like other generated input programs.
- The Table Information window also lets you generate a default form image from a table; in other words, you can create an Auto-Form.

Exercise 3

Objective: To create a credit table that holds credit codes, descriptions, and amounts. Such a table could hold the following values:

Credit Code	Credit Description	Credit Amount
AAA	Excellent	10,000
BBB	Good	5,000
CCC	Fair	1,000
DDD	Poor	250

Start the Form Painter

1. **Move to the `$HOME/labs/aw.4gm` directory:**

   ```
   cd $HOME/labs/aw.4gm
   ```

2. **Create a new directory to hold a credit entry program.**

   ```
   mkdir i_cred.4gs
   ```

3. **Move to the `i_cred.4gs` directory:**

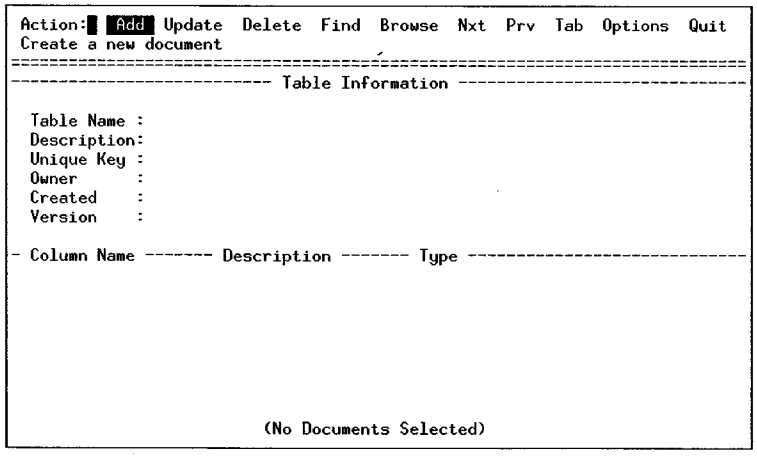
   ```
   cd i_cred.4gs
   ```

4. **Start the Form Painter.**

Open the Database Option

1. **From the File pull-down menu, select Database.**

   The Table Information window appears.



   The Database option is a data-entry program that allows you to change the structure of your database. You can add, delete, and alter tables by adding, deleting, and re-arranging columns, and changing column types. You can change the structure of your database much like using Informix ISQL to do so.

   Notice how the screen looks just like a typical input program created with *Screen*. It has the same ring menu that your customer entry program has.

2. **Select the Find ring menu option.**

   Your cursor moves to the Table Name field.

3. **Type `customer` in the Table Name field.**

4. **Press [ESC].**

   Information about the `customer` table appears.

```
Action:  Add  Update  Delete  Find  Browse  Nxt  Prv  Tab  Options  Quit
Select a group of documents
================================================================================
-------------------------- Table Information -----------------------------

  Table Name : customer
  Description: Customer Information
  Unique Key : customer_num
  Owner      : brianh
  Created    : ********
  Version    : ***

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

  customer_num        Customer Number     serial not null
  fname               First Name          char(15) not null
  lname               Last Name           char(15)
  company             Company Name        char(20)
  address1            Address Line #1     char(20)
  address2            Address Line #2     char(20)
  city                City                char(15)
                                (1 of 1)
```

   Notice how the upper half of the screen (the "header") portion contains information about the `customer` table. The lower half of the screen (the "detail" portion) displays all of the columns that make up the `customer` table.

Add the credit Table

1. **Select the Add ring menu option.**

2. **Add a table to hold `credit` information.**

   Name your new table credit and add a descriptive name for the table. Do not enter a Unique Key value yet.

3. **Press [TAB] to move to the detail portion of the screen and add the following columns as detail rows:**

Column Name	Description	Type
credit_code	Credit Code	char(3)
credit_desc	Credit Description	char(10)
credit_amt	Credit Amount	decimal(10,2)

4. **Press [TAB] to move back to the header portion of the screen and fill in the Unique Key as `credit_code`.**

   All tables must have a unique key (i.e., a column that uniquely identifies a row.

5. **Press [ESC] to store your new table.**

Create an AutoForm from the credit Table

1. **Select the Options ring menu and then choose AutoForm.**

   This builds a default data-entry form based on your `credit` table. It then copies this form to the Clipboard. Once on the Clipboard, you can paste it into a new form.

2. **Press [ENTER].**

3. **Select Quit from the ring menu to return to the Form Painter.**

Create a New Form

1. **From the File pull-down menu, select New.**

   The Define a New Form box appears.

2. **Name the form `cred`.**

   The Select the Screen Type box appears.

3. **Choose `header` as the screen type.**

Use the Clipboard

Instead of creating fields individually, you can copy the AutoForm you created and stored on the Clipboard.

1. **Select Clipboard from the Edit pull-down menu.**

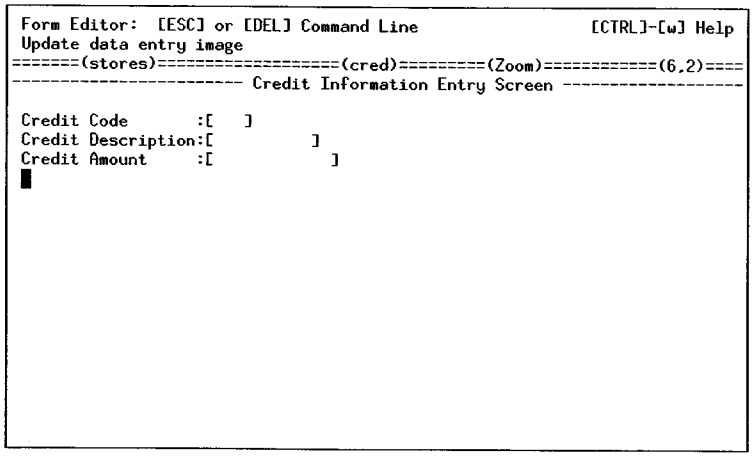
   Find the AutoForm for the credit table.

2. **Once you find the credit table AutoForm, choose Select.**

   The Select option pastes the AutoForm into the Form Editor. You can use the arrow keys to position it.

3. **Press [ESC] to stick it down.**

   Remove the extra heading line that came with the AutoForm.



Save, Generate, and Compile

1. **Save your newly-created form.**

   Use the Save Form option under the File pull-down menu.

2. **Select Generate 4GL from the Run pull-down menu.**

   When it is finished, the Code Generation Successful message appears.

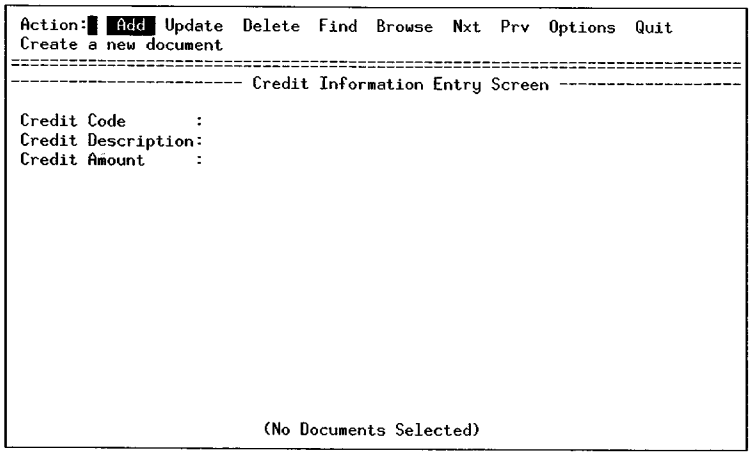
3. **Select Compile 4GL from the Run pull-down menu.**

   When it is finished, the 4GL Compile Succeeded message appears.

Run Your Credit Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**

   The Credit Entry program starts.



2. **Enter at least four new credit codes.**

   You can use the sample codes shown on page 3-6.

3. **When finished, exit the program and the Form Painter.**

4

Creating Zooms

Main topics:

- Zoom Screen Overview
- Painting a Zoom Image
- Attaching the Zoom Screen

Zoom Screen Overview

A Zoom is a data validation feature that shows the user a list of valid values for an input field Zooms are created from zoom screen types (see "Using Different Screen Types" on page 11-2) When users initiate a Zoom, they can enter selection criteria on the fields in the Zoom The Zoom then takes the selection criteria and returns all valid values that meet the criteria Users can select the value they want to use from the values the Zoom returns

Zooms make the data-entry process much more accurate and efficient Field values are validated before they get inserted. In general, creating Zooms is a two step process:

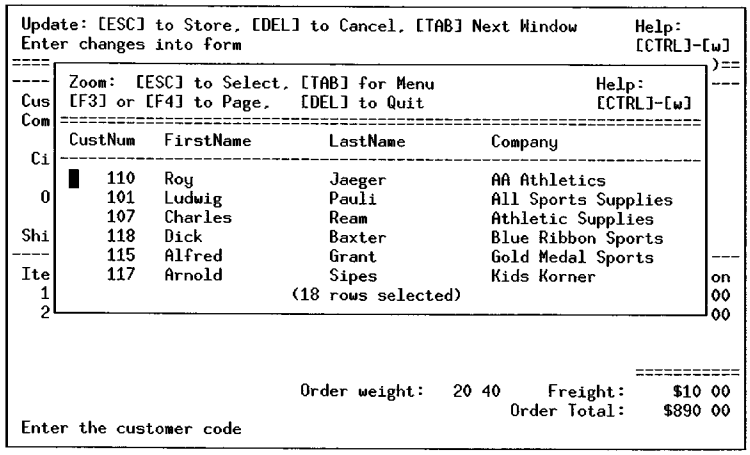
1 Paint and define the zoom screen image.

2. Attach the zoom screen to a field on your main input screen

Painting a Zoom Image

You define Zooms by using the Form Painter to paint their image Once you paint the image of the zoom screen, you must also specify from which field on your main input form the zoom screen can be activated. For example, the following application has a zoom screen attached to the Customer No. field

Zoom screens are attached to input fields Users initiate this Zoom from the Customer No field



To define a Zoom:

1. **Select New from the File pull-down menu.**
2. **Specify a name for the zoom screen.**

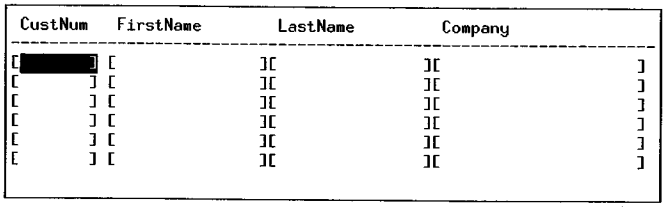
   Traditionally, zoom screens are given a name that includes the letters `zm`, such as `cust_zm`, `stockzm`, etc

3. **Select zoom as the screen type.**

4. **Use the Form Painter to paint and save the zoom image (see "Creating a Form Image" on page 2-5).**

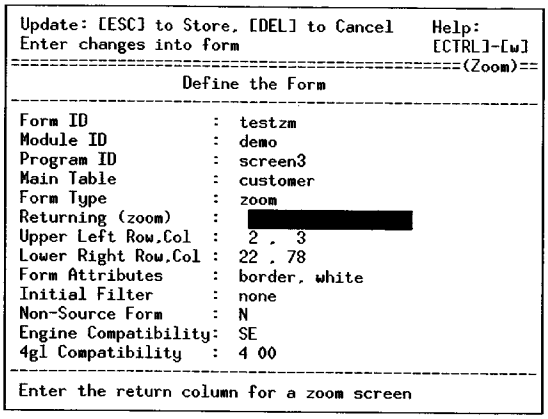
   Because zoom screens usually contain several rows of duplicate field definitions, use mark, copy, and paste to speed your creation of the zoom image (see "Marking, Copying, and Pasting" on page 2-8)

Zooms, such as this one, usually contain several rows of duplicate field definitions



After you paint and save your zoom image, you need to use the Form Defaults option on the Define pull-down menu. The Form Defaults option opens the Define the Form window This window lets you specify from which field the zoom screen can be activated.

After you paint and save your zoom image, you need to use the Define the Form window to set your Zoom attributes



Make sure to specify a value in the Returning (zoom) field. This field specifies where the returning value gets placed In most cases, this is the field in which you attach the Zoom If you are not sure of the field, press [CTRL]-[z] while to see a list of available fields

Attaching the Zoom Screen

You can attach a zoom screen to the main screen of your program using the Form Painter

To attach a zoom screen to an input field:

1. **Open the form that contains the field that you want to attach the zoom screen to.**

   In most cases, you attach zoom screens to header or header/detail screens, but this is not necessarily the case

2. **Highlight the field you want to attach the zoom screen to.**

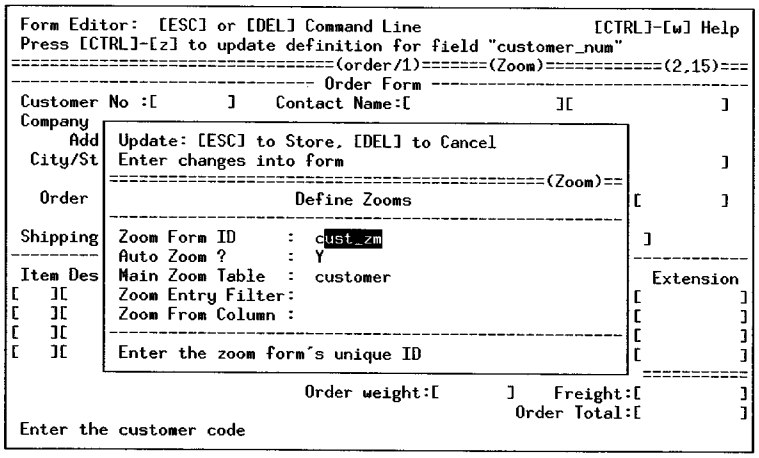
3. **Press [CTRL]-[z]**

   Note the irony here You activate a Form Painter Zoom in order to define a Zoom for your input program When you press [CTRL]-[z] a pop-up menu appears that contains all the items available for you to attach to the input field

4. **Select Zoom... from the list.**

   The Define Zooms window appears

The Define Zooms window lets you attach a zoom screen to an input field



5. Fill in the Define Zooms window and press [ESC].

The Define Zooms window lets you specify how you want the zoom screen to be attached.

Use the Define Zooms window to specify how you want the Zoom to be attached.

```
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.
```

The Define Zooms window contains several fields. Perhaps the Zoom Form ID field is most important. In this field, you place the name of your Zoom screen. You should make sure that the Main Zoom Table field contains the correct value. If you want to add AutoZoom capability, specify Y in the AutoZoom field.

The Zoom Entry Filter field lets you assign a selection filter to the Zoom. The last field, Zoom From Column, lets you specify a table and column name for the Zoom if they differ from the column on the main screen.

Section Summary

- A Zoom is a data validation feature that shows the user a list of valid values for an input field. Zooms are invoked by pressing [CTRL]-[z].
- You define Zooms by using the Form Painter to paint their image. Zooms are created from zoom screen types. Once you complete painting a Zoom, you can attach it to a field on your input program.
- To attach a zoom screen to an input field, you must identify which field the Zoom applies to. You can set all the Zoom attributes in the Define Zooms window.

Exercise 4A

Objective: To add a `credit` field to the `i_cust.4gs` program.

Start the Form Painter

1. **Move to `$HOME/labs/aw.4gm/i_cust.4gs`.**
2. **Start the Form Painter.**

Add the credit_code Column to the customer Table

1. **Select Database from the File pull-down menu.**

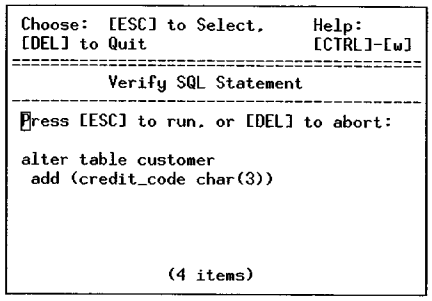
   The Table Information window appears.

2. **Select Find from the ring menu, enter `customer` in the Table Name field, and press [ESC].**
3. **Select Update and add a column named `credit_code` to the `customer` table:**

```
Update: [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                         [CTRL]-[w]
================================================================(Zoom)==
--------------------------- Table Information -----------------------------

  Table Name : customer
  Description: Customer Information
  Unique Key : customer_num
  Owner      : brianh
  Created    : ********
  Version    : ***

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

  address1            Address Line #1    char(20)
  address2            Address Line #2    char(20)
  city                City               char(15)
  state               State              char(2)
  zipcode             Zip Code           char(5)
  phone               Phone Number       char(18)
  credit_code         Credit Code        char(3)
Enter the data type for this column.
```

4. **Press [ESC].**

   A Verify SQL Statement box appears.



5. **Press [ESC] again to run the alter table SQL statement.**

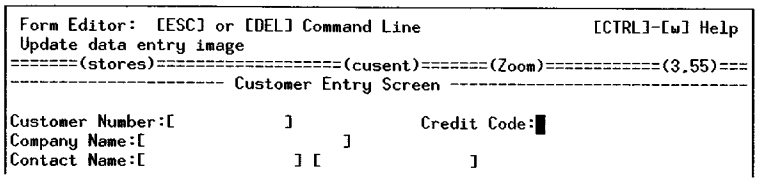
6. **Select Quit to return to the Form Painter.**

Add a Credit Code Field to Your Screen

1. **Select Open from the File pull-down menu.**

   The Form Painter opens your `cust.per` file. If you have additional form specification (*.per) files in this directory, you have to select `cust` from a list.

2. **Add a Credit Code field label in the upper half of your screen.**



3. **Define the Credit Code field by pressing a left bracket [after the field.**

   The Define Fields window appears.

4. **Define the Credit Code field using the values shown below, then press [ESC] to save the definition.**

```
Update: [ESC] to Store, [DEL] to Cancel              Help:
Enter changes into form                              [CTRL]-[w]
=============================================================
                        Define Fields
-------------------------------------------------------------
Table Name :  customer                    Input Area :  1
Column Name:  credit_code                 Entry ?    :  Y
Field Type :  char(3)                     Autonext ? :  N
Message    :  Enter a Credit Code         Downshift ?:  N
Picture    :  [redacted]                  Upshift ?  :  N
Display Fmt:                              Verify ?   :  N
Validate   :                              Required ? :  N
Default    :                              Skip ?     :  N
Translate  :
-------------------------------------------------------------
Enter the input mask (picture) for this field. (no quotes)
```

Save, Generate, and Compile

1. **Select Save Form from the File pull-down menu.**
2. **Select Generate 4GL from the Run pull-down menu.**
3. **Select Compile 4GL from the Run pull-down menu.**

Run Your Customer Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**
2. **Use Find to select an existing customer and add a credit code for that customer.**

```
Update: [ESC] to Store, [DEL] to Cancel                    Help:
Enter changes into form                                    [CTRL]-[w]
=============================================================================
--------------------------- Customer Entry Screen ---------------------------

Customer Number:        101          Credit Code: AAA
Company Name: All Sports Supplies
Contact Name: Ludwig           Pauli
```

3. **When finished, quit the Customer Entry program and the Form Painter.**

Exercise 4B

Objective: To create a zoom screen so users can select from a reference list of credit codes.

Create a Zoom Screen

1. **Start the Form Painter.**
2. **Select New from the File pull-down menu.**

   The Define a New Form box appears.
3. **Name the new form `cred_zm`.**

   The Select a Screen Type box appears.
4. **Use the down arrow to scroll down the screen type list and select zoom as the screen type.**

Create a Title

Enter a title for the zoom screen, such as:

---------- Credit Information Zoom ----------

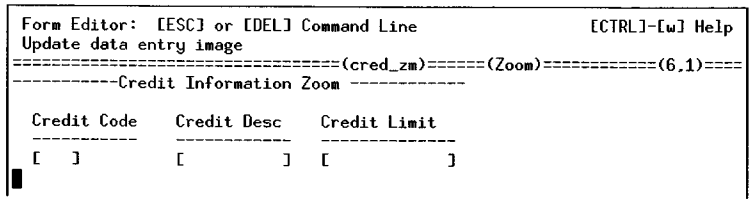
Create the Column Headings

1. **Create the column headings for the Zoom.**

```
Form Editor:  [ESC] or [DEL] Command Line                    [CTRL]-[w] Help
Update data entry image
==================================(cred_zm)======(Zoom)==(insert)==(5,3)====
----------Credit Information Zoom ------------

 Credit Code     Credit Desc     Credit Limit
 -----------     ------------    --------------
 ▮
```

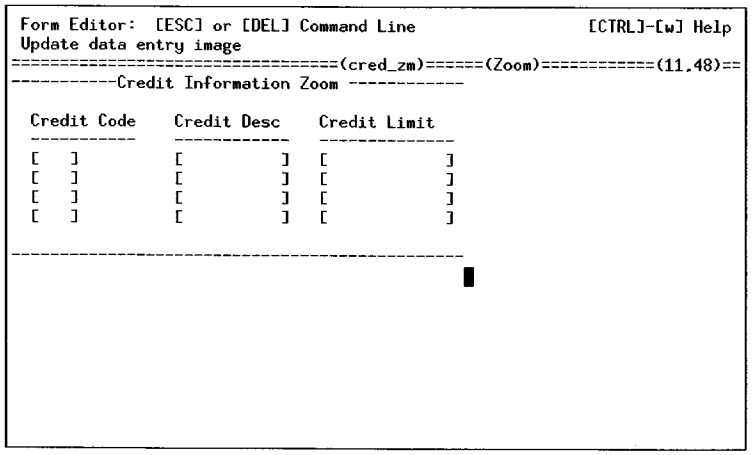
A zoom screen displays data in a row-by-row format.

2. **Add field definitions using the columns in your `credit` table. (`credit_code`, `credit_desc`, and `credit_amt`)**



3. **Use the Mark, Copy, and Paste options to add three more rows of field definitions, see "Marking, Copying, and Pasting" on page 2-8.**

Your finished zoom screen should look as follows:



Specify Form Defaults

1. **Select Form Defaults from the Define pull-down menu.**

The Form Defaults window appears.

2. **Enter credit in the Main Table field.**

Zooms typically return values to the field from which they were invoked. Since you will be Zooming from the Credit Code field on your Customer Entry program, you must specify from which column the data will be supplied.

3. **Add credit_code in the Returning (zoom) field.**

   You can bypass the other fields on the window.

4. **Select Save Form from the File pull-down menu.**

5. **Select Generate 4GL from the Run pull-down menu.**

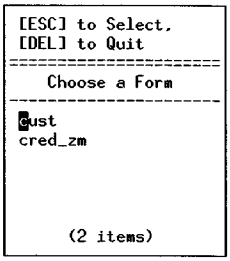
   The Generate 4gl: Enter Selection box appears.

6. **Select `cred_zm`.**

Attach cred_zm to the Credit Code Field

Now you must attach `cred_zm` to the Credit Code field that you credit on the Customer Entry program.

1. **Select Open from the File pull-down menu and open the file that corresponds to your Customer Entry program (`cust`).**



2. **Place your cursor in the Credit Code field and press [CTRL]-[z].**

   The Define Field pop-up menu appears.

3. **Select Zoom... from the Define Field pop-up menu.**

   The Define Zooms window appears.

4. **Enter `cred_zm` in the Zoom ID field.**

5. **Press [ENTER] in the Auto Zoom ? field and enter `credit` in the Main Zoom Table field.**
6. **Specify `credit_code` in the Zoom From Column field and press [ESC] to save your zoom definition.**

Save, Generate, and Compile

1. **Use the Save Form option under the File pull-down menu.**
2. **Select Generate 4GL from the Run pull-down menu.**

   The Generate 4gl: Enter Selection box appears.
3. **Select All Forms from this box.**

   The Local Forms Only box appears.
4. **Select YES.**

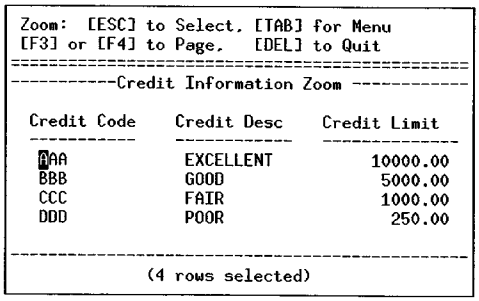
   As the *Screen* Generator runs, it builds code for both your zoom screen and your Customer Entry screen.
5. **Select Compile 4GL from the Run pull-down menu.**

Run Your Customer Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.
2. **Use Find to select an existing customer and select Update.**
3. **From the Credit Code field, press [CTRL]-[z] and press [ESC].**

The Credit Information Zoom appears.



4. **Use the cred_zm a few times. When finished, quit out of Customer Entry and the Form Painter.**

5

Creating Lookups

Main topics:

- Lookup Overview
- Attaching a Lookup to a Field

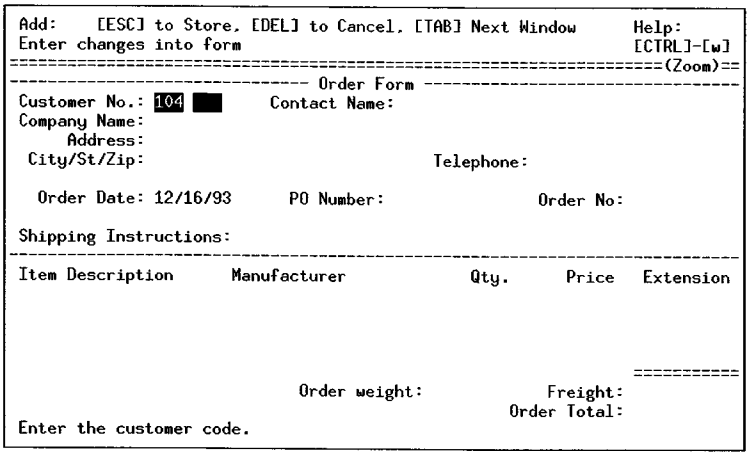
Lookup Overview

A Lookup performs a cross-check between two tables. You provide the lookup with a key value. The generator builds logic to open a cursor and fetch the key value from a reference table. If the key value does not exist in the reference table, an error is returned and the user is placed back in the Lookup field.

Lookups can also return data from the reference table keyed by the value you pass it. For example, if you pass a Lookup the customer number value, it can return a valid customer number, company name, owner name, steet address, and other customer information:

A Lookup validates data and returns related data

In this example, a Lookup is defined on the Customer No. field.



When the user enters a customer number, data relating to that number fills in the adjacent fields.

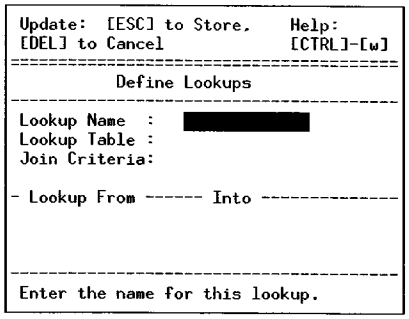
```
Add:    [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                         [CTRL]-[w]
================================================================================
------------------------------ Order Form ------------------------------------
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: 12/16/93      PO Number:                Order No:

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

                                                                  ===========
                              Order weight:            Freight:
                                                   Order Total:
Enter the order date.
```

Attaching a Lookup to a Field

Like Zooms, you attach Lookups to input fields. Before you create a Lookup, you must know which field you want to attach it to and which fields you want to return values to.

Lookups are defined with the Define Lookups window. This window lets you specify the Lookup name, table, and join criteria. You also specify which fields on your main form the Lookup should fill.

The Define Lookups window lets you specify the Lookup name, table, and join criteria.



The Lookup Name field holds the name of the Lookup. Uniquely naming Lookups lets you define multiple Lookups on the same field.

The Lookup Table field holds the name of the *looked up* table. In other words, this is the table from which values are being returned.

The Join Criteria field lets you specify the *where* clause of the join statement: you are specifying where the returned value is being put. The Join Criteria field uses the following syntax:

```
table_name.column_name = $field_name
```

Where *table_name* and *column_name* represent the looked up table and *field_name* represents the column where the value gets returned.

For example, the following join criteria instructs the Lookup to search the `customer_num` column in the `customer` table and verify that the value in the `customer_num` field exists:

```
customer.customer_num = $customer_num
```

The Lookup From and Into fields are optional. These fields let you specify the join criteria when the column and field names differ. For instance, if the column name is `description` and the field name is `desc`, you could define the Lookup as follows:

This example shows how the Lookup From and Into fields are used. You only need to use these fields when the column and field names do not match.

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

- Lookup From ------ Into --------------
description          desc
-----------------------------------------
Enter the column to lookup from.
```

If the fields and columns have the same name, you do not need to add them to the Lookup From and Into fields. The *Screen* Generator builds this logic when the field names and column names match.

To define a Lookup:

1. **Using the Form Painter, highlight the field that you want to attach a Lookup to.**
2. **Press [CTRL]-[z] to display the Define Field menu.**
3. **Select Lookups... from the Define Field menu.**

   The Define Lookups window appears. You can also access the Define Lookups window from the Define pull-down menu by choosing the Lookups... option.
4. **Fill in the Define Lookups window and press [ESC].**

When a user enters an invalid value into a field that has a Lookup attached, an error occurs. The user is not able to leave that field until a valid value has been entered.

Section Summary

- Lookups are placed on fields in a data-entry screen to evaluate the data entered by a user.
- Lookups check a key value against a reference data table. If the key value exists, the Lookup allows the user to continue. If the Lookup doesn't exist, an error occurs and the user is placed back in the Lookup field.
- Another purpose of a Lookup is to return data keyed by the Lookup value. A value entered by a user can cause a cross-referenced value to be looked up in the reference table and displayed on the input form.

Exercise 5A

Objective: To add a lookup on the Credit Code field. A lookup prevents users from entering invalid data.

Check the Credit Code Value

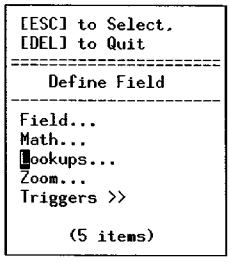
1. **Start the Form Painter and select Run 4GL Program from the Run pull-down menu.**
2. **From your Customer Entry program, use Find to select an existing customer.**
3. **Select Update and enter TTT in the Credit Code field.**

   Recall that TTT is not a record in the credit table. You only created four records in that table (AAA, BBB, CCC, and DDD). Despite this fact, however, the program still accepts TTT, a completely invalid value. You can use lookups to verify data in a field.
4. **Quit the Customer Entry program.**
5. **From the Form Painter, open the Customer Entry file (cust).**

Define the Lookup

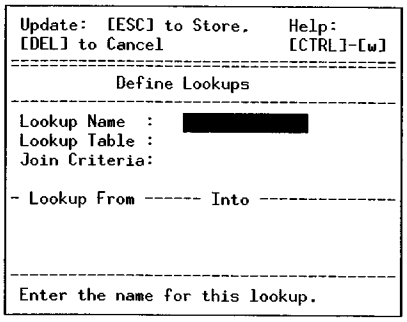
1. **From the Form Painter, place your cursor in the Credit Code field and press [CTRL]-[z].**

   The Define Field pop-up menu appears.



2. **Select Lookups... from the Define Field pop-up menu.**

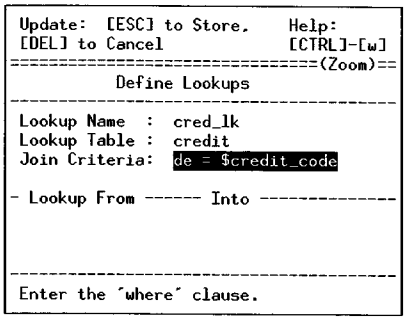
   The Define Lookups window appears.



3. **Enter `cred_lk` in the Lookup Name field.**

4. **Enter `credit` in the Lookup Table field.**

5. **Enter `credit.credit_code = $credit_code` in the Join Criteria field.**



6. **Press [ESC] to save your lookup.**

Save, Generate, and Compile

1. **Use the Save Form option under the File pull-down menu.**
2. **Select Generate 4GL from the Run pull-down menu.**

   The Generate 4gl: Enter Selection box appears.
3. **Select All Forms from this box.**

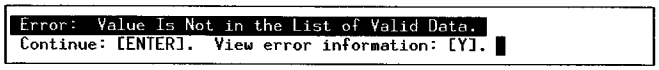
   The Local Forms Only box appears.
4. **Select YES**
5. **Select Compile 4GL from the Run pull-down menu.**

Run Your Customer Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.
2. **Find a customer and select Update.**
3. **Enter TTT in the Credit Code field.**

   An error message appears:



4. **Press [DEL] to return to the Credit Code field and enter a valid value (AAA).**

   This time the value is accepted and the cursor moves to the next field.
5. **Press [ESC] and select Quit to return to the Form Painter.**

Exercise 5B

Objective: To create a Credit Desc field that is linked to the Credit Code field. When the user specifies a Credit Code, the Credit Desc field will get filled automatically.

1. **Create a new field on the Customer Entry form called Credit Desc.**

   In other words, create a field label and press [to define it.

   On the Define Fields window, specify N in the Entry ? field.

```
Update: [ESC] to Store, [DEL] to Cancel          Help:
Enter changes into form                          [CTRL]-[w]
================================================================
                        Define Fields
----------------------------------------------------------------
Table Name :  credit                        Input Area :  1
Column Name:  credit_desc                   Entry ?    :  N
Field Type :  char(10)                      Autonext ? :  N
Message    :                                Downshift ?:  N
Picture    :                                Upshift ?  :  N
Display Fmt:                                Verify ?   :  N
Validate   :                                Required ? :  N
Default    :                                Skip ?     :  N
Translate  :
----------------------------------------------------------------
Enter a [N] if field is protected.
```

   When Entry ? is N, the user cannot enter/update the field.

2. **Press [ESC] to save the field definition.**

   You should now have the following fields on you Customer Entry program:

```
Form Editor:  [ESC] or [DEL] Command Line              [CTRL]-[w] Help
Press [CTRL]-[z] to update definition for field "credit_desc"
================================(cust)=========(Zoom)==(insert)==(4,53)===
------------------------ Customer Entry Screen -------------------------

Customer Number:[          ]          Credit Code:[    ]
Company Name:[                  ]     Credit Desc:[          ]
Contact Name:[              ] [                   ]

City:[              ]       State:[  ]       Postal Code:[     ]
Phone Number:[                ]
```

Save, Generate, and Compile

1. **Use the Save Form option under the File pull-down menu.**
2. **Select Generate 4GL from the Run pull-down menu.**

   The Generate 4gl: Enter Selection box appears.
3. **Select All Forms from this box.**

   The Local Forms Only box appears.
4. **Select YES.**
5. **Select Compile 4GL from the Run pull-down menu.**

Run Your Customer Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.
2. **Select Add to create a new customer entry.**
3. **In the Credit Code field, enter BBB.**

   Notice how the Credit Desc field is filled automatically.

```
Add:    [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
================================================================================
------------------------- Customer Entry Screen --------------------------

Customer Number:                        Credit Code: BBB
Company Name:  ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓       Credit Desc: GOOD
Contact Name:

City:                          State:          Postal Code:
Phone Number:
```

4. **Quit out of the Customer Entry program to return to the Form Painter.**

6

Input Areas and Specification Files

Main topics:

- Input Area Overview
- Creating Form Specification (*.per) Files

Input Areas Overview

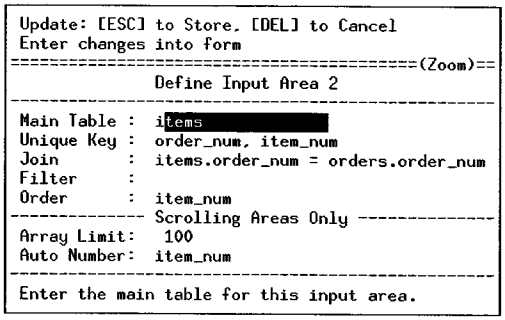
Input areas are where you specify characteristics about the header and/or detail portion of the form. The header portion is always given an input area equal to one and the detail portion is given an input area equal to two.

```
Input Area 1 = Header
Input Area 2 = Detail
```

You set input area characteristics with the Define Input Areas window. You can use the Input Areas option under the Define pull-down menu to access this window.

You set input area characteristics with the Define Input Areas window.

This example shows values for the detail portion of a form.



For a header/detail screen, you must specify characteristics about the detail portion of the form in order for the form to work properly. For instance, you must specify the join that connects the header table to the detail table.

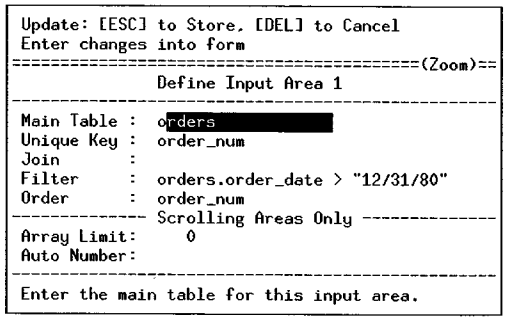
The most important field is the Join field. The Join field specifies how the header and detail tables are related. Use the following syntax to define the join between the two tables:

```
header_table.column = detail_table.column
```

Where *header_table.column* represents the name of the header column and *detail_table.column* represents the name of the detail column.

Another important field is the Unique Key field. This field specifies which columns uniquely define a row in the table.

If you do not specify the input area for the header, the Form Painter puts in default values for you.



When you define the header input area, you cannot enter the Join field. This field is only for detail input areas.

Creating Form Specification (*.per) Files

Every time you save a form image with the Form Painter, an Informix form specification (*.per) file is created. It is helpful for you to become familiar with this file.

The *Screen* Code Generator uses form specification (*.per) files to produce all the 4GL source code necessary to create an input program. In general, form specification files contain the following sections:

Section	Use
DATABASE	Specifies the database that the form is created and compiled against.
SCREEN	Contains the image of the form. Each input field is identified by a field tag.
TABLES	Identifies the tables that are used by the form.
ATTRIBUTES	Ties each field tag (in the SCREEN section) with a column in the table. Fields can also be classified as *formonly*. Formonly fields are not associated with columns of any database. They are used to enter or show the values of program variables. This section also contains Informix-related characteristics of the field (e.g., comments, required logic, verification logic, and formatting instructions).
INSTRUCTIONS	Specifies non-default field delimiters and defines screen arrays and records, such as the s record.
FOURGEN	Contains -specific instructions that are read by the *Screen* Code Generator. The *Screen* Generator builds the code logic based on what is specified in the section.

Once you become familiar with *CASE* Tools, you will learn how to read form specification files. You will learn how to recognize what the Form Painter creates in these files. A typical form specification file looks as follows:

```
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.lna
me,
    customer.company, customer.address1, customer.address2, customer.cit
y,
    customer.state, customer.zipcode, customer.phone, orders.order_date,
    orders.po_num, orders.order_num, orders.ship_instruct, orders.ship_w
eight,
    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_pr
ice)

delimiters "  "

{
########################################################################

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

defaults
    type    = header/detail
    init    = orders.order_num > 100
```

```
input 1
    table   = orders   (default = 1st table in the "tables" section)
    key     = order_num
    filter  = orders.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

}
```

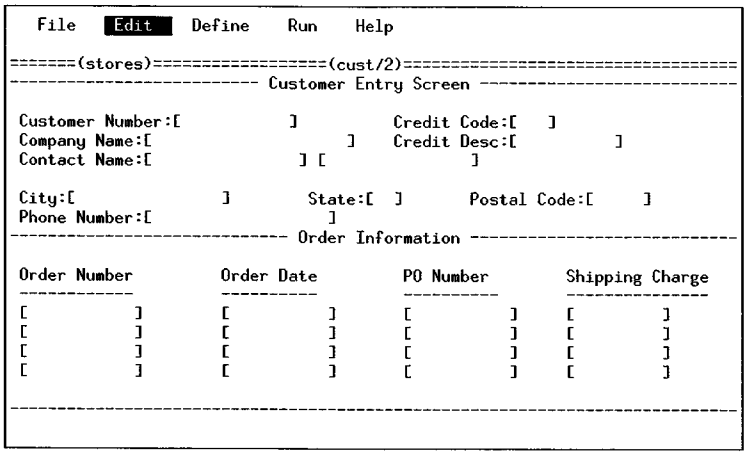
Section Summary

- All forms you create with the Form Painter contain input areas. Input areas correspond to the header and/or detail section of a form. The most important attribute that you set is the table attribute. It specifies which table the header portion of the form writes to and which the detail portion of the form writes to.
- The Form Painter creates an Informix form specification (*.per) file. As you become familiar with the Screen *CASE* Tools, you will learn how to read and alter form specification files.

Exercise 6

Objective: To convert the Customer Entry program from a header screen to a header/detail screen. The detail portion will write to a detail table, which is the "many" table in a one-to-many table relationship.

The detail portion will show data from the `orders` table. At the end of this exercise, your Customer Entry program will look as follows:



Change the Screen Type to Header/Detail

This exercise assumes you are already running the Form Painter with your `cust.per` form open. If this is not the case move to your program directory (`cd $HOME/labs/aw.4gm/i_cust.4gs`), start the Form Painter, and open `cust.per`.

1. **Select Form Defaults from the Define pull-down menu.**

   The Define the Form window appears. As you recall, this window specifies various characteristics about your form, including the screen type (which is set in the Form Type field).

2. **Change the Form Type field from header to header/detail.**

   This converts your form to a header/detail screen.

3. **Press [ESC] to store your change and close the window.**

Add the Detail Section

Now add a detail section called Order Information to your Customer Entry program.

1. **Creating a detail section title:**

   ---------- Order Information ----------

2. **Add the following field labels below the title:**

```
Order Number        Order Date        PO Number        Shipping Charge
------------        ----------        ---------        ---------------
```

3. **Place your cursor below the O in Order Number.**

4. **Press [.**

   The Define Fields window appears. Fields in this detail section correspond to the `orders` table. Remember that a detail section is considered Input Area 2.

5. **Define the Order Number field using the following values. (Note the Table Name and Input Area fields):**

```
Update: [ESC] to Store, [DEL] to Cancel          Help:
Enter changes into form                          [CTRL]-[w]
================================================================
                        Define Fields
----------------------------------------------------------------
Table Name :  orders                        Input Area :  2
Column Name:  order_num                     Entry ?    :  Y
Field Type :  serial not null               Autonext ? :  N
Message    :  Enter order number            Downshift ?:  N
Picture    :                                Upshift ?  :  N
Display Fmt:                                Verify ?   :  N
Validate   :                                Required ? :  N
Default    :                                Skip ?     :  N
Translate  :
----------------------------------------------------------------
Enter the input mask (picture) for this field. (no quotes)
```

6. **Press [ESC] to store these values and define the field.**

7. **Repeat these steps until you've created a complete row of detail fields.**

   Once you have a complete row, use the Mark, Copy, and Paste options to create three duplicate rows. As you recall, detail sections, much like zooms, display data in a row-by-row format.

   When you are finished you should have four detail lines with fields for the following columns:

   `orders.order_num`

   `orders.order_date`

   `orders.po_num`

   `orders.ship_charge`

   Your screen should look as follows:

```
Form Editor:  [ESC] or [DEL] Command Line                    [CTRL]-[w] Help
Update data entry image
=======(stores)==================(cust/2)========(Zoom)============(17,2)===
------------------------- Customer Entry Screen --------------------------

Customer Number:[           ]          Credit Code:[   ]
Company Name:[                    ]    Credit Desc:[          ]
Contact Name:[               ] [              ]

City:[              ]       State:[  ]       Postal Code:[     ]
Phone Number:[                 ]
---------------------------- Order Information ----------------------------

Order Number         Order Date          PO Number          Shipping Charge
------------         ----------          ----------         ---------------
[           ]        [          ]        [          ]       [          ]
[           ]        [          ]        [          ]       [          ]
[           ]        [          ]        [          ]       [          ]
[           ]        [          ]        [          ]       [          ]
▌
------------------------------------------------------------------------------
```

Define the Detail Input Area

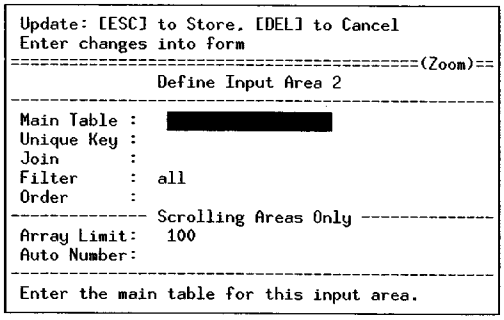
Once the image of the Customer Entry form's detail section is correct, you must define the Input Area.

1. **Select Input Areas from the Define pull-down menu.**

   The Input Area list box appears.

2. **Select Detail from the list box.**

   The Define Input Area 2 box appears.



3. **Specify orders as the Main Table.**

   Based on this value, the Unique Key value is automatically filled with the `order_num` value.

4. **In the Join field, enter:**

   `customer.customer_num = orders.customer_num`

5. **For now, disregard the other fields and press [ESC].**

   The Define Input Area 2 window closes.

Save, Generate, and Compile

1. **Use the Save Form option under the File pull-down menu.**
2. **Select Generate 4GL from the Run pull-down menu.**

   The Generate 4gl: Enter Selection box appears.
3. **Select All Forms from this box.**

   The Local Forms Only box appears.
4. **Select YES.**
5. **Select Compile 4GL from the Run pull-down menu.**

Run Your Customer Entry Program

1. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.
2. **Use Find to select all existing customers.**
3. **Use Nxt and Prv to scroll through the records.**

   As you scroll, notice how values from the orders table populate the detail section of the program. As you can see, some customers have made orders while others have not.
4. **Press [TAB] to move to the Detail section. When you are through, remain in your Customer Entry program. The next exercise starts from here.**

7

Working with the User Control Libraries

Main topics:

- User Control Library Overview
- Creating a To-Do List
- Adding Freefrom Notes
- Entering Error Messages
- Adding Help Text
- Defining Navigation Events
- Mapping Hot Keys to Navigation Events
- Setting up Hot Keys
- Logging Online Feature Requests
- Creating User-Defined Fields

User Control Library Overview

The User Control Libraries are a part of Enhancement Toolkit. These libraries provide a series of features that give your users more control over generated programs created by *Screen*.

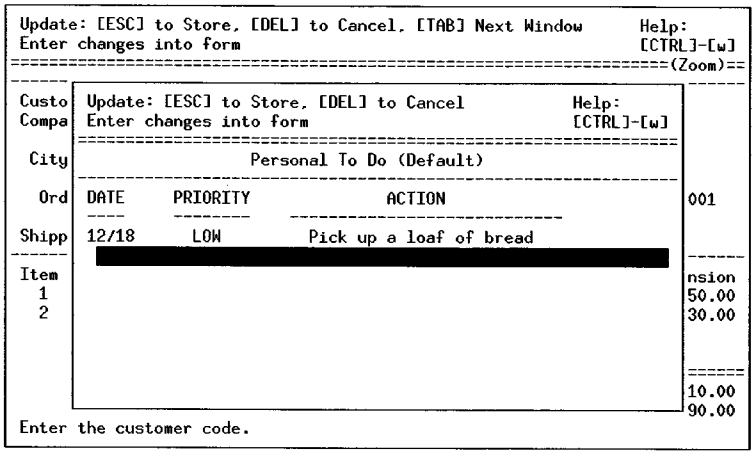
The User Control Libraries provide the following:

- a set of commonly-requested features that appear in programs you create with *Screen*.
- a set of features that makes supporting and servicing *Screen*-generated applications easier.

Creating a To-Do List

A To-Do List gives the user a note pad to track the tasks they need to complete. Users can access their To-Do List by pressing [CTRL]-[t] when they are running an input program. To-Do lists are attached to a user's login ID, so the user's To-Do List is available from every generated input program.

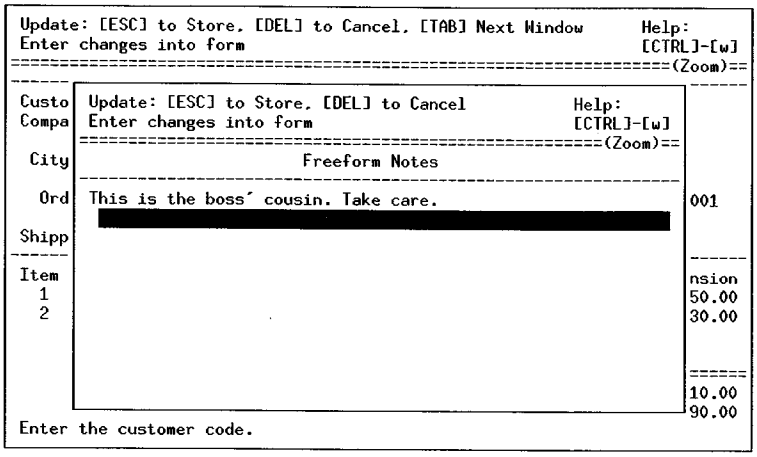
The To-Do List feature gives users a note pad to track the tasks they need to complete.



Adding Freeform Notes

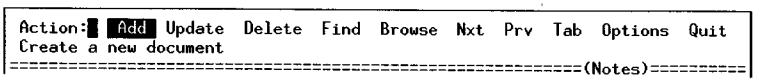
Freeform Notes let users place notes in a data-entry document. The user presses [CTRL]-[n] and adds the note. The note is bound to the header portion of the input program. When a user defines a note, the note is permanently attached, and other users can view it.

Freeform Notes let users place notes on a data-entry document.



When a noted is attached to a document, the Note lamp appears in the upper right portion of the screen.

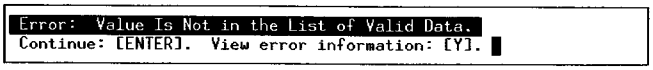
The Notes lamp indicates when a Freeform Note is attached.



Entering Error Messages

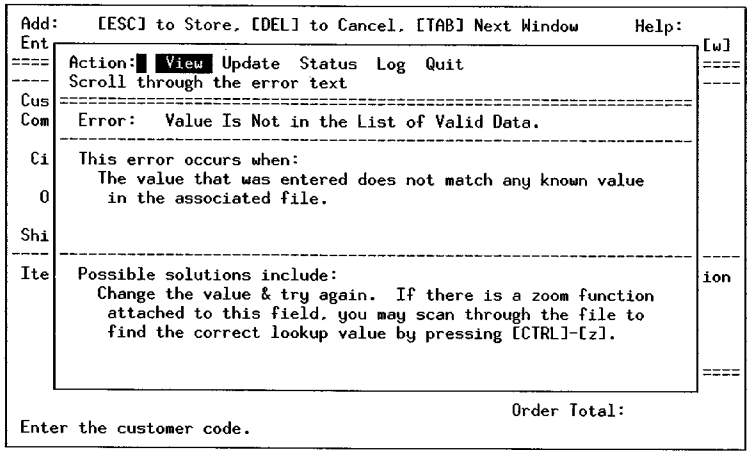
If an error occurs in a generated application, users see the following type of message.

If an error occurs, users see this type of error message.



To see more information about an error, users can press Y.

When users press Y, a more detailed description of the error appears.



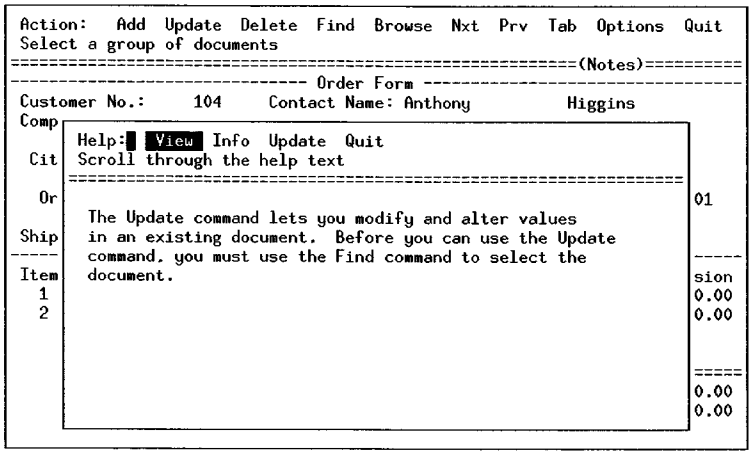
Users can use this window to check error information. In addition, users can log the errors they encounter and add more information describing the error to the error window.

You, as a programmer, may also want to add your own custom error messages.

Adding Help Text

Screen also provides a context sensitive help system, which both you and program users can update and modify. When users have questions about input fields, ring menu commands, or any program control, they can press [CTRL]-[w] to see help information.

Context sensitive help gives users the ability to access specific help information about input fields, ring menu commands, or any program control.

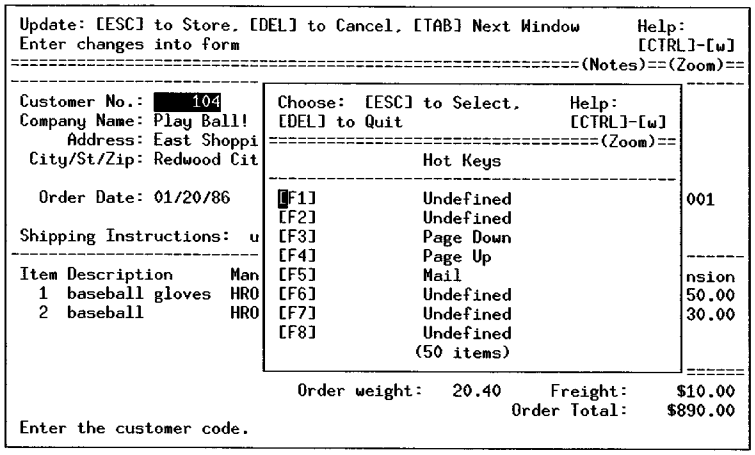


Setting up Hot Keys

Hot Keys let users map their keyboard to specific program events including custom Navigation events (see "Defining Navigation Events" on page 7-8). To access the Hot Keys pop-up menu, users can press [CTRL]-[e]. The Hot Keys pop-up menu serves three purposes:

1. It lets users see how their keys are mapped.
2. It lets users customize their work environment and change their default Hot Key settings.
3. It gives users the ability to assign their own Navigation events to Hot Keys.

Hot Keys let users map their keyboard to specific program events.



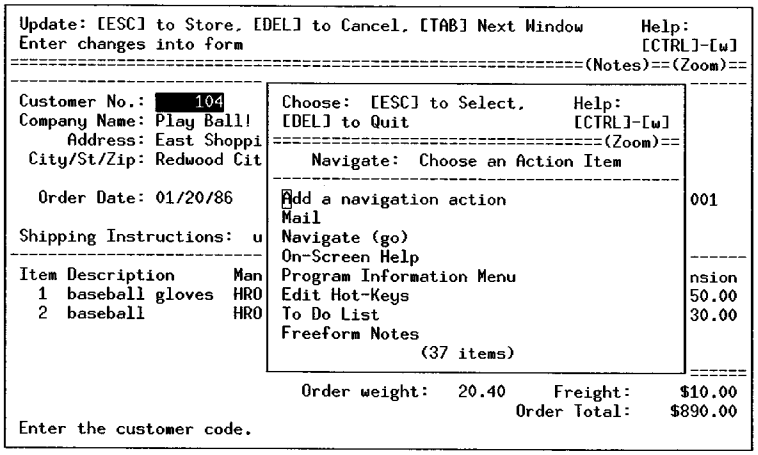
Hot Keys are defined in the Hot Keys window. To access the Hot Keys window, users must highlight the key they want to define on the Hot Keys menu and press [CTRL]-[z].

Defining Navigation Events

Navigation gives users the ability to define custom program events. These events can perform a number of useful tasks, such as suspending one program to jump to another one. Users can define Navigation events to go with an assortment of predefined Navigation events. When users press [CTRL]-[g], the Navigation pop-up menu appears.

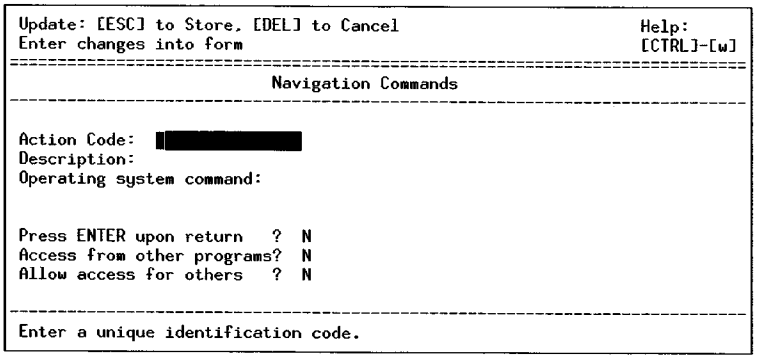
The Navigation pop-up menu lets users select from a list of predefined Navigation events.

Users can use this menu to create Navigation events.



Users can add Navigation events by selecting "Add a navigation action" from the Navigation menu.

The Navigation Commands window lets users define new Navigation events.



You must name your Navigation event in the Action Code field. You also need to describe your event in the Description field. If you are entering an operating system event, enter the operating system command in the "Operating system command" field. For example, if this events starts another program, enter the program command in this field.

The remaining fields on the Navigation Commands window are Y/N fields. "Press ENTER upon return" makes the user press [ENTER] once the event terminates. The "Access from other programs" field specifies whether this event can be run from other programs or not. The final field, "Allow access for others" specifies if others can use this event.

To define a Navigation event:

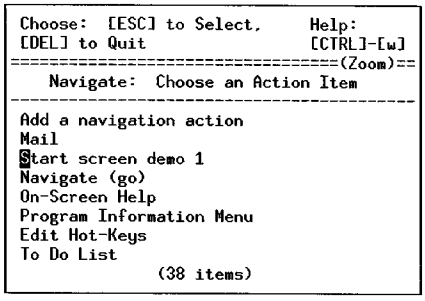
1. **Press [CTRL]-[g] to open the Navigation pop-up menu.**
2. **Select "Add a navigation action" from the menu.**

   The Navigation Commands window appears.

3. **Complete the Navigation Commands window and press [ESC].**

   Once you define a Navigation event, it appears on your Navigation menu.

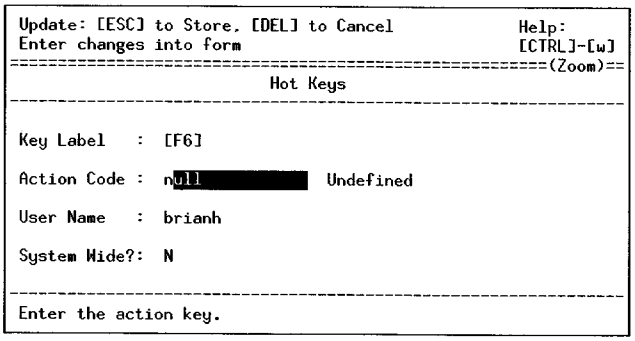
This example shows the Start screen demo 1 event, which is a custom Navigation event defined by the user.



Mapping Hot Keys to Navigation Events

You can also combine the power of Hot Keys and Navigation by defining custom Hot Keys to operate your Navigation events. You set Hot Keys to work with Navigation events in the Hot Keys window.

The Hot Keys window lets you assign Hot Keys to Navigation events.



The most important field is the **Action Code** field. This field corresponds to the Action Code you gave the event in the Navigation Commands window (page 9). The System Wide? field specifies if the Hot Key is available to all system users.

To map a Hot Key to a Navigation event:

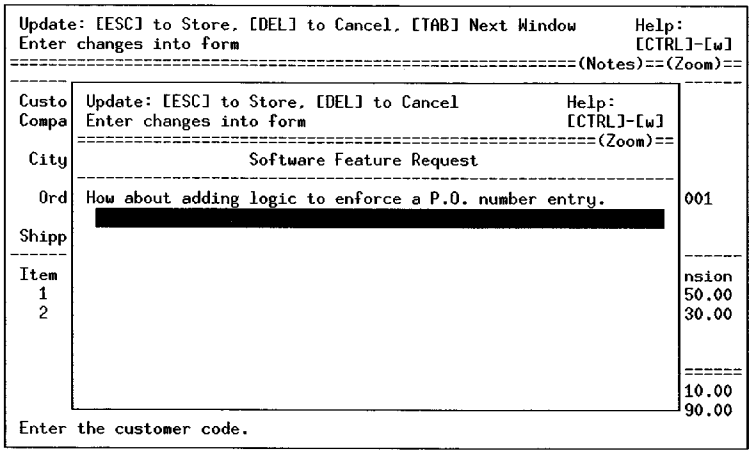
1. **Define your custom Navigation event.**
2. **Press [CTRL]-[e] to open the Hot Keys pop-up menu.**
3. **Highlight an undefined key and press [CTRL]-[z].**

   The Hot Keys window appears.
4. **Complete the Hot Key window and press [CTRL]-[z].**

Logging Online Feature Requests

Online Feature Requests let users communicate with you about the features they want. When users press [CTRL]-[y] and select Software Features, the Software Features Request window appears.

Online Feature Requests let users communicate to you what features they want.



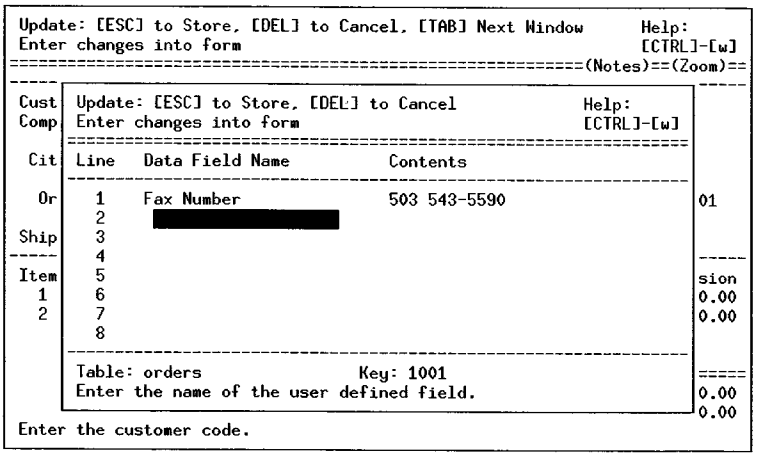
Once a request is entered, it gets appended to the `errolog` file, which you can review.

Creating User-Defined Fields

User-Defined Fields give you or your users the ability to add fields on the fly. When users press [CTRL]-[f], the User-Defined Fields window appears. Once a new field is defined, the User-Defined Fields window appears for every document that is created or updated.

User-Defined Fields give users the ability to add fields on the fly.

This example shows a user adding a Fax Number field.



User-Defined Fields are not physically entered into a column on the header or detail table. A separate table stores these field labels and contents. If users define a number of fields, you should consider adding those fields to the input program with the Form Painter.

Section Summary

- The User Control Libraries are a part of Enhancement Toolkit. These libraries provide a series of features that give your users more control over generated programs created by the *Screen*.
- A To-Do List gives users a note pad to track the tasks they need to complete.
- Freeform Notes let users place notes on a data-entry document.
- Users can use the Error Message window to check error information. In addition, users can log the errors they encounter and add more information describing the error to the Error window.
- *Screen* also provides a context sensitive help system, which both you and program users can update and modify.
- Hot Keys let users map their keyboard to specific program events including custom Navigation events.
- Navigation gives users the ability to define custom program events. These events perform a number of useful tasks, such as suspending one program to jump to another one.
- You can combine the power of Hot Keys and Navigation by defining custom Hot Keys to operate your Navigation events.
- Online Feature Requests let users communicate with you about the features they want.
- User-Defined Fields give you or your users the ability to add fields on the fly.

Exercise 7A

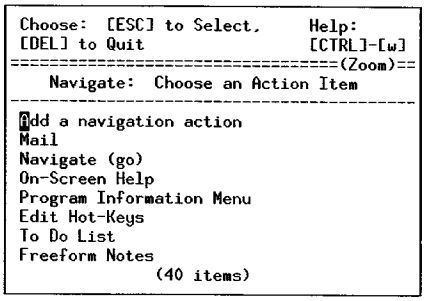
Objective: To place a navigation event in your Customer Entry program. You will add an event to check the amount of disk space available on your computer.

Access the Navigation Menu

This exercise starts from your running Customer Entry program. If not done already, start this program.

1. **From anywhere within your Customer Entry program, press [CTRL]-[g].**

   The Navigate pop-up menu appears.



   As you can see, this menu already has several navigation events already defined. You can select any of these events to see what they do.

2. **Select Add a navigation event (option one) from the Navigate menu.**

   The Navigate Commands window appears.

Enter a Navigation Command to Check Disk Space

1. **Using the Navigate Commands window, set Action Code to `check_disk`.**

   The Action Code field contains a unique name for the event you are defining. You should try to make this name as descriptive as possible.

2. **Set Description to Check Disk Space.**

3. **Set the Operating system command field to the UNIX command that checks your disk space (typically the `df` command).**

4. **Enter a Y in the Press ENTER upon return? field.**

   When the `df` command is executed, it will return to the program. Many times commands, such as `df`, return too quickly. Therefore, the Press [ENTER] prompt pauses after the UNIX command terminates so you can read its output.

5. **Press [ESC] to save `check_disk`.**

Run check_disk

1. **Invoke the Navigate menu again by pressing [CTRL]-[g].**

   Notice how `check_disk` appears as the second option on the menu.

2. **Select `check_disk`.**

   The `df` command runs and its output is displayed to the screen. Once complete, the Press [ENTER] prompt appears.

3. **Press [ENTER] to return to your program.**

Edit check_disk

You can always edit a navigation event.

1. **Invoke the Navigate menu again (press [CTRL]-[g]).**
2. **Highlight the Check Disk Space option and press [CTRL]-[z].**

   The Navigate Commands window appears.
3. **Edit `check_disk` or press [ESC] to save it as is.**
4. **Remain in your Customer Entry program and continue to Exercise 7B.**

Exercise 7B

Objective: To create a navigation event that runs a separate program.

Access the Navigation Menu

1. **From anywhere within your Customer Entry program, press [CTRL]-[g].**

   The Navigate pop-up menu appears.

2. **Select Add a navigation event (option one) from the Navigate menu.**

   The Navigate Commands window appears.

Add an Event to Call the Credit Entry Program

1. **Set Action Code to `credit_program`.**
2. **Set Description to Run Credit Info Program.**
3. **Set Operating system command to:**

   For RDS users:

   ```
   cd $HOME/labs/aw.4gm/i_cred.4gs; fglgo i_cred.4gi
   ```

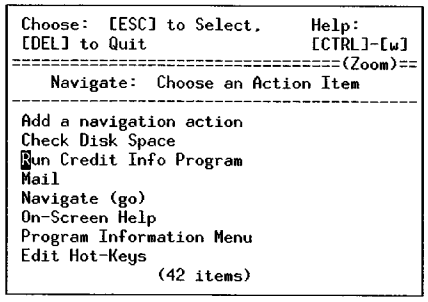
   This command changes to the `i_cred.4gs` directory and starts the Credit Entry program.

4. **Press [ESC] to save credit_program.**

   Note that you do not have to set the Press [ENTER] upon return field. When you exit the Credit program, you return directly to the Customer Entry program.

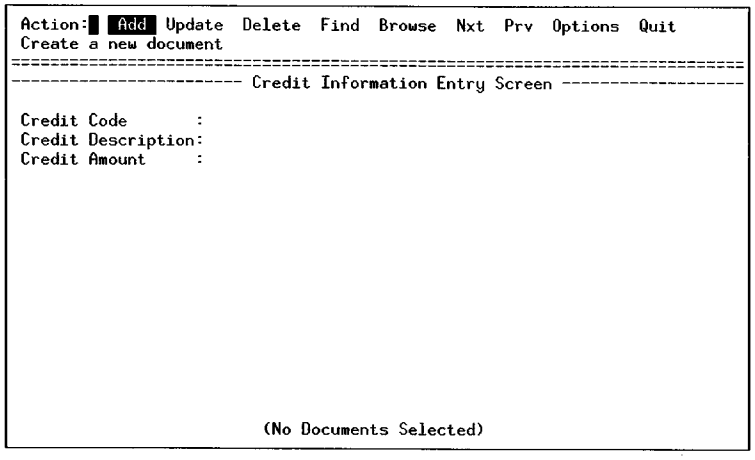
Use the credit_program Event

1. **Initiate the Navigate pop-up menu.**



2. **Select Run Credit Info Program.**

   The Credit Entry program, which you created in Exercise 3, starts.



3. **Select Quit from the Credit Entry program's ring menu.**

   The Credit Entry program exits and you return to the customer Entry program.

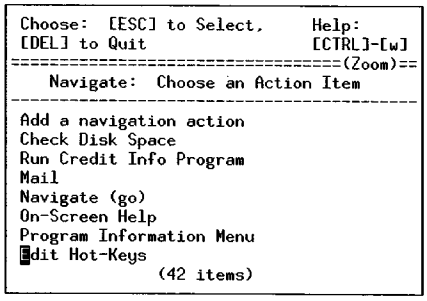
Exercise 7C

Objective: To map a hot key to the `credit_program` event.

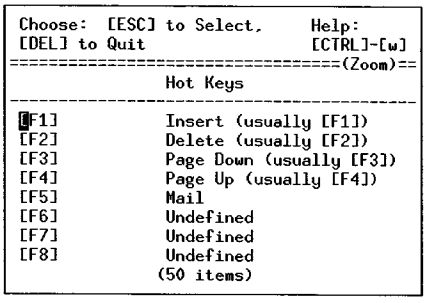
Edit Hot Keys

1. **From Customer Entry, initiate the Navigate pop-up menu again (press [CTRL]-[g]).**

2. **Select the Edit Hot-Keys option.**



The Hot Keys pop-up menu appears.



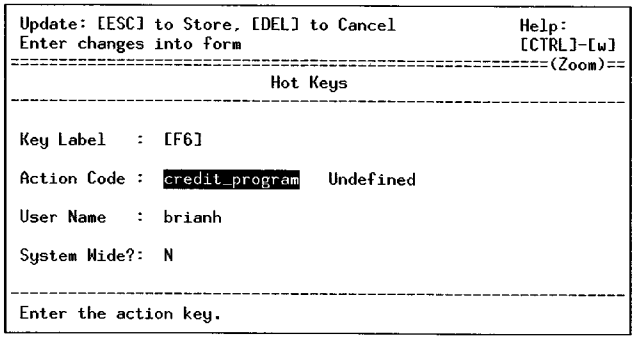
3. **Highlight [F6] (which is Undefined) and press [CTRL]-[z].**

   The Hot Keys window appears.

Enter the Navigation Event Codes

1. **Set the Action Code field to `credit_program`.**

If you forget the Action Code, you can Zoom on this field.

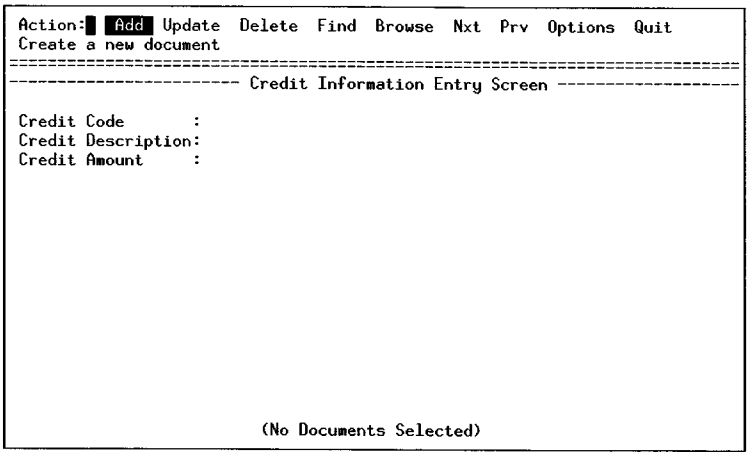


2. **Press [ESC] to save the [F6] hot key mapping.**

Press [F6] to Start the Credit Info Program

1. **From anywhere in your Customer Entry program, press [F6].**

   The Credit Entry program starts.



2. **When finished, exit the Credit Entry program and return to the Customer Entry program.**

Edit a Hot Key Definition

If you ever need to remap a hot key you can change its definition.

1. **Press [CTRL]-[e] to initiate the Hot Keys pop-up menu.**

   The [CTRL]-[e] sequence lets you access this menu directly, you can also select Edit Hot-Keys from the Navigate pop-up menu.

2. **Highlight the Hot Key you want to edit. For example, highlight the [F6] key.**

3. **Press [CTRL]-[z] to bring up the Hot Keys window.**

```
Update: [ESC] to Store, [DEL] to Cancel              Help:
Enter changes into form                              [CTRL]-[w]
=========================================================(Zoom)==
                          Hot Keys
------------------------------------------------------------------

Key Label   : [F6]

Action Code : credit_program  Run Credit Info Program

User Name   : brianh

System Wide?: N

------------------------------------------------------------------
Enter the action key.
```

4. **From the Hot Keys window, you can edit the Action Code value.**

   For this exercise, do not change the [F6] hot key. It is enough for you to know how to edit the values in this window.

5. **Press [ESC] to return to the Customer Entry program.**

6. **Quit out of both the Customer Entry program and the Form Painter.**

8

Using the Screen *Code Generator*

Main topics:

- *Screen* Code Generator Overview
- Understanding Library Code and Local Code
- Classifying Functions
- Starting the Tools from the Command Line
- Using the Tags Feature

Screen Code Generator Overview

The *Screen* Code Generator functions as the back-end to the Form Painter. You use the Form Painter to create a form image and the *Screen* Code Generator to create code based on that form image.

The *Screen* Code Generator relies on a form specification (*.per) file to create the 4GL source code. When you save a form image with the Form Painter, a *.per file is created automatically. In a general sense, you must complete the following steps to develop an input program:

1. Create a form image with the Form Painter.
2. Save your form image in the Form Painter to create a form specification (*.per) file.
3. Invoke the *Screen* Code Generator, which reads the *.per file and creates INFORMIX-4GL source code based on the instructions in the specification file (see "Creating Form Specification (*.per) Files" on page 6-4).
4. Use the make utility (`fg.make`) to compile the source code into object code and then link it into a (*.4ge) executable or (*.4gi) pseudo code.

Note

The make utility (or `fg.make`) produces either a (*.4ge) executable or (*.4gi) pseudo code file depending on the type of development system you are using. If you are using the Informix C compiler, `fg.make` creates a *.4ge executable file. If you are using the RDS compiler, `fg.make` creates a *.4gi pseudo code file.

5. Run the input program and use its ring menu to add, update, and delete data from the database.

The following figure outlines the steps you take to develop a complete input program using *Screen*.

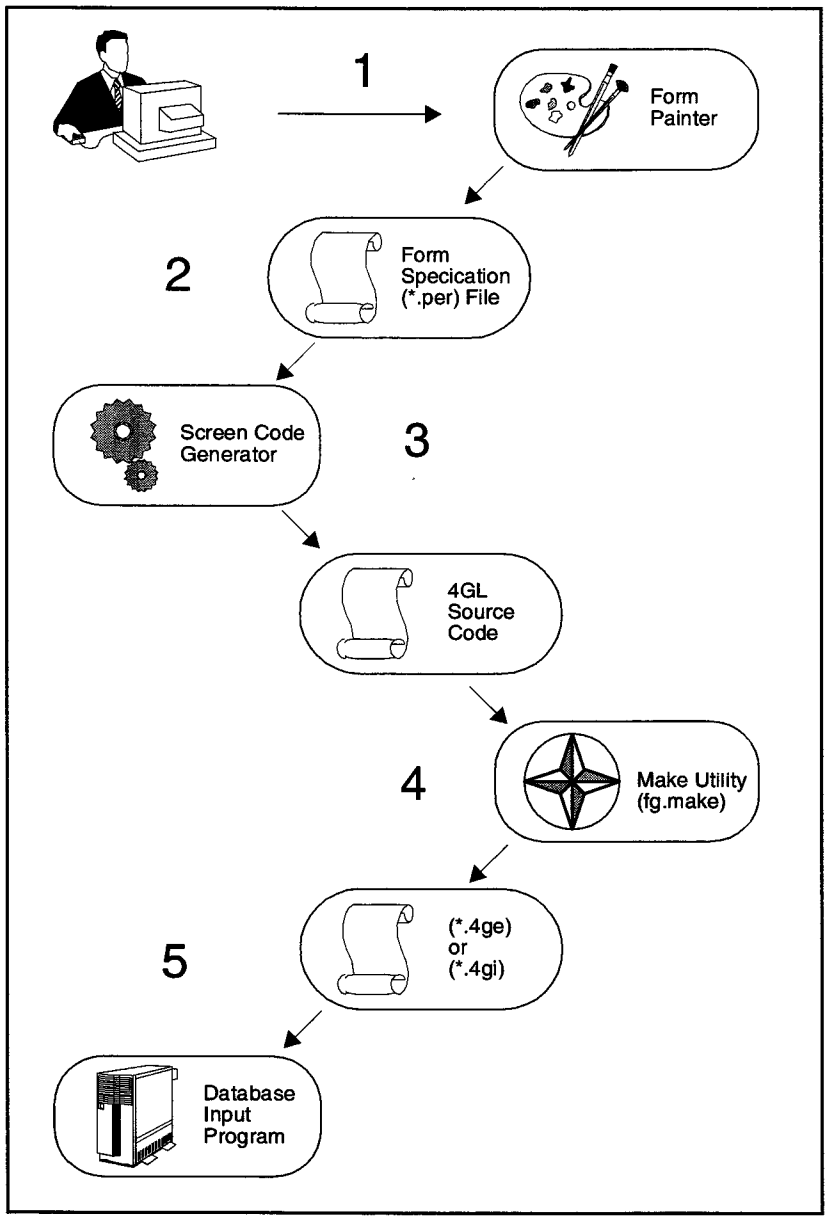
Step 1: Create a form image with the Form Painter.

Step 2: Save your form image to create a form specification (*.per) file.

Step 3: Invoke the *Screen* Code Generator to read the *.per file and create 4GL source code.

Step 4: Use the make utility (fg.make) to compile the source code into object code and link it into a (*.4ge) executable or (*.4gi) pseudo code file.

Step 5: Run the input program and use its ring menu to add, update, and delete data from the database.



Files Created During the Development Process

During the development process, there are several files that get created. Each file is given a special file extension to help you identify its file type.

Tool	File Type	File Extension
Form Painter	Form Specification Files	*.per
Screen Code Generator	Compiled Form Files	*.frm
Screen Code Generator	INFORMIX-4GL Source Code Files	*.4gl
`fg.make`	Compiled Object Files	*.4go or *.o
`fg.make`	Executable Files	*.4gi or *.4ge

For example, if you build screen demo 3, these files are created using INFORMIX-RDS:

Form Specification	Compiled Form	Source Code	Object	Executable	Other
browse.per	browse.frm	browse.4gl	browse.4go	screen3.4gi	Makefile
cust_zm.per	cust_zm.frm	cust_zm.4gl	cust_zm.4go		errlog
order.per	order.frm	detail.4gl	detail.4go		filelist.RDS
stk_mnu.per	stk_mnu.frm	globals.4gl	globals.4go		tags
stockzm.per	stockzm.frm	header.4gl	header.4go		
		main.4gl	main.4go		
		midlevel.4gl	midlevel.4go		
		stk_mnu.4gl	stk_mnu.4go		
		stockzm.4gl	stockzm.4go		

Note also provides a code merge (fglpp) utility that creates original (*.org) files (see "Featurizer Overview" on page 18-2).

Understanding Library Code and Local Code

You can classify code into two main categories:

1. Library Code
2. Local Code

Library code has the following characteristics:

- It is shared by different programs.
- It is static; the code never changes.
- It is data independent.
- It is generic.
- It is not created by the *Screen* Code Generator. Library code is hand-coded and always available for use.

Many program features, such as the ring menu commands, are created from library code:

Local code has the following characteristics:

- It is used by only one program.
- It is designed to change over time.
- It is data dependent.
- It is specific.
- It is created by the *Screen* Code Generator.

There are several visible examples of local code as well, such as reading in a record, adding a record, and saving a record.

Classifying Functions

code is highly modular, which means that all the code is written within functions. Most of these functions are small, less than 20 lines long. functions have the following characteristics:

- All code is organized into logical code blocks.
- Possible points of modification are easily identifiable.
- All functions contain comments that describe specifically what they do.
- Function code can be reused.
- Generated functions have similar names, thus establishing consistent naming conventions.

Functions are classified according to thier use. functions can be divided into three classes:

1. Upper-level Functions
2. Low-level Functions
3. Mid-level Functions

Upper-level functions have the following attributes:

- they are data independent.
- they are generic.
- they are usually library functions.
- they are not created by the *Screen* Code Generator.
- they are typically left unchanged.
- they are usually prefixed with `ring_`.

Low-level functions have the following attributes:

- they are data dependent.
- they are specific.

- they are created by the *Screen* Code Generator.
- they are frequently changed.
- they are usually prefixed with `llh_` or `lld_`.

Midlevel functions have the following attributes:

- they perform *housekeeping* tasks, such as initializing variables, preparing cursors, and performing construct statements.
- they are created by the *Screen* Code Generator.
- they are typically left unchanged.
- they are always prefixed with `mlh_` or `mld_`.

Starting the Tools from the Command Line

In chapter 2, you learned how to start the Form Painter and run the *Screen* Code Generator and FourGen make utility from within the Form Pianter. These programs can also be run from the UNIX command line.

Tool	Command
Form Painter	`fg.form`
Screen Code Generator	`fg.screen`
Make Utility	`fg.make`

Each command also uses several command flags that you can use to alter how the command works.

Form Painter Command Syntax

The Form Painter uses the following command flags and syntax.

`fg.form [-dbname` *`database`*`] [-o{0-5}] [-f] [-y|-n] [-p` *`file`*`.per]`

`-dbname` *`database`*	Specifies the database on which the Form Painter operates.
`-o{0-5}`	Specifies the level of information displayed during code generation. To display the least amount of information use `-o0`. To display the greatest amount of information use `-o5`.
`-f`	Specifies a *fast* generation. The `-f` flag and `-o0` are synonymous.
`-y|-n`	Specifies interactive or non-interactive generation mode. The `-y` flag answers yes to all code generation prompts.
`-p` *`file`*`.per`	Specifies the name of the form specification file to automatically loads upon start-up.

Screen Code Generator Command Syntax

The *Screen* Code Generator uses the following command flags and syntax.

`fg.screen [-dbname database] [-o{0-5}] [-f] [-y|-n] [file.per...]`

`-dbname database`	Specifies the database on which the *Screen* Code Generator operates.
`-o{0-5}`	Specifies the level of information displayed during code generation. To display the least amount of information use `-o0`. To display the greatest amount of information use `-o5`.
`-f`	Specifies a *fast* generation. The `-f` flag and `-o0` are synonymous.
`-y|-n`	Specifies interactive or non-interactive generation mode. The `-y` flag answers yes to all code generation prompts.
`file.per...`	Specifies the name(s) of the form specification file(s) that the *Screen* Code Generator reads and processes.

For a description and the syntax of the `fg.make` script see "Compiling Generated Code" on page 17-2. And for a description of code merging utility (`fglpp`) see "Featurizer Overview" on page 18-2.

Using the Hypertext Feature

The hypertext feature lets you quickly view functions. It is used when you are viewing source code and come across a function that is unfamiliar. Hypertext lets you jump to the body of the function to view it.

Hypertext is particularly useful for library functions. If you come across an unfamiliar library function, you can make the computer do the work of finding the function for you.

Setting up Tags

Hypertext makes use of the tags capability in the vi text editor. Various index files (called tags files) are created with *Screen* so you can jump between functions.

In your home directory, open your `.exrc` file. This file holds all your custom vi settings. There is a setting called tags that you must set up in your `.exrc` file before you can take advantage of hypertext. The tags setting merely points to the tags file in your local directory and any library directories.

Add the following line to your `.exrc` file:

```
set tags=tags\ ../tags\ /usr/fourgen/lib/tags
```

Subtitute the path name of your `$fg` variable for `/usr/`

Also add these lines:

```
map ] ^]

map [ ^^
```

To type a [CTRL] key character in a vi file, you must first type [CTRL]-[v]. In other words, to enter the lines above, you should use the following keystrokes:

```
map ] [CTRL]-[v] [CTRL]-[]]

map [ [CTRL]-[v] [CTRL]-[^]
```

Using Hypertext

There are three ways to use hypertext:

1. You can use vi in command mode and issue a tag command and the function name.
2. You can pass the `-t` flag and function name when initiating vi.
3. You can open a source code file, select the first character of the function call, and press the left ([) bracket key to jump to the function.

To use the tag command in a vi file:

1. **Press [ESC] to place vi into command mode.**
2. **Type `:tag` *`function-name`* and press [ENTER].**

   Where *function-name* represents the function you want to jump to. For example:

   ```
   :tag lib_before
   ```

   To return to your starting location, type:

   ```
   :e#
   ```

To use the `-t` flag from the command line, type:

```
vi -t function-name
```

To use hypertext from within a source code file:

1. **Place the cursor on the first character of the function call.**
2. **Press the left bracket ([) key.**

   You immediately jump to the function. To return to your starting location, press the right bracket (]) key.

Section Summary

- The *Screen* Code Generator functions as the backend to the Form Painter. You use the Form Painter to create a form image and the *Screen* Code Generator to create code based on that form image.
- During the development process, there are several files that get created. Each file is given a special file extension to help you identify its file type.
- You can classify code into two main categories: (1) Library Code and (2) Local Code.
- code is highly modular, which means that all the code is written within functions. Most of these functions are small, less than 20 lines long.
- Functions are classified according to their use. functions can be divided into three classes: (1) Upperlevel Functions, (2) Lowlevel Functions, and (3) Midlevel Functions.
- The Form Painter, *Screen* Code Generator, and make utility can be run from the UNIX command line.
- The hypertext feature lets you quickly view functions. It is used when a you are viewing source code and come across a function that is unfamiliar. Hypertext lets you jump to the body of the function to view it.

Exercise 8A

Objective: To build the Customer Entry program from outside the Form Painter. You will rebuild the entire application from the form specification (*.per) files that you created with the Form Painter.

Make a Backup Directory

1. **Move to the `$HOME/labs/aw.4gm` directory:**

   ```
   cd $HOME/labs/aw.4gm
   ```

2. **Create a `i_cust.bak` directory to hold a copy the files in your `i_cust.4gs` directory:**

   ```
   mkdir i_cust.bak
   ```

3. **Copy all of the files in `i_cust.4gs` to `i_cust.bak`:**

   ```
   cp i_cust.4gs/* i_cust.bak
   ```

4. **Move to your `i_cust.4gs` directory:**

   ```
   cd i_cust.4gs
   ```

Remove Everything Except Your *.per Files

1. **Remove all the files in `i_cust.4gs` except those with a *.per extension:**

   ```
   cp *.per ../
   rm *
   mv ../*.per ./
   ```

   This command leaves i_cust.4gs with two files: `cred_zm.per` and `cust.per`.

2. **List your files to verify that only these two files remain:**

   ```
   ls
   ```

Generate 4GL Code

The `cred_zm.per` and `cust.per` files contain all the information that is needed for the *Screen* Code Generator to re-create source code for your Customer Entry program.

1. **From the `i_cust.4gs` directory, enter:**

   ```
   fg.screen -o0 -y *.per
   ```

   The `-o` flag specifies the amount of screen output to display. A 0 indicates the minimum amount of output. A 5 indicates the maximum amount. Finally the `-y` flag automatically answers "yes" to all prompts.

   The *Screen* Code Generator reads the instructions in the *.per files and creates 4GL source code. When the *Screen* Code Generator is finished, the UNIX prompt reappears.

2. **From the UNIX prompt, list the files in i_cust.4gs:**

   ```
   ls
   ```

   As you can see, the *Screen* Code Generator creates a number of files, including a Makefile and multiple source code (*.4gl) files.

Compile the Code

After generating code, you must convert it into object code, link it to the libraries, and build an executable. All these tasks are handled by the compilation utility, which is known as `fg.make`.

1. **From the `i_cust.4gs` directory, enter:**

   ```
   fg.make
   ```

   The `fg.make` utility runs. When it is finished, the UNIX prompt reappears.

2. **List your files again:**

   ```
   ls
   ```

   Notice that now there are object files (*.4go) and a program file (*.4gi or *4ge). Which set of files you see depends on your Informix development type.

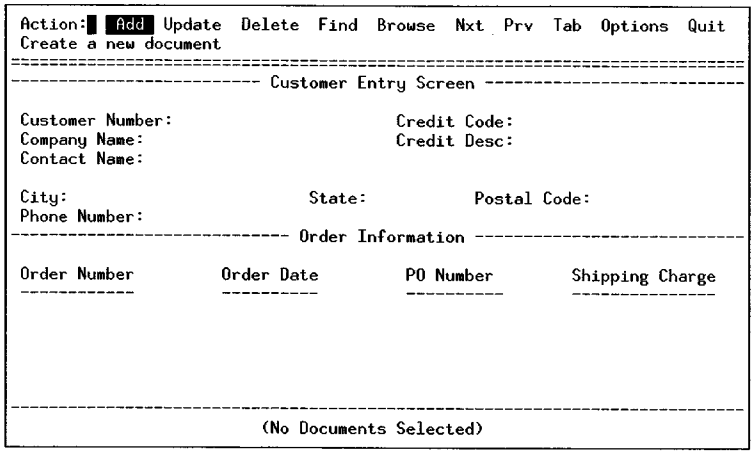
Run the Customer Entry Program

Once `fg.make` is finished, you can run the Customer Entry program again.

1. **Use the following command to run the Customer Entry program:**

   ```
   fglgo i_cust.4gi
   ```

   The Customer Entry program starts.



2. **Quit the Customer Entry program.**

Exercise 8B

Objective: To gain a basic knowledge of the INFORMIX-4GL source code built by the *Screen* Code Generator and to become familiar with FourGen standards and code structures.

List the Files

- **List the files in i_cust.4gs:**

  ```
  ls
  ```

  Notice that there are several files with a *.4gl extension. These are source code files.

Examine midlevel.4gl

1. **Use vi to open `midlevel.4gl`:**

   ```
   vi midlevel.4gl
   ```

   This file contains generated source code that handles "housekeeping" chores such as initializing variables, preparing cursors, and locking records.

   Notice how all the code is contained in functions. source code is extremely modular.

   Each function in `midlevel.4gl` is prefaced with `ml`. These characters stand for midlevel. Both the header and detail portion of Customer Entry have midlevel functions associated with them. For this reason, midlevel functions are further classified as `mlh` and `mld`, which stand for midlevel header and midlevel detail respectively.

2. **Exit from `midlevel.4gl`.**

Examine header.4gl and detail.4gl

1. **Use vi to look through both `header.4gl` and `detail.4gl`.**

   Both files contain lowlevel functions. The `header.4gl` lowlevel functions handle header section activities such as inserting, updating, deleting, and validation checking. The `detail.4gl` lowlevel functions do much of the same, but they control the detail portion of the screen.

   Notice how each `header.4gl` function names are prefaced with `llh` and `detail.4gl` functions are prefaced with `lld`.

2. **Exit these files.**

Examine cred_zm.4gl

1. **Use vi to open `cred_zm.4gl`**

   This file corresponds to your Credit Information zoom screen. Notice that there are sets of functions, prefaced by different capital letters that perform different tasks.

Preface	Use
A	Opens a Zoom window.
Q	Queries for selection criteria.
R	Reads records into the program.
D	Displays records to the zoom screen.
Z	Closes the zoom screen.

2. **Exit cred_zm.4gl.**

Exercise 8C

Objective: To use hypertext capability to find functions.

Set up Your .exrc File

1. **Use vi to open the .exrc file in your home directory:**

   ```
   vi $HOME/.exrc
   ```

2. **Add the following line to your .exrc file:**

   ```
   map ] ^]
   map [ ^^
   set tags=tags\ ../tags\ /usr/        /lib/tags
   ```

   The ^ characters are created in a special way. First press [CTRL]-[v], then type subsequent key. For example, to add the first two lines (`map ] ^]` and `map [ ^^]`), use the following key strokes:

   ```
   map ] [CTRL]-[v] [CTRL]-[]]
   map [ [CTRL]-[v] [CTRL]-[^]
   ```

3. **Save your .exrc file.**

4. **Make the values in this file current:**

   ```
   . $HOME/.exrc
   ```

Jump to a Function

1. **Use vi to open `header.4gl`.**

2. **Find the function call to `error_handler`.**

3. **Place your cursor on the first letter (an e) in `error_handler` and press the right bracket] key.**

   Your cursor jumps to the `error_handler` function. If your .exrc file is not set properly, you'll get the message:

   ```
   error_hanler: No such tag in tags file
   ```

4. **Return by using the left bracket [, then exit from `header.4gl`.**

9

Creating Triggers

Main topics:

- Trigger Overview
- Understanding the Trigger Concept
- Creating Triggers
- Merging Triggers into Code

Trigger Overview

In most cases, you can use the Form Editor in the Form Painter to accomplish everything an input program requires. The Form Editor lets you:

- define input fields
- specify field attribute logic, such as whether the field can be entered
- attach zoom screens
- attach lookups to validate input values

On occasion, however, you must make custom enhancements to an input program that you cannot create in the Form Editor. For example, you might want to include some of the following enhancement types:

- after field logic
- before field logic
- after input logic
- after change in logic
- before input logic
- after row logic
- before row logic
- event handling logic

You can create all these enhancements using triggers, which are essentially code-level modifications to an input program.

Understanding the Trigger Concept

Triggers are enhancements made directly to the source code generated from the *Screen* Code Generator. A trigger is an automatic way of placing code-level enhancements into the source code.

Triggers are named for logical points in the code. The following list contains some common triggers:

- `after_field`
- `before_field`
- `after_input`
- `before_input`
- `on_event`

Triggers get placed in trigger (*.trg) files. A trigger file functions much like a form specification (*.per) file. Both contain instructions that the *Screen* Code Generator reads and understands.

A single trigger file can contain more than one trigger.

Triggers do not require you to be an expert on code structure. You simply work with the Form Painter to define the logical points at which your triggers act.

Trigger (*.trg) files should have the same name as the form specification file that they relate to. For example, the `order.trg` file relates to the `order.per` form specification file.

Creating Triggers

Creating triggers is a straightforward task. There are two ways you can construct triggers:

1. You can use the Form Painter.
2. You can create them by hand in trigger (*.trg) files.

Perhaps the best way to write your first trigger is with the Form Painter; it provides the simplest environment to learn about trigger creation.

Using the Form Painter to Create a Trigger

Before you create a trigger using the Form Painter, you should create a form image and form specification file (see "Creating a Form Image" on page 2-5).

Once you create a program from which to work, you can define a trigger.

To add a new trigger using the Form Painter:

1. **Select Triggers >> from the Define pull-down menu.**

   If your screen type contains more than one input area, the Choose a Trigger Class pop-up menu appears.

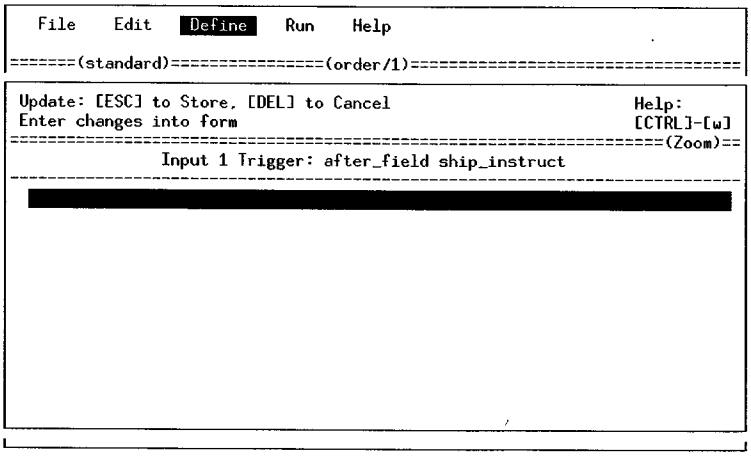
2. **Select the input area for your trigger.**

   The Choose a Trigger pop-up menu appears.

3. **Select the trigger you want to create.**

   Depending on the trigger you select, subsequent pop-up menus appear. For example, if you select the `after_field` trigger, the Choose a Field pop-up menu appears. After you choose a field, the Form Painter opens the Trigger Editor.

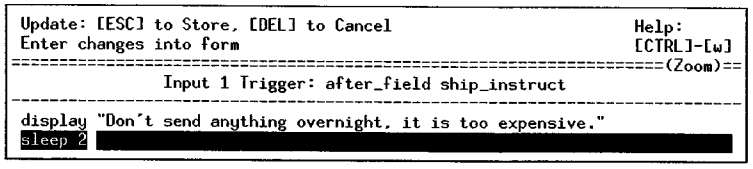
Use the Trigger Editor to enter custom 4GL logic.



4. **Enter the custom 4GL logic of your trigger using the Trigger Editor.**

   For example, after the shipping instruction field, you might want to display shipping rate information. With the Trigger Editor, you can specify 4GL logic that displays this information.

Your custom logic can simply display a message after a field.



5. **Once you enter your trigger code, press [ESC] to store your trigger.**

   Your trigger gets saved to a trigger (*.trg) file.

Creating Triggers by Hand

After a while, you might find it faster and more convenient to create triggers manually. That is to say, you might want to create trigger files directly using vi or some other UNIX text editor. Creating triggers by hand can be as simple as using the Form Painter as long as you follow the correct syntax.

All triggers follow the same general syntax:

```
input #
    trigger argument
        custom 4GL logic...
        ;
```

Where # indicates the input area number, *trigger* indicates the trigger command, and *argument* indicates any argument that the trigger accepts.

For example, the following after_input trigger displays a short message:

```
input 1
    after_input
        display "After input logic"
        sleep 2
        ;
```

Some triggers accept arguments. For example, this trigger accepts a field name (company) as a trigger command argument:

```
input 1
    after_field company
        display "After field logic"
        sleep 2
        ;
```

For a complete list of triggers, trigger descriptions, and syntax refer to the *Screen* Technical Reference.

Merging Triggers into Code

Once you create a trigger, you can merge it into your source code. To merge a trigger, however, you do not need to regenerate all your code. You can simply run either the make utility (fg.make) or the Featurizer (fglpp).

If you are using the Form Painter, simply select the Compile 4GL option under the Run pull-down menu. If you are working from the command line, type:

```
fg.make
```

or:

```
fglpp
```

Both commands initiate the Featurizer. The Featurizer reads your trigger (*.trg) file and places your code enhancements into the generated source code. When you run `fg.make`, the final source code (*.4gl) files contain your enhancement logic. The Featurizer saves your original source code in files with an *.org extension.

Section Summary

- Triggers are enhancements made directly to the source code generated from the *Screen* Code Generator. A trigger is an automatic way of placing code-level enhancements into the source code.
- You can create triggers using the Form Painter or by hand.
- Triggers let you create custom modification to logical points in your program flow.
- There are a number of triggers that can be merged into 4GL source code. Triggers are saved in trigger (*.trg) files, these files are given the same name as the form specification files they relate to. For example the `order.trg` trigger file relates to the `order.per` form specification file.
- The Featurizer reads *.trg files and merges the enhancements into the generated source (*.4gl) code files.

Exercise 9

Objective: To add a simple `before_input` trigger to the Customer Entry program.

Open cust.per in the Form Painter

1. **Move to `$HOME/labs/aw.4gm/i_cust.4gs` directory:**

   `cd $HOME/labs/aw.4gm/i_cust.4gs`

2. **Start the Form Painter.**

3. **Select Open from the File pull-down menu to load `cust`.**

Create a before_input Trigger

1. **Select Triggers >> from the Define pull-down menu.**

   The Choose a Trigger Class box appears.

2. **Select Input Area 1 from the Choose a Trigger Class box.**

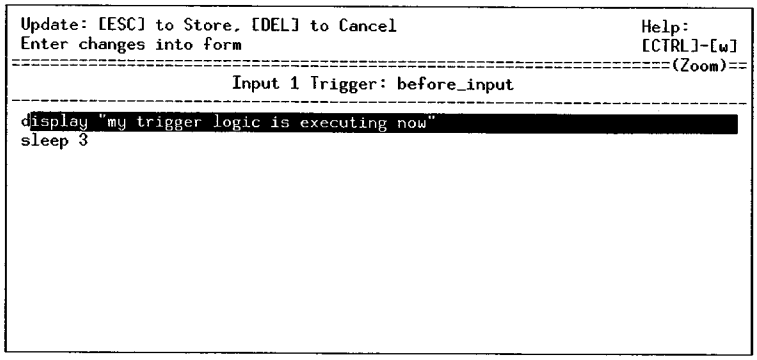
   The Choose a Trigger list box appears.

```
Choose:  [ESC] to Select.
[DEL] to Quit
================================
       Choose a Trigger
--------------------------------
define
static_define
before_input
after_input
before_field
after_field
after_change_in
        (16 items)
```

3. **Select before_input from the Choose a Trigger list box.**

   An editing window appears.

4. **Complete a "display" statement as follows:**



5. **Press [ESC] to save this `before_input` trigger.**

   The Choose a Trigger list box appears again.

6. **Press [DEL] to close the Choose a Trigger box.**

7. **Select Save Trg File from the File pull-down menu.**

Compile the Code

- **Select Compile 4GL from the Run pull-down menu.**

  The compilation utility calls the Featurizer (which is a code merging utility). The Featurizer merges your custom "display" logic into the generated source code.

Run the Customer Entry Program

- **Select Run 4GL Program from the Run pull-down menu.**

  The Customer Entry program starts.

Check the before_input Trigger

1. **Select Add from the ring menu.**

   Your custom "display" logic appears at the bottom of the screen.

2. **Finish adding the record.**

3. **Use Find to select a record and select update.**

   Again, your custom logic appears.

4. **Quit the program and the Form Painter.**

Examine header.4gl

1. **Use vi to open `header.4gl`.**

2. **Search for `before_input`.**

   Notice that your custom logic is inserted just before the input command:

```
#_before_input
    display "my trigger logic is executing now"
    sleep 3
#_end

#_input - Main input loop
```

   The #_ characters mark a trigger tag. In other words, these symbols define locations where triggers can be inserted.

3. **Using vi, search for other trigger tags.**

   This step familiarizes you with the types of triggers that are available. You will be adding custom logic to some of these locations at a later time.

4. **Exit `header.4gl`.**

Note When you make a change to a form (such as adding a field or field label), you must rebuild the program by running both the *Screen* Code Generator and `fg.make`. If you are only adding custom code via triggers, save the trigger file then run the `fg.make`. The *Screen* Code Generator is not required

---

10

Managing Screen to Table Flow

Main topics:

- Understanding Program Data Flow
- I/O Triggers
- Referencing Input Fields
- Common Global Variables
- Using the Scratch Variable

Understanding Program Data Flow

Before you start building input programs with *Screen*, it is helpful to understand how data is handled by programs created with the *Screen* Code Generator. In a general sense, input programs must perform two tasks:

1. Move data entered by the program user to the database.
2. Move data stored in the database to the screen.

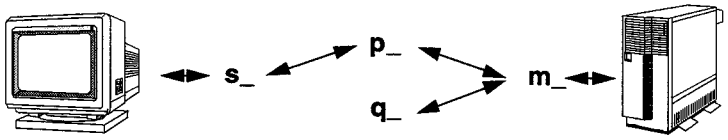
The Code Generator accomplishes both tasks by creating four records (`p_`, `m_`, `q_`, and `s_`) and two "prep" functions (`p_prep` and `m_prep`).

Data Flow Records

The p_ record: This record parallels the data elements defined on the screen. The p_ record only contains those fields displayed on your input program.

The m_ record: This record parallels information in the columns of a table. The m_ record contains variables with the same names as the columns in the database table.

Four records transfer data between the program user and the database.



The q_ record: This record contains all the columns not used by the input program but contained in the table.

The s_ record: This record contains values that get entered from or passed to the screen.

All records start with their various type (p_, m_, etc.). After the type, the record is named with the last six characters of the table name. For example `p_stomer` represents the p_ record for the `customer` table.

After the table name, the p_ record is built from all the input fields used by the input program. The following example shows a typical p_ record:

```
p_orders record   # Record like the order screen
    customer_num like orders.customer_num,
    fname like customer.fname,
    lname like customer.lname,
    company like customer.company,
    address1 like customer.address1,
    address2 like customer.address2,
    city like customer.city,
    state like customer.state,
    zipcode like customer.zipcode,
    phone like customer.phone,
    order_date like orders.order_date,
    po_num like orders.po_num,
    order_num like orders.order_num,
    ship_instruct like orders.ship_instruct,
    ship_weight like orders.ship_weight,
    ship_charge like orders.ship_charge,
    t_price money(10)
end record,
```

The m_ record does not use the column names like the p_ record. Instead the m_ record uses *. notation. For example, m_stomer.* represents a the m_ record for the customer table. The .* notation is used to allow the m_ record to accept data all at once. The following shows two example m_ records:

```
m_orders record like orders.*,  # Record like the header table
m_items record like items.*,  # Record like the detail table
```

The q_ record is defined like the p_ record, but it contains all the table columns not used by the program as input fields. For example:

```
q_orders record    # Parallel order record
    row_id integer, # SQL rowid
    backlog like orders.backlog,
    ship_date like orders.ship_date,
    paid_date like orders.paid_date,
    ship_method like orders.ship_method
    #_define_1
    #_end
end record,
```

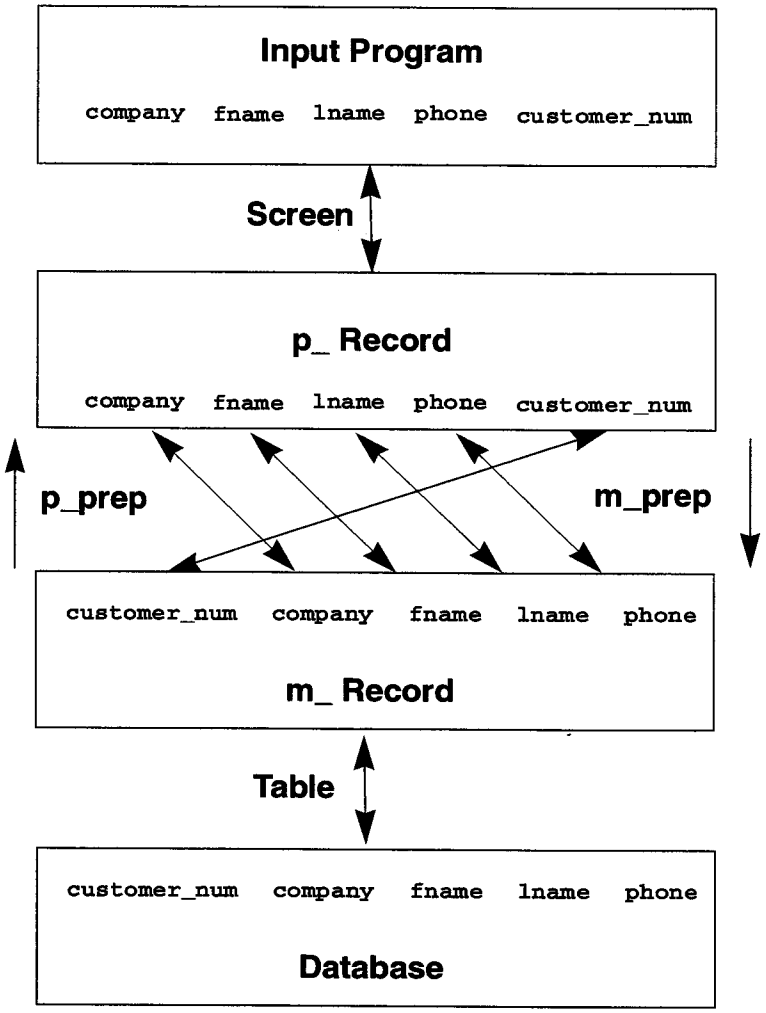
The s_ record gets defined in the Instruction section of the form specification file. It reflects the actual values displayed by the input program.

Data Flow Functions

The p_prep function: This function transfers data from the m_ record to the p_ record.

The m_prep function: This function transfers data from the p_ record to the m_ record.

Data flows between the input program and database by way of four records and two "prep" functions

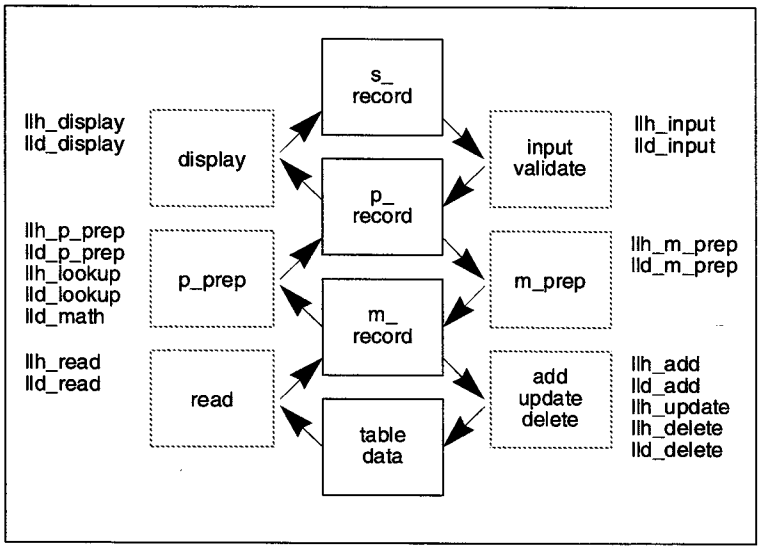


Lowlevel Functions Used by the Data Flow

Lowlevel functions control data flow (as illustrated below). The `header.4gl` and `detail.4gl` files contain the data flow functions.:

Data Flow	Header Functions	Detail Functions
From Database to Input Program		
database to m_ record	llh_read()	lld_read()
m_ record to p_ record	llh_p_prep()	lld_p_prep()
p_ record to s_ record	llh_display()	lld_display()
From Input Program to Database		
s_ record to p_ record	llh_input()	lld_input()
p_ record to m_ record	llh_m_prep()	lld_m_prep()
From m_ Record to Database		
create a new row	llh_add()	lld_add()
update a row	llh_update	none
delete a row	llh_delete	lld_delete()

Data input and display as it is associated with lowlevel functions



I/O Triggers

There are several useful triggers that are involved with the p_prep and m_prep functions. The following shows some of the triggers that insert code into the `llh*` and `lld*` functions shown on the previous page.

Trigger	Use
`on_disk_read`	Inserts code just after the SQL select loads the m_ record.
`on_disk_add`	Inserts code just after m_ record variables are inserted into the table.
`on_disk_update`	Inserts code just after a record is updated.
`on_disk_delete`	Inserts code just after a record is deleted.
`on_disk_record_prep`	Inserts code just after the m_ record is loaded with p_ record values.
`on_screen_record_prep`	Inserts code just after the p_ record is loaded with m_ record values.

Referencing Input Fields in Triggers

Frequently, you want to manipulate data in fields. You can do so with various triggers. When you reference an input field, though, you must always qualify it with its p_ record name. For example, illustrates an after_field trigger with an incorrect input field reference:

```
after_field company
    if company is null
    then
        error "You must fill in the company field"
    end if ;
```

Instead, you must qualify input fields with there p_ record name. This example shows a correctly referenced input field:

```
after_field company
    if p_stomer.company is null
    then
        error "You must fill in the company field"
    end if ;
```

In this case, `p_stomer.company` is the name of the p_ record that coincides with the company field.

The p_ record name is always found in globals.4gl. Field names are found in the globals.4gl file as well or in the form specification (*.per) file under the ATTRIBUTES section.

To reference table columns, you must qualify the column name with its m_ record.

Common Global Variables

The Code Generator always creates a common set of variables in your `globals.4gl` file. These variables, which can also be referenced in triggers, are very useful. You can find these variables under the Library communications section of your `globals.4gl` file.

```
###################################################################
# Library communication area 4.11.UD1
###################################################################
# Global variables in this section should not be changed.
# They are used to communicate to the screen library functions,
# and must be of the same type as defined in the library.
# Don't remove these comments.  The codegenerator keys on them.
#
progid        char(17),    # Program identification
scr_id        char(7),     # Current screen id
menu_item     char(10),    # Current menu item running
scr_funct     char(20),    # Current screen function being run
sql_filter    char(512),   # Filter portion of SQL statement
sql_order     char(100),   # Order portion of SQL statement
input_num     smallint,    # Current input section within screen
p_cur         smallint,    # Current input array element
s_cur         smallint,    # Current screen array element
scr_fld       char(40),    # Current screen field
nxt_fld       char(40),    # Programmatic next screen field
prev_data     char(80),    # Data before field entry
this_data     char(80),    # Data after field entry
data_changed  smallint,    # Has the field data changed?
hotkey        smallint,    # The hot key that has been pressed
scratch       char(2047)   # Scratchpad for scribbling on and
                           # communicating between functions
# End library communication area
###################################################################
```

Using the Scratch Variable

The `scratch` variable is used as a *scratch pad* for temporary data values. It is used throughout generated code.

Quite frequently, `scratch` is used for passing character type data between functions, such as SQL statements, messages, table names and column names.

Section Summary

- Input program data gets passed through the program code by way of records. In all, there are four records that the generator creates: the p_ record, m_ record, s_ record, and q_ record.
- The s_ record reflects the actual values displayed by the input program. The p_ record is formatted to parallel the input program fields. The q_ record contains table values not used by the input program. The m_ record parallels the columns in the database table.
- Two functions convert the m_ record to the p_ record and vice versa. These functions, known as `p_prep` and `m_prep`, control the mapping between the table columns and the program input fields.
- Data movement outside the program occurs all at once. Data values are accepted into a program from the screen *en masse* by the Informix input command. Values are displayed to the screen all at once by the Informix display command. The same holds true for inserts and most selects.
- Several lowlevel functions control the flow of data between the database, m_ record, p_ record, and input program. There are several I/O triggers that let you add custom logic to these functions.
- When you reference a column or input field in a trigger, you must preface it with its record type. For example, the `lname` field in the `customer` table, when called in a trigger, should be referenced as `p_stomer.lname`.
- There are a variety of useful variables that are always generated in the `globals.4gl` file. One of these variables is the scratch variable, which temporarily holds character values.

Exercise 10A

Objective: To reference a field on the screen and perform error-checking logic on that field.

You will reference a `p_` record variable and use an `after_field` trigger to perform validation. The error-checking logic that you create will require the user to supply a phone number when entering a new customer record in the Customer Entry program.

Add a Trigger

Your trigger will test for a null value in the Phone Number field.

1. **Start the Form Painter in your `i_cust.4gs` directory.**
2. **Open the main Customer Entry form (cust) in the Form Painter.**
3. **Move to the pull-down menus and select Triggers >> from the Define pull-down menu.**

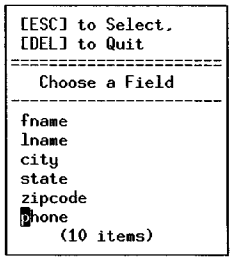
   The Choose a Trigger Class box appears.

4. **Since your Phone Number field is in the header section, select Input Area 1 from the Choose a Trigger Class box.**

   The Choose a Trigger list box appears. Because you want check a field for a null value, you want to evaluate the field once the user has moved past it. You want to use an `after_field` trigger.

5. **Select `after_field` from the Choose a Trigger list box.**

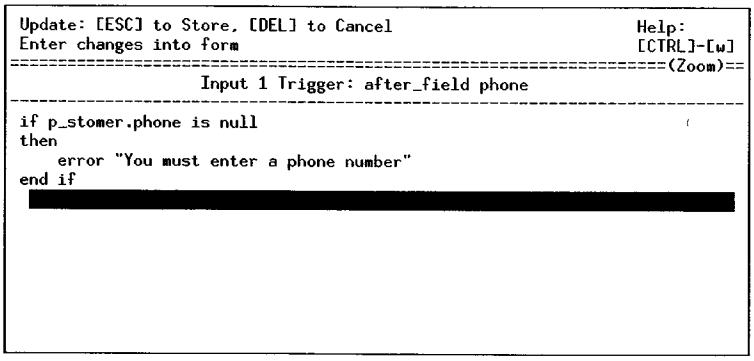
   The Choose a Field list box appears.



6. **Select phone from the Choose a Field list box.**

   The editing window appears.

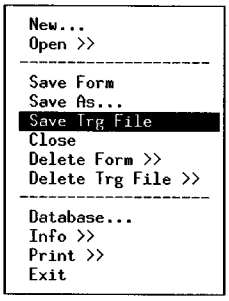
7. **In the editing window, add the following custom logic:**



Important Since you are referencing a field on the screen, the field name in your Informix logic must be qualified with its p_ record. If this is not done, a syntax error will occur.

8. **Press [ESC] to save your custom logic then [DEL] to close the Choose a Trigger list box.**

9. **Select Save Trg File from the File pull-down menu.**



This option writes your trigger logic into a trigger (*.trg) file.

Compile the Code

- **Select Compile 4GL from the Run pull-down menu.**

  The compilation utility calls the Featurizer. The Featurizer reads the trigger (*.trg) file and merges the `after_field` logic into the generated source code.

Run the Customer Entry Program

- **Select Run 4GL Program from the Run pull-down menu.**

  The Customer Entry program starts.

Test the after_field Trigger

1. **Select Add from the ring menu.**

   The `before_input` logic that you wrote in Exercise 9 appears.

2. **Enter data into the fields preceeding the Phone Number field.**

3. **Leave the Phone Number field blank and press [ENTER].**

   Your error message appears at the bottom of the screen and your cursor moves to the Credit Code field.

   ```
   You must enter a phone number
   ```

   This result is not entirely desirable. The error message works great, but you also must control the cursor movement. As it stands, you can save a record without entering a phone number.

4. **Press [ESC] to save this record and press Quit to return to the Form Painter.**

Modify the after_field Trigger

You can use the `nxt_fld` global variable in your trigger to control the condition on which the cursor can move to the next field.

1. **Return to the trigger editing window:**

Select	From
Triggers >>	The Define pull-down menu.
Input Area 1	The Choose a Trigger Class box.
after_field	The Choose a Trigger list box.
phone	The Choose a Field list box.

2. **Modify your trigger code to look as follows:**

```
Update: [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
=================================================================(Zoom)==
                    Input 1 Trigger: after_field phone
-----------------------------------------------------------------------------
if p_stomer.phone is null
then
    error "You must enter a phone number"
    let nxt_fld = "phone"
end if
```

3. **Press [ESC] to save your custom logic then [DEL] to close the Choose a Trigger list box.**

4. **Select Save Trg File from the File pull-down menu.**

Compile and Run

1. **Select Compile 4GL from the Run pull-down menu.**

2. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.

Test the after_field Trigger

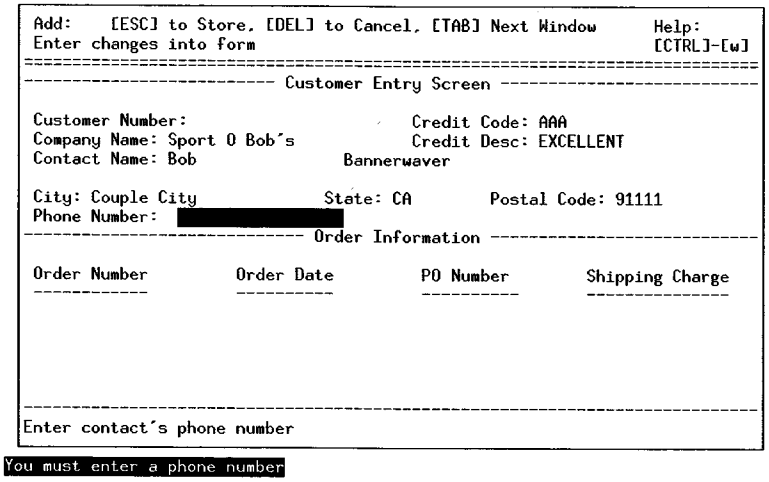
1. **Select Add from the ring menu.**

   The `before_input` logic that you wrote in Exercise 9 appears.

2. **Enter data into the fields proceeding the Phone Number field.**

3. **Leave the Phone Number field blank and press [ENTER].**

   This time the error message appears and your cursor remains trapped in the Phone Number field until you add a value.



4. **Add a phone number, press [ESC], and then Quit to return to the Form Painter.**

Remove a Trigger

By now you're tired of seeing the `before_input` logic you wrote in Exercise 9. You can remove this logic as simply as you added it.

1. **Move to the trigger editing window:**

Select	From
Triggers >>	The Define pull-down menu.
Input Area 1	The Choose a Trigger Class box.
before_input	The Choose a Trigger list box.

2. **Delete both lines of the `before_input` trigger.**

   You can delete a line quickly by pressing [CTRL]-[d].

3. **Press [ESC] to save your deletion then [DEL] to close the Choose a Trigger list box.**

4. **Select Save Trg File from the File pull-down menu.**

   The `before_input` logic is removed.

Note The File pull-down menu also has a Delete Trg File >> option. In this case you do not want to delete a trigger file because both your `after_field` and `before_input` triggers were in the same file. Use the Delete Trg File >> option when you want to remove all the triggers in that file.

Compile and Run

1. **Select Compile 4GL from the Run pull-down menu.**

2. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.

3. **Press Add to Verify that the `before_input` logic has been removed.**

4. **Once you have proven this to yourself, remain in the Customer Entry program and continue to Exercise 10B.**

Exercise 10B

Objective: To replace the Informix error statement with FourGen's `fg_err` function. This function lets you write custom error messages.

Test the after_field Trigger

1. **From the Customer Entry program, select Add.**
2. **Enter an invalid value in the Credit Code field (TTT).**

   An error message appears informing you that the value is not in the list of valid data. This message also include the ability to see more information about the error.

```
Error:  Value Is Not in the List of Valid Data.
Continue: [ENTER].  View error information: [Y].
```

3. **Press [Y] to see additional error information.**

   An error window appears.

```
Action: View Update Status Log Quit
Scroll through the error text
=============================================================
 Error:   Value Is Not in the List of Valid Data.
-------------------------------------------------------------
 This error occurs when:
   The value that was entered does not match any known value
    in the associated file.


-------------------------------------------------------------
 Possible solutions include:
   Change the value & try again.  If there is a zoom function
    attached to this field, you may scan through the file to
    find the correct lookup value by pressing [CTRL]-[z].
```

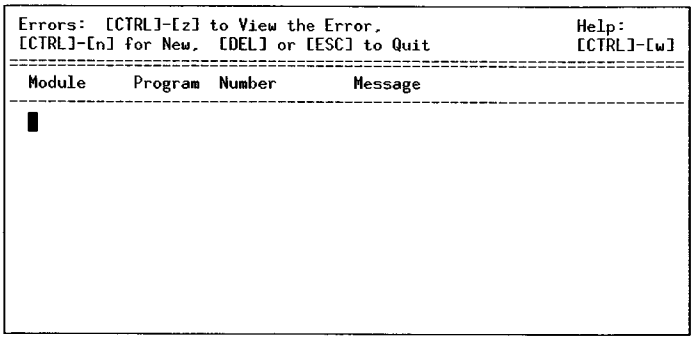
In this exercise, you will call a similar error window when the user leaves the Phone Number field empty (null).

Create Error Text

1. **Return to the Customer Entry ring menu.**

2. **Press [CTRL]-[g] to open the Navigate pop-up menu and select Edit Error Text.**

   An error text editing window appears.

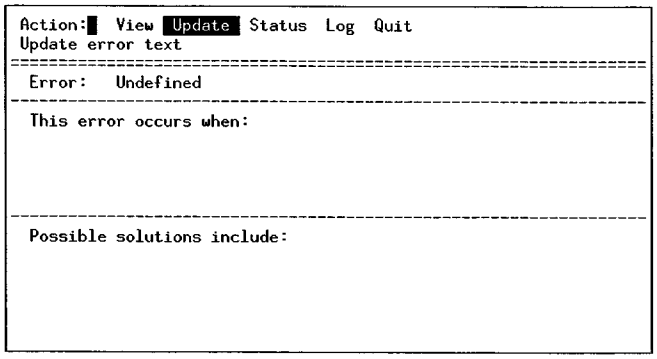


3. **Press [CTRL]-[n] to create a new error message.**

   A prompt appears requesting you to enter a new error number for this module/program.

4. **Enter 20 as the number.**

   A Problem/Solution window appears.



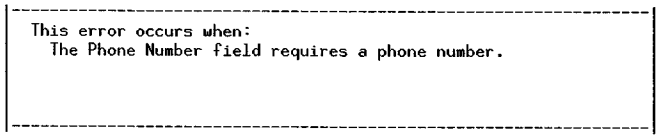
5. **Select Update to enter a new error message.**

   The cursor moves to the Error field.

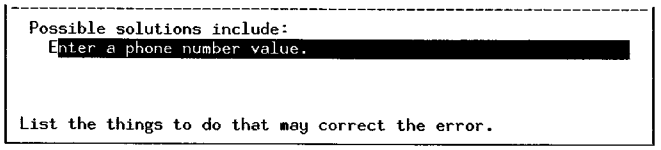
6. **Enter "This field requires a value," press [ENTER] then [TAB].**

   The cursor moves to the "This error occurs when" field and a default line appears. Press [CTRL]-[d] to delete the default line.

7. **Add the following message:**



8. **Press [ENTER] then [TAB] to move to the "Possible solutions include" field and add the following message:**



9. **When complete, press [ENTER], press [ESC] to save your error message, then Quit to return to the Customer Entry program.**

10. **Finally, select Quit to return to the Form Painter.**

Add a Call to fg_err in Your after_field Trigger

1. **Return to the trigger editing window:**

Select	From
Triggers >>	The Define pull-down menu.
Input Area 1	The Choose a Trigger Class box.
after_field	The Choose a Trigger list box.
phone	The Choose a Field list box.

2. **Replace the error line with a call to `fg_err`:**

```
Update: [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
=================================================================(Zoom)==
                    Input 1 Trigger: after_field phone
-----------------------------------------------------------------------------
if p_stomer.phone is null
then
    call fg_err(20)
    let nxt_fld = "phone"
end if
```

3. **Return to the pull-down menus and select Save Trg File >> from the File pull-down.**

Compile, Run, and Test

1. **Select Compile 4GL from the Run pull-down menu.**
2. **Select Run 4GL Program from the Run pull-down menu.**

   The Customer Entry program starts.
3. **Leave the Phone Number field empty to see what happens.**
4. **Remain in the Customer Entry program and continue to Exercise 10C.**

Exercise 10C

Objective: To require input in the Credit Code field.

You will build an `after_input` trigger that requires the user to enter a value in this field before the record can be saved.

Examine the Credit Code Field

1. **From the Customer Entry program, select Add.**
2. **Press [ENTER] to move past the Credit Code field.**
3. **Complete the record and press [ESC].**

   Notice how the program accepts this record without a value in the Credit Code field.

4. **Quit from the Customer Entry program and the Form Painter.**

Examine the cust.trg File

1. **Use vi to open the `cust.trg` file.**

```
##############################################################################
# Copyright (C)
# 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%
##############################################################################
# Screen Generator version: 4.11.UE1

input 1
    after_field phone
        if p_stomer.phone is null
        then
            call fg_err(20)
            let nxt_fld = "phone"
        end if;
```

To this file, you will add your `after_input` trigger. It is important to note that you can create triggers by hand using vi. You do not need to build them using the Form Painter, although the Form Painter makes it easier.

2. **Below the last line (`end if;`) add the following custom logic:**

```
after_input
    if p_stomer.credit_code is null
    then
        error "You must fill in the Credit Code field"
        let nxt_fld = "credit_code"
    end if;
```

3. **Save and exit `cust.trg`.**

Merge Your New Trigger Logic

- **At the UNIX prompt, run `fg.make`:**

  `fg.make`

  Remember that `fg.make` (which is analogous to the Compile 4GL Code option under the Form Painter's Run pull-down menu) calls the Featurizer. The Featurizer is the utility that merges trigger (*.trg) files into 4GL source code files.

Run the Customer Entry Program

- **Run the Customer Entry program using one of the following commands:**

  For RDS users:

  `fglgo i_cust.4gi`

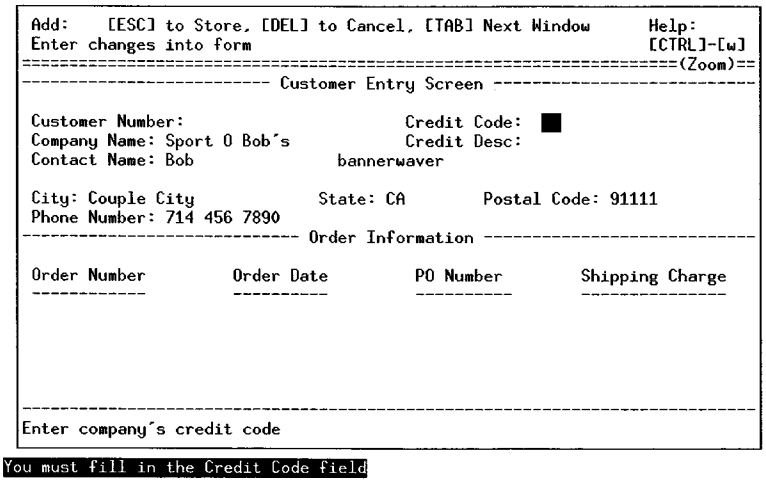
  The Customer Entry program starts.

Test Your after_input Logic

1. **Select Add from the ring menu.**

2. **Fill in all the fields except the Credit Code field and press [ESC].**

   Your error message appears and the cursor moves to the Credit Code field:



   You cannot save this record until you enter a value in the Credit Code field.

3. **Enter a Credit Code value, save this record, and press Quit to return to the UNIX command line.**

11

Screen Handling and Add-on Headers

Main topics:

- Using Different Screen Types
- The socketManager Function
- Linking in Add-On Screens

Using Different Screen Types

In chapter two, you learned about the different screen types you can build using the Form Painter. These screen types are classified into three groups:

1. Main Screens
2. Secondary Screens
3. Auxiliary Screens

Main Screens

A main screen constitutes the main part of your input program. There are two main screen types: header and header/detail screens.

header: This is a flat type. Header screens contain one input area and one main table.

header/detail: This is a flat type (header) with another scrolling (detail) section joined to the header. Header/detail screens are suited for order forms where there is one occurrence for customer information and multiple line items for merchandise.

Secondary Screens

Secondary screens are not used as stand-alone data-entry screens. Instead, they are called from the main screen. There are four secondary screen types: add-ons, extension, query, and view.

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.

add-on detail: This is a scrolling detail-only screen. This screen can be called from any other screen to display detail information. This screen type generates disk read and write functions.

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.

query: this screen is used for building an SQL query. It can replace the `mlh_construct` function.

view-detail: This is a detail-only screen that allows you to view data but not alter it.

view-header: This is a flat screen used to view header information.

Auxiliary Screens

Auxiliary screens are unlike any other screen type. These types are used in conjunction with the main screen and are basically used to locate and select information.

browse: This is a scrolling type screen. Its main table is the same as the header section main table. A browse screen enables you to view one row of the header table per line rather than one row per screen. Only one browse screen can be used per program.

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").

Linking Different Screen Types to the Main Screen

You can divide the input program creation process into two main tasks: painting the form images and linking screens together. This chapter shows you how to create and link add-on header screens.

In an earlier chapter, you learned how to link in zoom screens, and in later chapters you will learn how to link in other secondary screens.

Linking in an add-on header screen requires you to create a special trigger file. You call the `socketManager` function from within this file.

The socketManager Function

The `socketManager` function controls which code block or flow different screen types use. For every screen type, there exists default library code that is processed when that screen type gets initiated. When you link secondary screens to your main screen, you must use the `socketManager` function to call the library code associated with your secondary screen type.

The `socketManager` function syntax looks as follows:

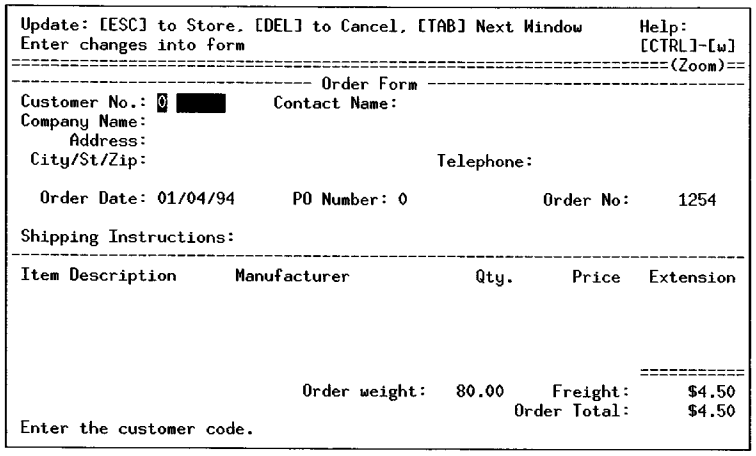
```
socketManager("screen_name", "screen_type", flow")
```

screen_name	This argument represents the form specification (*.per) file less the .per extension.
screen_type	This argument represents the screen type. Valid screen types include: add-on header, add-on detail, extension, query, view header, and view detail.
flow	Flow indicates a default block of library code associated with each screen types. In most cases, the flow is `default`. Extension screen types, however, require you to specify between one of three screen types: `flat_ext`, `deep_ext`, and `view`.

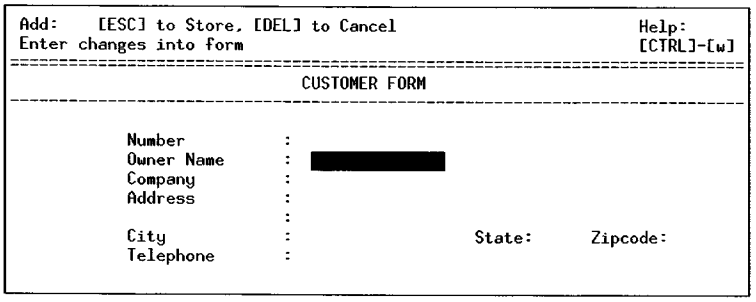
Designing Add-On Header Screens

Add-on header screens provide input fields to an additional table. Many times, you may want users to add data to this table during the data-entry process. While inputting orders a user might come across an order from a new customer. When the Customer No. field is assigned a zero, an add-on header screen appears, and the program user can enter information about the new customer before entering that customer's order.

A value of zero in the Customer No. field triggers an add-on header screen.



The program user can quickly enter information about a new customer, in an add-on header screen, before concluding order entry.



Building Add-On Header Screens

To build an add-on header, use the Form Painter to create the form image (select add-on header as the screen type). After you define the form image, save it to a form specification (*.per) file.

Linking in Add-On Header Screens

To link in your add-on header, you must create a trigger file that contains both the `switchbox_items` trigger and one or more initiating event triggers, such as an `after_field` trigger. For more on trigger files, see "Creating Triggers" on page 9-4.

You can use either the Form Painter or a text editor to create this trigger file. For it to work correctly, you must specify four pieces of information:

1. The name of the add-on header file, less the .per extension.
2. The trigger or event that initiates the add-on header screen. For example, the add-on header discussed on the previous page was initiated when the Customer No. field contained a value of zero.
3. The condition in which the add-on header screen is called. You specify condition settings with the `fgStack_push` function. All add-on header screens require you to set three attributes with the `fgStack_push` function: mode, filter, and order by.
4. The `socketManager` function.

In addition, your trigger file should be named after the main screen from which the add-on header gets called. For example, if the main screen is defined in the `order.per` specification file, the trigger file where you link your add-on header should be named `order.trg`.

The following code illustrates a `switchbox_items` trigger and an `after_field` trigger. Together these triggers specify all the information necessary to link in the `cust.per` add-on header screen.

switchbox_items trigger

after_field trigger

three calls to fgStack_push

a call to socketManager

```
default

switchbox_items
      cust S_cust;

input 1

   after_field customer_num
      if p_orders.customer_num=0
      then
          call fgStack_push("A")
          call fgStack_push("")
          call fgStack_push("")
          call socketManager("cust", "add-on header", "default")
      end if;
```

Using the example, you can see where each piece of information necessary to link in the add-on header screen gets supplied.

The `default` section contains the `switchbox_items` trigger. This trigger requires two arguments: the add-on header form specification file name (less the *.per extension) and the screen function. (The screen function name is always an S_ followed by the form specification file name.)

```
default

   switchbox_items
      cust S_cust;
```

The `input 1` section contains the trigger or event that initiates the add-on header screen. In the example, an `after_field` trigger initiates the add-on header screen.

```
input 1

   after_field customer_num
```

In addition, the `input 1` section contains the `fgStack_push` function, which sets add-on header conditions. For add-on header screens, you need to call this function three times. Even if you do not want to set some of these conditions, you still must pass this function three times passing null values for the conditions you do not want to set.

The first call indicates the mode that the add-on header screen starts in. An `A` indicates add mode. You can also specify a `U` for update mode.

```
call fgStack_push("A")
```

The second call indicates the selection filter. If you are opening your add-on header in update mode, you can pass it a filter indicating which records you want updated.

```
call fgStack_push("")
```

The last call relates to both update mode and the filter you specify. It constitutes an order by clause. If your filter selects multiple records, you can order those records by the criteria you specify in the third `fgStack_push` function call.

```
call fgStack_push("")
```

Finally, this section calls the `socketManager` function, which designates the correct flow for your add-on header screen.

```
call socketManager("cust", "add-on header", "default")
```

Section Summary

- You build input programs based on many different screen types. Each type has its own function.
- In all there are ten screen types. These ten types can be classified into three groups: main, secondary, and auxiliary.
- When you build input programs you must first create the form images and then link these images together using the `socketManager` function.
- The `socketManager` function controls which code block or flow different screen types use. For every screen type, there exists default library code that is processed when that screen type gets initiated. When you link secondary screens to your main screen, you must use the `socketManager` function to call the library code associated with your secondary screen type.
- Add-on header screens provide input fields to an additional table. Many times, you may want users to add data to this table during the data-entry process.
- To build an add-on header, use the Form Painter to create the form image (select add-on header as the screen type). After you define the form image, save it to a form specification (*.per) file.
- To link in your add-on header, you must create a trigger file that contains both the `switchbox_items` trigger and one or more initiating event triggers, such as an `after_field` trigger.

Exercise 11A

Objective: To become familiar with add-on header screens.

Add-on header screens provide additional data-entry screens that can be incorporated into your input programs. These screens write to tables other than the header or detail table.

Recall that in Exercise 3, you created the Credit Entry program. You later built a hot key to initiate this program from within the Customer Entry program. Add-on header screens provide much the same functionality, but they are further integrated into your base program.

Run scr_demo 5

The screen demonstration five program shows a good example of an add-on header screen.

1. **At the UNIX prompt type:**

   ```
   scr_demo 5
   ```

2. **From the Screen Demo prompt compile, generate, and run:**

   ```
   fg.screen -o0 -y
   ```

   ```
   fg.make
   ```

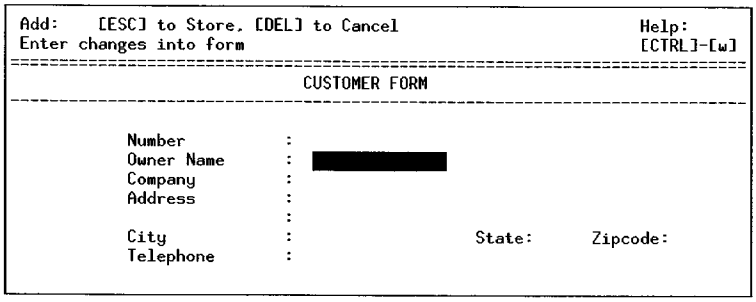
   ```
   fglgo screen5.4gi
   ```

   Screen demo 5 starts.

Add a Customer

1. **Select Add from the ring menu and enter 0 into the Customer Number field.**

   An add-on header screen appears, which looks similar to your Customer Entry program.



   This screen lets you add another customer record to the `customer` table.

2. **Fill in the Customer Form and press [ESC].**

   You've just added a new customer on the fly. Notice how back on the Order Form, the new customer number is returned and placed in the Customer Number field.

3. **Complete the Order Form and press [ESC] to save it.**

Add a Navigation Event

Add-on headers can also be used to update customer information in addition to creating new customer records.

1. **Use Find to select the record you just added.**
2. **Select Update to update this record.**
3. **Press [CTRL]-[g] to open the Navigate pop-up menu.**
4. **Select Add a navigation action.**

5. **Complete the Navigate Commands window as follows:**

In This Field	Type This
Action Code	update_cust
Description	Update a Customer

6. **Leave the other fields as they are and press [ESC].**

7. **Select the Update a Customer event from the Navigate menu.**

   The Customer Form add-on header screen reappears.

```
Update: [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
================================================================================
                              CUSTOMER FORM
--------------------------------------------------------------------------------

          Number          :    3516
          Owner Name      : Joe            Morgan
          Company         : Big Red Sports
          Address         : 123 St James AVE
                          :
          City            : Redbluff          State: CA   Zipcode: 90034
          Telephone       : 712-543-4567
```

8. **Change this customer's address and press [ESC].**

   Notice how the address is updated on the Order Form screen.

9. **Press [ESC] to save this change and Quit out of the Order Form program.**

10. **Exit screen demonstration five (type exit at the Screen Demo prompt) and return to your `i_cust.4gs` directory:**

    ```
    cd $HOME/labs/aw.4gm/i_cust.4gs
    ```

Exercise 11B

Objective: To create and use your own add-on header screen.

This add-on header will let users enter sales representatives to a new table from within the Customer Entry program.

To complete this exercise, you must perform the following major steps:

1. Add a column named `sales_code` to the `customer` table.
2. Add a Sales Code field to your Customer Entry program.
3. Create a new table called `salesrep`.
4. Create an add-on header screen based on the `salesrep` table.
5. Incorporate this screen into your Customer Entry program.

Add a Column

If you haven't done so already, move to your `i_cust.4gs` directory.

1. **Start the Form Painter and select Database from the File menu.**

   The Database option, as you recall, lets you change the structure of your database. You can add, delete, and alter the columns in a table.

2. **Find the `customer` table and add the `sales_code` column.**

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

  city               City               char(15)
  state              State              char(2)
  zipcode            Zip Code           char(5)
  phone              Phone Number       char(18)
  credit_code        Credit Code        char(3)
  sales_code         Sales Person Code  char(2)

Enter the data type for this column.
```

3. **Save this addition and press Quit to return to the Form Painter.**

Add a Field

1. **Back in the Form Painter, open your `cust` form file.**

2. **Add the Sales Code field to your Customer Entry form.**

   Probably the best location for this field is just above the Order Information detail section. Use Mark, Cut, and Paste to return the Phone Number field to its original location (below the Contact Name field). Then add the Sales Code field. Define this field using the following settings:

Table Name:	customer
Column Name:	sales_code
Input Area:	1
Entry ?:	Y
Message:	Enter sales code

   When you're finished, your form should look as follows:

```
Form Editor:  [ESC] or [DEL] Command Line                  [CTRL]-[w] Help
Press [CTRL]-[z] to update definition for field "sales_code"
=======(stores)==================(cust/1)========(Zoom)==(insert)==(8,13)===
------------------------- Customer Entry Screen ---------------------------

Customer Number:[           ]          Credit Code:[   ]
Company Name:[                    ]    Credit Desc:[          ]
Contact Name:[               ] [              ]
Phone Number:[                  ]
City:[             ]       State:[  ]       Postal Code:[     ]
Sales Code:[  ]
---------------------------- Order Information ------------------------------

Order Number         Order Date         PO Number          Shipping Charge
------------         ----------         ----------         ---------------
[           ]        [          ]       [          ]       [          ]
[           ]        [          ]       [          ]       [          ]
[           ]        [          ]       [          ]       [          ]
[           ]        [          ]       [          ]       [          ]

--------------------------------------------------------------------------------
Enter sales code
```

3. **Select Save Form from the File pull-down to save this change.**

Function of an Add-On Screen

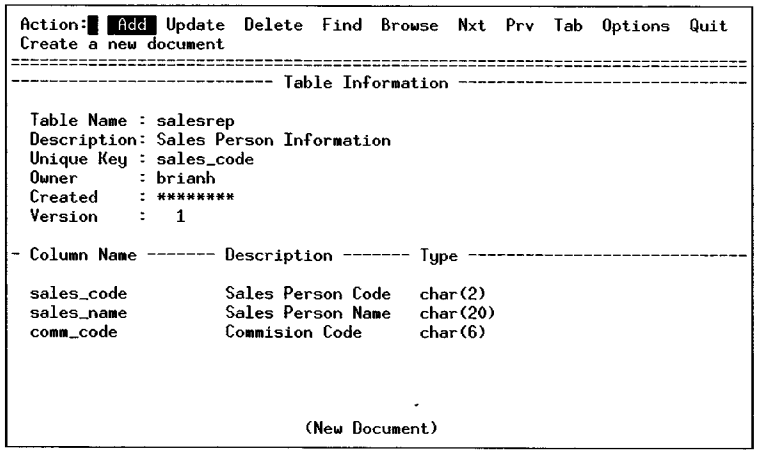
At this point, you could rebuild your Customer Entry program and start entering a sales person code for each customer. But this would simply be meaningless data; you could enter any characters into this field, none of which would stand for anything useful.

A better approach is to build an add-on screen based on a separate table. This table can contain information that is relevant to the sales code. You could add informative columns to this table, such as the sales person's name and rate of commission.

Add a New Table

Once again select Database from the File pull-down menu.

1. **Select Add from the ring menu and create the following entry:**



2. **Press [ESC] to save this table, but remain in the Table Information window.**

Note You may receive a Warning message about the Unique Key field. If so, simply press OK to continue.

Use AutoForm

Once `salesrep` is built, you can use the AutoForm option to build a default data-entry screen based on `salesrep`.

1. **Select the Options command and then choose AutoForm.**

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

```
Options: AutoForm Quit
Generate a default form into the clipboard
================================================================================
```

   A default entry screen is built and placed on the Clipboard in the Form Painter.

2. **Select Quit from the ring menu to return to the Form Painter.**

Create a New Add-On Header Form

1. **Select New from the Form Painter's File pull-down menu.**
2. **Name the new form "`reps`."**
3. **Select add-on header as the screen type.**
4. **Place the following title on the top line of the form:**

   ---------- Sales Person Add-On Screen ----------

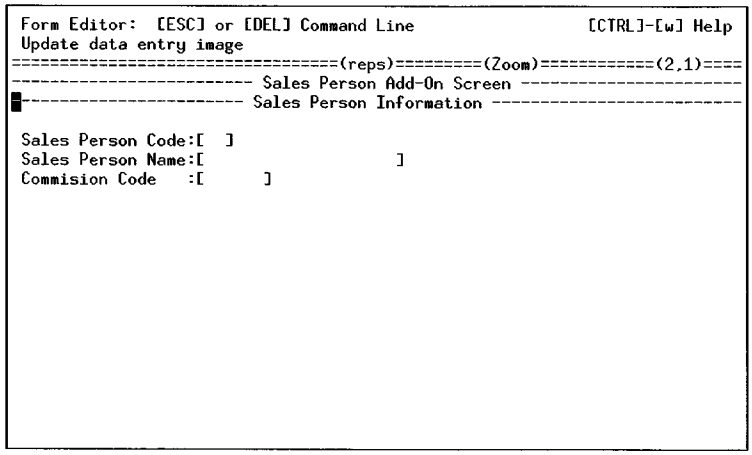
Paste in the AutoForm

Now add the default AutoForm image.

1. **After you add the title line, press [CTRL]-[p] to add the Auto-Form image.**

   A form built from the `salesrep` table appears. You can use the arrow keys to position in on your screen.

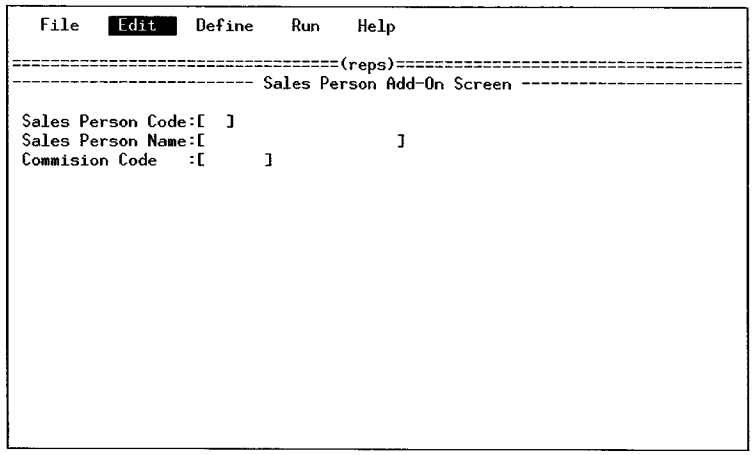
2. **"Tack" the image down by pressing [ESC].**



As you can see the AutoForm image also contains a title line. You can delete this extra title line with the [F2] key.

3. **Place your cursor on the first character of the extra title line and press [F2].**

When complete, your `reps` form should look as follows:



Generate Code

Once you save your `reps` add-on form, you can generate code for it.

- **Select Generate 4GL from the Run pull-down menu.**

  At this point, you do not have to compile it.

  Instead, use the Form Painter to reopen your `cust` form.

Incorporate Your reps Add-On

After `reps` is built, you need to attach it to your to your Customer Entry program. You attach add-on screens using triggers.

For your Customer Entry program, you will build custom logic in an `after_field` trigger. This trigger will evaluate your Sales Code field. When this field contains an `xx` value, it will call your add-on.

1. **In your `cust` form (i.e., your Customer Entry screen), build the following `after_field` trigger:**

```
Update: [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
================================================================(Zoom)==
                  Input 1 Trigger: after_field sales_code
----------------------------------------------------------------------------
if p_stomer.sales_code = "xx"
then
    call fgStack_push("A")
    call fgStack_push("")
    call fgStack_push("")
    call socketManager("reps", "add-on header", "default")
end if
```

2. **After you create this trigger, select Save Trg File from the File pull down menu.**

3. **Once your trigger is saved, select Compile 4GL Code from the Run pull-down menu.**

   Don't try to run your program yet, it won't work until you complete the next exercise.

12

Working with Switchboxes

Main topics:

- Switchbox Overview
- How do Screens Get Into Switchbox
- Zooms and Switchboxes

Switchbox Overview

generated code features Switchbox logic. In general terms, a Switchbox manages flow control between library functions and local functions. defines two types of Switchboxes:

1. Screen-Level Switchbox
2. Function-Level Switchbox

Screen-Level Switchbox

The screen level switchbox resides in `main.4gl` and passes control to the appropriate program screen. Screen-level switchbox is controlled by the `switchbox` function. This function reads the value in the global scr_id variable. The scr_id variable can contain any valid form specification file in your program less the .per extension. For example, your input program might contain the following form specification files:

Filename	Screen Type	scr_id Value
browse.per	browse	browse
cust.per	add-on header	cust
cust_zm.per	zoom	cust_zm
order.per	header/detail	default
stockzm.per	zoom	stockzm

As you can see, your header/detail screen (or main screen) receives default as its `scr_id` value. If your program contained a header screen instead of a header/detail screen, the header screen would receive default as its `scr_id`.

Depending on the value in `scr_id`, flow is passed to the function level Switchbox.

Function-Level Switchbox

The function-level switchbox determines what happens next. For each form specification file in your program (i.e., for each screen used by your program) a function-level switchbox is generated. The function-level switchbox reads the value in the scr_funct variable. Once this value is read, the function level switchbox uses a large case statement to determine the appropriate action.

When the *Screen* Code Generator creates each function-level switchbox, it names the swichbox after the form specification file or `scr_id` that it relates to. The only exception being header and header/detail form specification files. These files use the `lib_screen` function as their function-level switchbox.

For example, if the `scr_id` variable equals `cust_zm`, a `cust_zm` function is generated in the `cust_zm.4gl` file. This function contains all the possible actions that can take place from within the `cust_zm` screen.

The following code illustrates an example `cust_zm` switchbox function.

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

As you can see from the sample code, there are several logical points within a switchbox function. The extended `case` statement provides several code points that you can customize using triggers or block commands (see "Creating Triggers" on page 9-4 and "Block Commands" on page 18-3).

How Screens Get Into Switchbox

The screen level switchbox function, which actually uses the name `switchbox`, determines which program screen is active and selects the correct program flow based on the active screen. The `switchbox` function evaluates the value in scr_id to know which screen and thus which series of code to process. For this reason, it is important that you define the links between your main program screen and your secondary screens accurately. In chapter 11, you learned how to link an add-on header screen to a main screen using the `switchbox_items` trigger, an `after_field` trigger, the `fgStack_push` function, and the `socketManager` function. By using the `switchbox_items` trigger, you declared your add-on header screen to the `scr_id` variable. In essence, you made the `switchbox` function aware of your add-on header screen.

For the screen level `switchbox` function to work, you must make sure that all your secondary screens get linked in properly using the `switchbox_items` trigger.

Code to place zoom screens into the `switchbox` function gets generated automatically. When the *Screen* Code Generator reads a zoom attachment (i.e., the zoom= line in the form specification (*.per) file), it places not only the library function that invokes the zoom screen, but also the entry into the `switchbox` function. The *Screen* Code Generator adopts responsibility for placing all zoom support logic into code.

The main program screen (your header or header/detail) also gets placed in automatically when you run the *Screen* Code Generator.

The switchbox_items Trigger

You make screens known to the `switchbox` function with the `switchbox_items` trigger. The `switchbox_items` trigger uses the following syntax:

```
default
  switchbox_items
    screen_name screen_function_name
```

Here is an example of an add-on screen being placed into the `switchbox` function by the `switchbox_items` trigger:

```
default
  switchbox_items
    cust S_cust ;
```

The above code, placed in a trigger (*.trg) file results in the following line added to `switchbox` in main.4gl:

```
when scr_id = "cust" call S_cust()
```

If a request is passed to `switchbox` by a library function and the `switchbox` function does not know the screen to pass it to, then the following error message appears:

```
Screen not attached to program
```

Section Summary

- Numerous screens combine to constitute an input program. All programs have a main screen (called the "default" screen) which is either a header or header/detail type screen. Other screens such as zoom screens and add-on header screens are attached to the main screen.
- All screens that interact with an input program must be known to the `switchbox` function. The `switchbox` function constitutes the screen-level `switchbox`. There is also a function-level `switchbox`. Both types of `switchbox` functions exist in every input program.
- Library functions pass generic requests to local code via the two `switchbox` function levels. The first `switchbox` level (the screen level) uses the `switchbox` function. Its job is to receive the request from the library functions and determine which program screen to use. The `switchbox` function is generated in local code and placed in the main.4gl file.
- The second `switchbox` level (the function level) evaluates the screen-level request and passes control to the appropriate low-level function, which handles the request. The low-level function contains all the code to process the request. When complete, program control returns to the library function.
- The second-level `switchbox` contains functions with a variety of names. The `lib_screen` function is the second-level `switchbox` function for the main screen. This function handles requests including highlighting fields and recording values.
- Since the `switchbox` function passes requests based on the program screen, all screens interacting with the input program must be "known" to the `switchbox` function. In other words, all screens must have logic in `switchbox` so that when a request is passed to the `switchbox` function, it knows where to pass the request.
- You can use the `switchbox_items` trigger to make your screen known to the `switchbox` function. Thus, when requests to perform something to your screen are received by the `switchbox` function, it can direct control to the appropriate code.

Exercise 12

Objective: To create a `switchbox_items` trigger that "links" the Sales Person add-on screen to the Customer Entry screen.

Examine main.4gl

Had you tried to run your Customer Entry program at the end of Exercise 11, and attempted to access your new add-on screen, the following error message would have occurred:

```
Screen not attached to program
```

1. **Exit the Form Painter and use vi to open `main.4gl`.**

2. **Search for the `switchbox` function.**

```
##########################################################################
function switchbox(funct)
##########################################################################
# This is the switchbox function for version 4.11.UE1 screens.
# It is used to pass flow control to the appropriate screen function.
#
    #_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 = "cred_zm" call cred_zm()
      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()
```

This function contains a "flow control" `case` statement that is based on the `scr_id` variable. As you can see, your `reps` add-on header screen is not yet a part of this statement. Before your add-

on header screen works properly, you have to create a special trigger, known as the `switchbox_items` trigger. This trigger makes your add-on header screen known to `switchbox`.

Add the switchbox_items Trigger

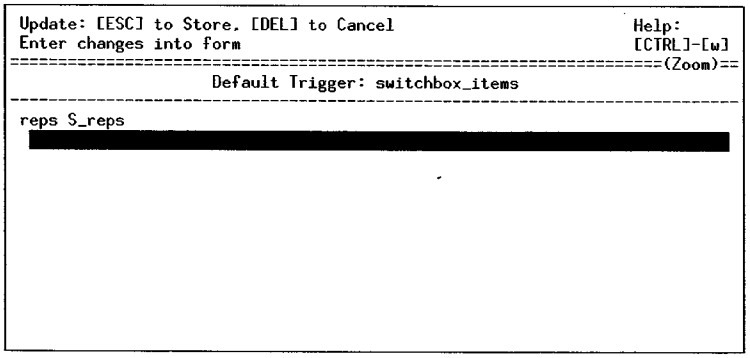
The `switchbox_items` trigger creates a "when" clause in the `swichbox` function. This trigger goes in the "defaults" section of the trigger file.

1. **Start the Form Painter and open your `cust` form.**
2. **Select Triggers >> from the Define pull-down menu.**

   The Choose a Trigger Class box appears.
3. **Select Default as the Trigger Class.**

   The Choose a Trigger list box appears.
4. **Select `switchbox_items` trigger.**

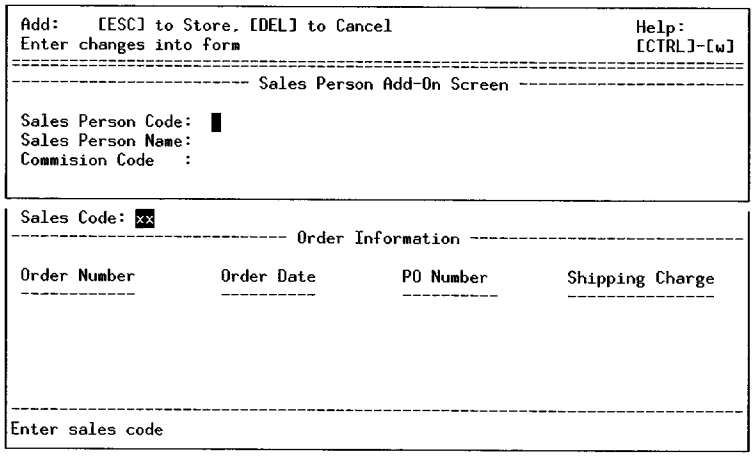
   The editing window appears.
5. **Add the following line then save your trigger (select Save Trg File then Save Form from the File pull-down menu).**



The first value (in this case `reps`) represents the name of your add-on header screen. The second value (`S_reps`) represents the name of the function that will control your screen.

6. Compile and run the Customer Entry program.

What happens when you type xx in the Sales Code field? You should see the Sales Person add-on screen.



13

Working with Program Events

Main topics:

- Program Event Overview
- Program Event and Hot Key Tables
- The on_event Trigger
- The at_eof Trigger

Program Event Overview

Program events are either internal or external actions that you can execute from within an input program. You can suspend your input program at any moment and run a program event.

You can add program events to your Navigation menu, which you activate with [CTRL]-[g]. You can also map program events to hot keys.

External and Internal Events

As mentioned above, program events are classified either as external events or internal events.

Events that contain UNIX operating system commands constitute external events.

Events that issue Informix commands are internal. Internal events can be further classified into local and global events.

Local and Global Events

A local event is an internal event that is executable only on one portion of the screen. An event that is "local to the header" can only be executed on the header portion of the screen. Whereas "local to the detail" specifies an event that only takes place on the detail portion of the screen.

A global event is an internal event that is executable from anywhere on the screen. A global event can be executed on the header or detail portions of the screen, from the ring menu, from a zoom screen, an add-on screen, etc.

Program Event and Hot Key Tables

All events and hot keys that you set up are kept in reference tables in the database.

Navigation Event Reference Table

Program events are kept in the Navigation Event Reference table, which goes by the name `stxactnr`.

`language`	Holds the language variable for the event, such as [ENG] for English.
`act_key`	Holds the event name. When you define events, you specify a value in an Action Code field. Whatever value you specify gets placed in this column.
`description`	Holds a description of your program event.
`os_command`	Holds the operating system command associated with your program event (for external events only).
`press_enter`	Holds a Y/N value. When your event completes, you can set a prompt to appear before returning to the input program. The "Press Enter to Continue" prompt gives you an opportunity to check error messages if an error occurred during event execution.

An internal event does not contain a value in the `os_command` column and it sets the `press_enter` column to N.

Navigation Event Detail Table

All the program events that you set up are also kept in the Navigation Event Detail table, which uses the name `stxnvgtd`. This table specifies what program and user the event is associated with.

act_key	Holds the event name. When you define events, you specify a value in an Action Code field. Whatever value you specify gets placed in this column.
line_no	Holds the line number value for the program event.
nav_module	Holds the module name for the event.
nav_prorgram	Holds the program name for the event.
nav_user	Holds the user name for the event. This value can be set to all or specify a single user.

Hot Key Definitions Reference Table

The Hot Keys Reference table assigns a unique number to most control keys and function keys. Control keys correspond with the order the letters appear in the alphabet. Function key number start with 101. This table has two fields: `key_code` and `key_desc`. The following list shows some default hot key entries:

key_code	**key_desc**
[2]	[[CTRL]-[b]
[5]	[CTRL]-[e]
[6]	[CTRL]-[f]
[101]	[F1]
[102]	[F2]
[103]	[F3]

Hot Key Definitions Detail Table

The `stxhotkd` table maps program events to control or function keys. It contains the following columns.

hot_key	Holds the numeric hot key value.
act_key	Holds the event name. When you define events, you specify a value in an Action Code field. Whatever value you specify gets placed in this column.
hot_module	Holds the module name for the hot key.
hot_program	Holds the program name for the hot key.
hot_user	Holds the user name for the hot key.

The on_event Trigger

The `on_event` trigger lets you place custom logic for an internal event into your program code. The `on_event` trigger uses the following trigger:

```
on_event event_name
  informix_instruction... ;
```

For example, the following `on_event` trigger displays a message when the `show_message` event takes place:

```
on_event show_message
  display "internal event logic"
  sleep 3 ;
```

The event name, which is `show_message` in the above example, correlates with the `act_key` value in the `stxactnr` table and the Action Code field in the navigation window.

You must decide what section of the trigger file your `on_event` trigger belongs. Depending on the section, the `on_event` trigger goes into a different source code file.

defaults	Puts the trigger code into a `global_events` function in main.4gl.
input 1	Relates to the header portion of your main screen. The trigger code is placed in header.4gl.
input 2	Relates to the detail portion of your main screen. The trigger code is placed in detail.4gl.

The at_eof Trigger

The `at_eof` trigger places whatever you put in it at the end of a file. It is commonly used for putting in functions that your write or library functions that you customize. The following `at_eof` trigger illustrates a custom function:

```
at_eof

#########################################################
function my_funct()
#########################################################

    display "this is my own personal function"
    sleep 3

end function
# my_funct
    ;
```

There are three common uses for the `at_eof` trigger:

1. Adding custom functions.
2. Modifying library functions.

   Library functions exist outside your program directory. They are shared by many programs. If you modify a library functions in the library, you change how it works throughout all your applications. It is much safer to alter library functions in your local directory using the `at_eof` trigger. Even though this creates two functions with the same name, the function in the local directory takes precedence.

3. Modifying a locally generated function.

   By placing a locally generated function into your trigger file you can modify it, but the original function still exists in source code. You must use a `do_not_generate` trigger to keep the original function from generating. For example if you alter the `mlh_clear` function in your trigger file, add the following `do_not_generate` trigger as well.

```
do_not_generate
      mlh_clear;
```

Section Summary

- Events have instructions attached to them. You can execute events at any time within a program. All events may be viewed, setup, and executed via the Navigation Menu, which you access by typing [CTRL]-[g].
- Any event can be mapped to a hot key. Once mapped to a hot key, the user can press the hot key to execute the event.
- External events have operating system instructions attached to them.
- Internal events have Informix instructions attached to them.
- Local events are internal events that can only be executed on one portion of the screen (either the header or detail).
- Global events are internal events that can be executed on any portion of the screen: header, detail, or ring menu.
- All events are set up as rows in two tables: the Navigation Event Reference table (`stxactnr`) and the Navigation Event Detail table (`stxnvgtd`).
- All hot key mappings are set up as rows in two tables: the Hot Key Definition Reference table (`stxkeysr`) and the Hot Key Definitions Detail table (`stxhotkd`).
- The `on_event` trigger is used for placing Informix instructions for a local event into generated code.
- The `at_eof` trigger is used to add code at the end of a file. You can modify library functions so that they behave a certain way just for the program you are running. You can even use the `at_eof` trigger to modify a function generated in local code, but if you do, you must use the `do_not_generate` trigger to prevent the original function from being created.

Exercise 13A

Objective: To add a simple internal event to your Customer Entry program. This event will display a message to the bottom of the screen.

Add a Navigation Event

1. **Start your Customer Entry program.**
2. **Press [CTRL]-[g] to display the Navigate menu.**
3. **Select Add a navigation action from the Navigate menu.**

   The Navigate Commands window appears.

```
Update: [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
================================================================================
                              Navigation Commands
--------------------------------------------------------------------------------

Action Code:
Description:
Operating system command:


Press ENTER upon return   ? N
Access from other programs? N
Allow access for others   ? N

--------------------------------------------------------------------------------
Enter a unique identification code.
```

4. **Create and save the following Navigate Commands entry:**

```
================================================================================
                              Navigation Commands
--------------------------------------------------------------------------------

Action Code:  display_test
Description:  Run Test Display
Operating system command:


Press ENTER upon return   ? N
Access from other programs? Y
Allow access for others   ? N

--------------------------------------------------------------------------------
```

Insert an on_event Trigger

Now you will build an `on_event` trigger to add logic that drives the event you just created. When your user selects the event, the custom logic is run.

1. **Quit the Customer Entry program and use vi to edit your `cust.trg` file.**

Note

Remember, there are two ways to build triggers. You can use the Form Painter, or you can build them manually using vi.

```
########################################################################
# Copyright (C)
# 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%
########################################################################
# Screen Generator version: 4.11.UE1

defaults
    switchbox_items
        reps S_reps;

input 1
    after_field phone
        if p_stomer.phone is null
        then
            call fg_err(20)
            let nxt_fld = "phone"
        end if;

    after_field sales_code
        if p_stomer.sales_code = "xx"
        then
            call fgStack_push("A")
            call fgStack_push("")
            call fgStack_push("")
            call socketManager("reps", "add-on header", "default")
        end if;

    after_input
        if p_stomer.credit_code is null
        then
            error "You must fill in the Credit Code field"
            let nxt_fld = "credit_code"
        end if;
```

As you can see, you have already defined several triggers.

2. **Use vi to add the following lines of code to the bottom of `cust.trg`.**

```
on_event display_test
    let scratch = "Hello World"
    call lib_message("scr_bottom")
    sleep 3;
```

3. **Save `cust.trg` and use `fg.make` to compile.**

4. **When complete, run Customer Entry.**

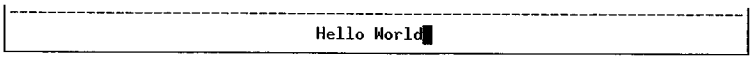
Add a Record

1. **Select Add from the ring menu.**

2. **While in the header portion of the screen, press [CTRL]-[g].**

3. **Select Run Test Display from the Navigate menu.**

   What happens? You should see the Hello World message on the bottom of your screen.



4. **Now move to the detail portion of the screen.**

5. **Press [CTRL]-[g] again and select Run Test Display.**

   What happens this time? You don't see a message appear because the `display_test` action is only defined for the header portion of the screen (i.e., Input Area 1). Your next step is to make this a global event, i.e., an event accessible from anywhere within your Customer Entry program.

Exercise 13B

Objective: To convert the `display_test` event into a global event.

Move the on_event Trigger

1. **Quit from your Customer Entry program and vi `cust.trg`.**
2. **Move the entire `on_event` trigger from the `input 1` section of the trigger file to the `defaults` section.**

```
defaults
    switchbox_items
        reps S_reps;

    on_event display_test
        let scratch = "Hello World"
        call lib_message("scr_bottom")
        sleep 3;
```

   As you can see, you already have a `switchbox_items` trigger in this section. Place your `on_event` trigger directly below it.

3. **Save the `cust.trg` file and compile the code (`fg.make`).**
4. **Run Customer Entry and test your on_event trigger from different portions of your screen.**

   Can you display the Hello World message from the ring menu? From the header section? From the detail section?

5. **When you are finished testing this trigger, quit from Customer Entry.**

Exercise 13C

Objective: To use the at_eof trigger to disable a ring menu option.

You will use the `at_eof` file trigger to add custom logic to the `ok_delete` function. This function is called when the user selects the Delete ring menu command.

Under normal conditions, the `ok_delete` function returns true. You are going to alter `ok_delete` so that it returns false.

Add an at_eof Trigger

1. **Use vi to open your cust.trg file.**
2. **In the defaults section add the following `at_eof` trigger.**

```
at_eof
    function ok_delete()
        let scratch = "You are unable to delete a record"
        call lib_message("scr_bottom")
        sleep 3
        return false
    end function;
```

3. **Save your trigger and compile the code.**
4. **Run Customer Entry.**
5. **Select a record and then try to delete it.**

   What do you see? Your message should appear at the bottom of the screen, and you should be unable to delete the record.

14

Creating Pop-Up Menus

Main topics:

- Pop-Up Menu Overview
- Assigning Pop-Up Menus to Program Events
- Initiating Secondary Screens from Pop-Up Menus

Pop-Up Menu Overview

Pop-up menus provide program users with a list of program options. You can think of a pop-up menu as a single column zoom screen. If you have used the Form Painter, you have already seen and used many pop-up menus.

To create a pop-up menu, you have to answer three questions:

1. What items go on the menu?
2. What happens when a user selects an item?
3. What action initiates the pop-up menu?

You can use the `textput()` and `textpick()` functions to answer the first two questions. The final question depends on how you want your program to operate.

Textput

This function loads the items that appear on a pop-up menu. It is called once for each item on the menu. The following example places three items on the pop-up menu:

```
call textput("Add a Contact")
call textput("Update a Contact")
call textput("Delete a Contact")
```

Textpick

This function displays the pop-up menu with all of the items on it.

```
call textpick("Select an Option")
   returning item
```

It also assigns each item a value according to the items position on the menu (i.e., the top item is assigned number one). When the user selects an item, the value corresponding to that item is returned (so if the user selects the top item, 1 is returned). In the above example, the numeric value gets placed into the `item` variable. You must declare a variable before you can use it in a returning statement. To declare a variable, you can use the `define` trigger. For example:

```
defaults
    define
        item smallint;
```

You can follow the `textpick()` function with a case statement that describes what should take place when an item is selected by the user. For example:

```
case
  when item = 1
    call add_contact()
  when item = 2
    call upd_contact()
  when item = 3
    call del_contact()
end case;
```

Initiating a Pop-Up Menu

There are a number of ways to initiate a pop-up menu. You can use a trigger, such as an `after_field` trigger or a program event that is assigned to a hot key. For example, the code to initiate this pop-up menu is placed in an `after_field` trigger:

```
input 1
    after_field customer_no

call textput("Add a Contact")
call textput("Update a Contact")
call textput("Delete a Contact")

call textpick("Select an Option")
   returning item

case
  when item = 1
    call add_contact()
  when item = 2
    call upd_contact()
  when item = 3
    call del_contact()
end case;
```

When the user presses [ENTER] from on the `customer_num` field, the pop-menu appears.

When a menu item is selected, the appropriate function is called. For example when a user selects "Update a Contact" from the menu, the The `upd_contact()` function is called.

Assigning Pop-Up Menus to Program Events

Sometimes you might want to make a pop-up menu available throughout the header or detail portion of an application. If this is the case, you can define a program event and initiate your pop-up menu with an `on_event` trigger.

For example, suppose you want to display the same pop-up window every time the user presses [F6] from within the header section.

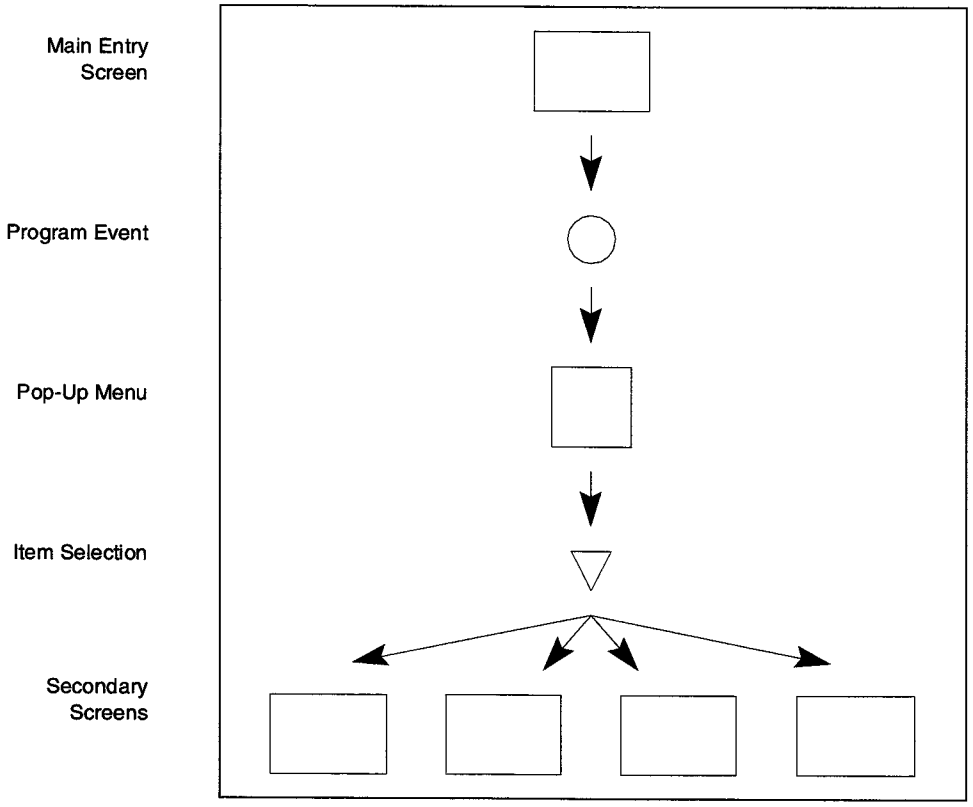
First define a program event called `popup` and map it to the [F6] key (see "Mapping Hot Keys to Navigation Events" on page 7-10).

Next, create an `on_event` trigger that initiates the pop-up menu when the [F6] key is pressed. Put this `on_event` trigger in `input 1` section of your trigger file. For example:

```
input 1
    on_event popup

    call textput("Add a Contact")
    call textput("Update a Contact")
    call textput("Delete a Contact")

    call textpick("Select an Option")
       returning item

    case
      when item = 1
        call add_contact()
      when item = 2
        call upd_contact()
      when item = 3
        call del_contact()
    end case;
```

Initiating Secondary Screens from Pop-Up Menus

Now that you can create pop-up menus and initiate them from any program event, you can turn your attention toward their functionality. Perhaps your program users need global access to a zoom, add-on header, or view detail screen. You can create pop-up menu items that open secondary screens.



Recall from chapter 11, you used the `switchbox_items` trigger, the `fgStack_push()`, and the `socketManager()` function to attach an add-on header screen to your main screen. To attach secondary screens to a pop-up menu, you follow much of the same process. But you should note that different secondary screens are attached in different ways. At this point, you've only looked at add-on headers and zooms (and zoom screens are attached automatically).

Suppose for now that you want to create a pop-up menu that contains two items. One item initiates a zoom screen and the other item initiates a view detail screen. To create such a menu, you must complete the following steps:

1. Use the Form Painter to paint the zoom and view detail images. Save both images as form specification (*.per) files (see "Creating a Form Image" on page 2-5).
2. Use the `switchbox_items` trigger to declare your zoom and view detail screens see the chapter.
3. Build an `on_event` trigger that initiates the pop-up menu.
4. Create the frame work for the pop-up menu.
5. Create a `case` statement that calls both screens.

Create the Zoom and View Detail Screens

Use the Form Painter to build the images for both these screens. For the zoom screen, don't worry about setting any zoom definition values: you don't want the *Screen* Code Generator to attach the zoom screen automatically. Instead, you must build your own attachment logic, which you place within the `on_event` trigger. Once you build a form image save the image to a form specification file (i.e., select the Save Form option from the File pull-down menu).

Build the switchbox_items Trigger

After you create the screens, declare them to the `switchbox_items` trigger. In this example, the screens are `shipzm` and `infodt`.

```
defaults
    switchbox_items
        shipzm shipzm
        infodt S_infodt;
```

Build an on_event Trigger

Create logic for an `on_event` trigger. You can use an `on_event` trigger similar to the one that you created in the previous section:

```
input 1
    on_event popup
    ...
```

Remember that the `popup` event was assigned to the [F6] key.

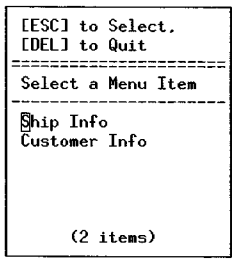
Add the textput and textpick Logic

After you create the `on_event` trigger, add your `textput` and `textpick` logic. For example:

```
call textput("Ship Info")
call textput("Customer Info")

call textpick("Select a Menu Item")
    returning item
```

These functions combine to create the following pop-up menu:

Pop-up menu created by the above code sample.



Create a case Statement that Calls Both Screens

Now you can call the screens by using a case statement following the `textpick()` function. Different screen types require different syntax. Recall that an add-on header required you to use three `fgStack_push()` functions and the `socketManager()` function. Zoom and view detail screens have a different attachment syntax as well.

For zoom screens, you must call `fgStack_push()` once before calling `socketManager`. The `fgStack_push()` function passes a filter. You can leave it null if you do not want to pass a filter value.

For view detail screens, you can use the `put_vararg()` function instead of the `fgStack_push()` function. The `put_vararg` function works in much the same way as `fgStack_push`. View detail screens require four calls to `put_vararg()`. Notice below how the the `put_vararg()` function works in pairs. The first call establishes what is coming next. So in essence, you only need two elements to call a view detail screen. You need an order clause and a join clause.

The following case statement calls both secondary screens:

```
case
    when item = 1
      call fgStack_push("")
      call socketManager("shipzm", "zoom", "default")
    when item = 2
      call put_vararg("order")
      call put_vararg("company")
      call put_vararg("join_elems")
      call put_vararg("p_stomer.customer_num")
      call socketManager("infodt", "view detail", "default")
```

Don't forget that you also must define the variable used by the case statement, like the `item` variable above, before using it in the case statement.

When you put all these code pieces together and run the Featurizer to merge the code, you create a pop-up menu that calls secondary screens. For example, the Customer Info option initiates the following view detail screen:

A view detail screen initiated from a pop-up menu.

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

      Company                First Name        Last Name
      ---------------------  ----------------  ----------------
      AA Athletics           Roy               Jaeger
      All Sports Supplies    Ludwig            Pauli
      Athletic Supplies      Charles           Ream
      Blue Ribbon Sports     Dick              Baxter
      Gold Medal Sports      Alfred            Grant
      Kids Korner            Arnold            Sipes
```

Section Summary

- You can add pop-up menus to your input programs. Pop-up menus can present users with a set of program options.
- The `textput` and `textpick` functions build the frame work for a pop-up menu. The `textput` function loads the items that appear on a pop-up menu. It is called once for each item on the menu. The `textpick` function displays the pop-up menu with all of the items on it. It also assigns each item a sequential numeric value according to the items position on the menu (i.e., the top item is assigned number one). When the user selects an item, the value corresponding to that item is returned.
- You can initiate a pop-up menu from any standard program event. For example, you can create a pop-up menu that appears after the user moves past a certain input field. Such a pop-up menu is placed in an `after_field` trigger.
- You can also initiate pop-up menus from defined program events that you've mapped to a hot key. For example, you can map the [F6] key to the popup event. When the user presses [F6], the pop-up menu appears.
- Pop-up menus can supply the user with all sorts of menu items. One of the most useful items initiates a secondary screen, such as a zoom or view detail screen. You can use the `socketManager` function to attach these screen types to a pop-up menu.

Exercise 14

Objective: To create a pop-up menu that is accessible from anywhere within the Customer Entry program.

This pop-up menu will contain two options. One option will open the Sales Person add-on screen that you created in Exercise 11B. The other option will open a new zoom screen, which will display a list of sales people. The following steps outline the method you will use to create this pop-up menu:

1. Create the Sales Person zoom screen.
2. Create the pop-up menu and the logic to initiate it.
3. Build the logic to link the pop-up menu options to their respective screens.

Create a Zoom

1. **Start the Form Painter.**
2. **Create a new zoom screen that shows all the values of the `salesrep` table. Name the screen `saleszm` and make it look as follows:**

```
Form Editor:  [ESC] or [DEL] Command Line                    [CTRL]-[w] Help
Update data entry image
=======(stores)==================(saleszm)======(Zoom)============(11.5)===

     -------------- Sales Representative Zoom Screen --------------------

          Sales Code   Sales Name              Commission Code
          ----------   --------------------    ---------------
          [  ]         [                   ]   [      ]
          [  ]         [                   ]   [      ]
          [  ]         [                   ]   [      ]
     ------------------------------------------------------------------------
```

3. **When finished, save `saleszm` and generate 4GL code for it.**

Add an on_event Trigger

Instead of attaching `saleszm` to a field, you will attach it using an `on_event` trigger. This method gives you the ability to initiate `saleszm` from your pop-up menu instead of from a field.

You will initiate the pop-up menu with the zoom event. This event is run whenever a user presses [CTRL]-[z]. Instead of opening a specific zoom screen, however, this event will now open a pop-up menu, which will lead to either your `saleszm` screen or your `reps` add-on header screen.

1. **Use vi to open `cust.trg` and add the following code to the input 1 section.**

```
on_event zoom
    call textput("View a sales person")
    call textput("Add a sales person")

    call textpick("Select a Screen")
        returning picker_item

    case
        when picker_item = 1
            call fgStack_push("")
            call socketManager("saleszm", "zoom", "default")
        when picker_item = 2
            call fgStack_push("A")
            call fgStack_push("")
            call fgStack_push("")
            call socketManager("reps", "add-on header", "default")
    end case;
```

   As you can see, this code builds a pop-up menu. The `textput` and `textpick` functions create the menu itself. The case statement evaluates which menu item gets selected and calls the appropriate screen.

2. **In the defaults section of `cust.trg`, add a `define` trigger and a saleszm line to your `switchbox_items` trigger.**

```
defaults
    switchbox_items
        reps S_reps
        saleszm saleszm;

    define
        picker_item smallint;
```

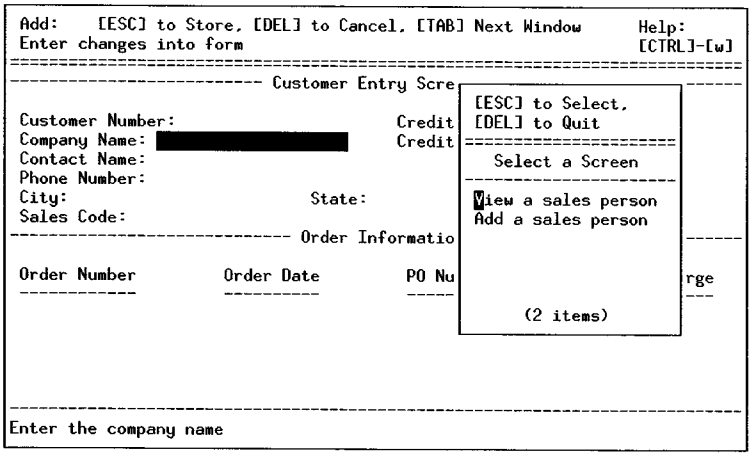
The `saleszm` line in the `switchbox_items` trigger declares the `saleszm` screen to the `switchbox` function. The `define` trigger simply assigns a variable type to `picker_item`.

3. **Save `cust.trg` and compile code.**
4. **When complete, run your Customer Entry program.**
5. **Select Add to create a new record.**
6. **Press [CTRL]-[z] in the Credit Code field.**

   Notice that the Credit Information zoom appears. Although your pop-up menu is "triggered" by the zoom event, your Credit Information zoom takes precedence in the Credit Code field.

7. **Add a credit code and move to the Company Name field.**
8. **Press [CTRL]-[z] again.**

   This time your pop-up menu appears.



Try out your pop-up menu. Do both screens work? You probably need to add some records to the salesrep table. Once you add a couple records, try out the Sales Person zoom screen.

9. **When your finished experimenting, quit out of Customer Entry. Remain in your `i_cust.4gs` directory, however.**

15

Creating Extension Screens

Main topics:

- Extension Screen Overview
- Attaching Extension Screens to Main Screens

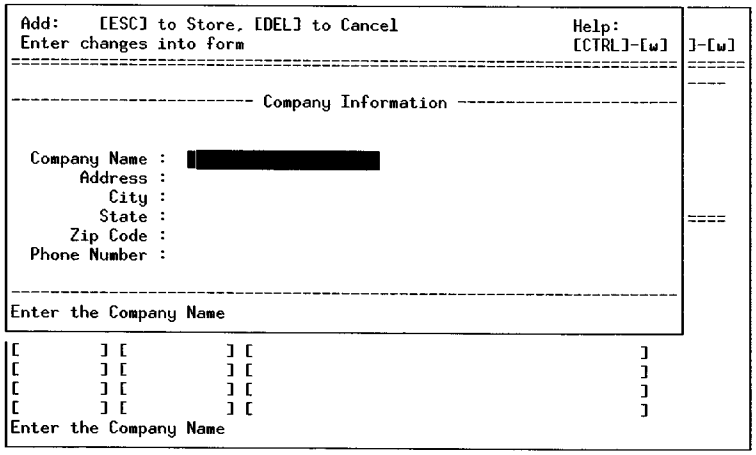
Extension Screen Overview

Extension screens provide users with additional screens. In effect, extensions screens "extend" the main screen.

Many times, tables contain too many columns to fit on a single input screen. Because of a limited amount of "screen geography," it is sometimes useful to create extension screens off of the main screen. By adding extension screens you can simplify and clarify your main screen.

In addition, extension screens can provide conditional data-entry logic. For example, one of your input programs might contain a Payment Method field. Perhaps your company recognizes three types of payment methods: cash, check, and charge. Depending on the value in the Payment Method field, you can initiate different extension screens. Say for example that the charge value initiates an extension screen that contains Card Type, Number, and Expiration Date fields.

The following figure shows an extension screen for adding additional customer information:



Attaching Extension Screens to Main Screens

Extension screens, like the other screen types you've learned about, are attached to the main screen by the socketManager function. But also like the other screens, extension screens use a syntax all their own.

You can initiate an extension screen from an program event. There are useful triggers that work well with extension files, such as:

- `after_input`
- `after_field`
- `before_field`
- `on_event`

You can also map hot keys to custom program events to initiate extensions screens or you can initiate them from a pop-up menu (see "Initiating Secondary Screens from Pop-Up Menus" on page 14-5).

You must complete the following basic steps to create and attach an extension screen:

1. Use the Form Painter to paint the extension image and save the image to a form specification (*.per) file (see "Creating a Form Image" on page 2-5).
2. Create a switchbox_items trigger to declare the extension screen to the screen-level switchbox function (see "The switchbox_items Trigger" on page 12-6).
3. Create a trigger that initiates the extension screen.
4. Use `socketManager` to attach your extension screen.

1. Paint the Extension Screen Image

Use the form painter to create the extension screen image for your extension screen. When you create the screen, make sure to select extension as the screen type. Remember, extension screens are for

additional input fields that cannot fit or are not contained on the main screen. Unlike add-on header screens, extension screens work off the same table as the main screen.

Once you create the image, save it with the Save Form option under the File pull-down. The Save Form option generates a form specification (*.per) for your extension screen.

2. Add the Extension Screen to the switchbox_items Trigger

Like other screens, you need to declare extension screens using the `switchbox_items` trigger. For example, if your extension screen is named `custext`, your `switchbox_items` trigger would look as follows:

```
defaults
switchbox_items
    custext S_custext;
```

3. Create a Trigger to Initiate the Extension Screen

Next, create a trigger that initiates the extension screen. For example, if you want to initiate your extension screen after the user moves past the Customer No. field, insert the following lines of code:

```
input 1
after_field customer_num
```

4. Use socketManager to Attach the Extension Screen

Finally, use `socketManager` to attach the extension screen. Unlike the add-on header and zoom screen types, extension screens don't require you to use the `fgStack_push` function. You only need to use the `socketManager` function. For example, to attach the `custext` extension screen to your main screen, insert:

```
input 1
after_field customer_num

    call socketManager("custext", "extension", "flat_ext");
```

When you attach extensions screens with `socketManager`, the flow parameter differs slightly. Instead of using `default` as the flow parameter, extension screens use one of three values: `flat_ext`, `deep_ext`, and `view`. Extension screens require multiple flow values

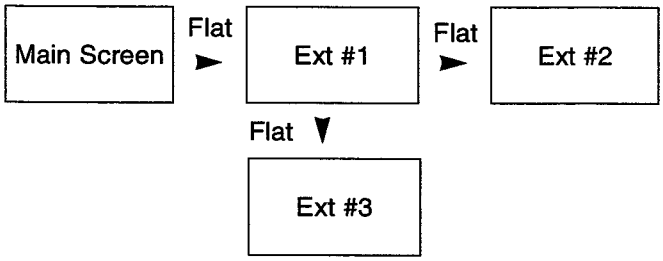
because you can link multiple extension screens together. The following list explains the different flow parameters available with extension screens.

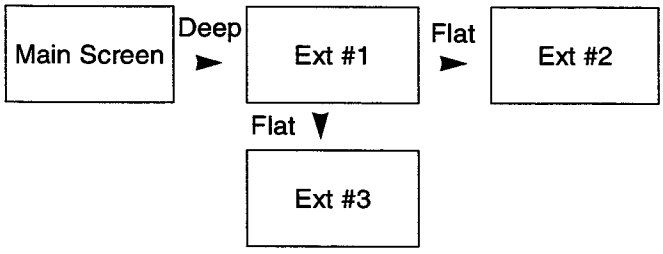
`flat_ext` The `flat_ext` flow parameter determines how the program handles an interrupt (i.e., user pressing [DEL]). If a user presses [DEL] in Ext #1 in the first diagram below, all edits to Ext #2 and #3 are retained.

`deep_ext` The `deep_ext` flow parameter operates in the exact opposite of the `flat_ext` parameter. If a user presses [DEL] in Ext #1, all edits in Ext #2 and Ext #3 are rolled back.

`view` This flow only lets users view the data within extension screens.

If a user pressed [DEL] in Ext #1, all edits to Ext #2 and #3 are rolled back.





By putting all these code pieces together and using the Featurizer to merge your trigger file, your extension screen gets attached.

Section Summary

- Extension screens provide users with additional screens. In effect, extensions screens "extend" the main screen.
- Extension screens provide extra space, so you can simplify and clarify the main screen. In addition, extension screens can be used as data-entry control devices.
- You can initiate an extension screen from any program event, such as a trigger, a pop-up menu, or a mapped hot key.
- You attach extension screens with the `socketManager` function. The `socketManager` recognizes three different flow parameters for extension screens: `flat_ext`, `deep_ext`, and `view`.
- Each flow parameter has a different function. The `flat_ext` flow is for extension screens that are independent from calling screens. The `deep_ext` flow is for extension screen that are dependent on calling screens. The `view` flow is for extension screens that only display data (i.e., users can't add or update values on a view extension screen).

Exercise 15

Objective: To create an extension screen that allows you to enter additional data onto the `customer` table.

You will start by adding three columns: `card_no`, `exp_date`, and `card_holder`. You will then place these columns on a `custext` extension screen. Finally, you will incorporate this screen into your Customer Entry program with an `after_input` trigger.

Add the Columns

1. **Using the Form Painter (or isql) add the following columns to the `customer` table.**

Column Name	Description	**Type**
card_no	Card Number	char(20)
exp_date	Expiration Date	date
card_holder	Card Holder	char(20)

2. **Save these changes and return to the Form Painter.**

Create the Extension Screen

1. **Use the Form Painter to create a new form. Name it `custext` and define it as type extension.**

2. **Create a title line.**

   ---------- Additional Customer Fields ----------

3. **Label and define three fields, one for each of the columns you just added. Your extension screen should look as follows:**

```
|------------------------ Additional Customer Fields -------------------------|
|                                                                             |
| Card No.:[                    ]                                             |
| Expiration Date:[          ]                                                |
| Card Holder:[                      ]                                        |
|                                                                             |
|-----------------------------------------------------------------------------|
```

Save and Generate

1. **Use Save Form to save your new form.**
2. **Invoke the Screen Code Generator to create 4GL code for your new form.**
3. **When the Generator has finished, exit the Form Painter and list your files (type `ls` at the UNIX prompt).**

   Notice that the Generator has created a new source code file for your `custext.per` file. This source code file contains all the lowlevel source code to drive your `custext` extension screen.

Create an after_input Trigger

You can use the `after_input` trigger to attach your `custext` extension screen to your Customer Entry program. Several other triggers will work as well, but the `after_input` trigger is a common choice.

1. **Use vi to open `cust.trg`.**
2. **In the input 1 section add the following lines of code:**

```
# after_input trigger to call my custext extension screen
      call socketManager("custext", "extension", "flat_ext");
```

Note

If you already have an `after_input` trigger defined, which you should because you created one in Exercise 10C, you must add these lines below it. You do not, and cannot, add two identical triggers (for example, two `after_field customer_num` triggers). You should just combine the code under one trigger. Make sure to remove the semi-colon that terminates the first `after_input` trigger or a syntax error will occur.

Your complete `after_input` trigger should look as follows:

```
after_input
    if p_stomer.credit_code is null
    then
        error "You must fill in the Credit Code field"
        let nxt_fld = "credit_code"
    end if
```

```
# after_input trigger to call my custext extension screen
     call socketManager("custext", "extension", "flat_ext");
```

3. **Also add a custext line to your `switchbox_items` trigger.**

   This trigger should now include three lines. A `reps` line, a `saleszm` line, and a `custext` line.

```
switchbox_items
    reps S_reps
    saleszm saleszm
    custext S_custext;
```

4. **Save `cust.trg`.**

5. **Compile the code and run Customer Entry.**

6. **Select Add to create a new record.**

7. **Fill in all the fields on the header portion of the screen and press [TAB] to move to the detail portion.**

   Your `custext` extension screen appears.

```
Add:    [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
================================================================================
---------------------- Additional Customer Fields ------------------------

Card No..: ████████████
Expiration Date:
Card Holder:

--------------------------------------------------------------------------------
Enter the card number
```

```
Order Number        Order Date        PO Number        Shipping Charge
------------        ----------        ---------        ---------------

--------------------------------------------------------------------------------
Enter sales code
```

8. **Complete the Additional Customer screen and quit out of your Customer Entry program.**

16

Version Control and Conventions

Main topics:

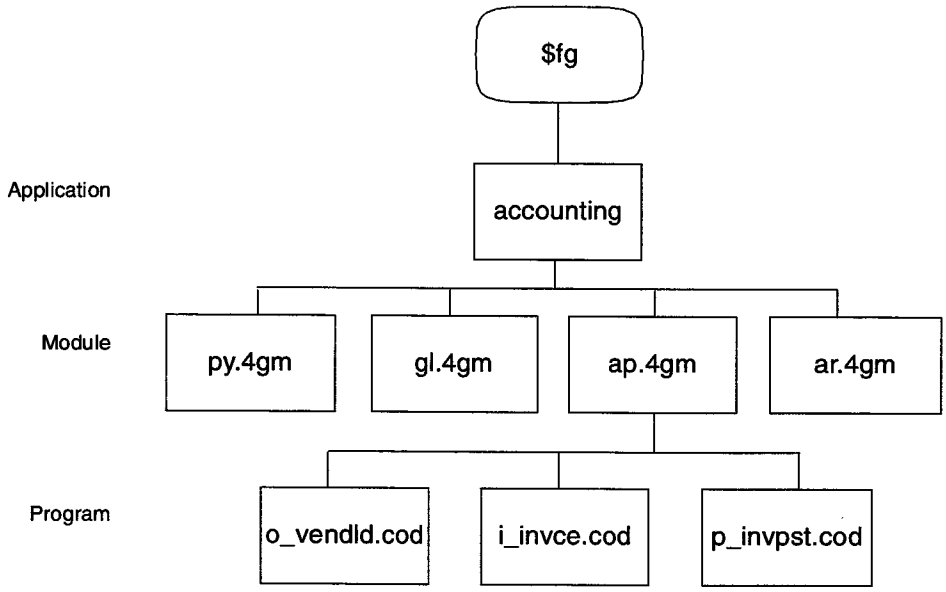
- The Directory Structure
- Version Control Overview
- Building Custom Versions
- Table Naming Conventions

The Directory Structure

All software products utilize a four-tiered directory structure: fourgen, application, module, and program.

The directory contains all your programs. It is usually represented by the `$fg` environment variable. The application tier is rather general. It contains a set of related modules. The module level is more specific. Every module directory is given a .4gm extension. Within each module directory exists a set of related programs. The program tier is the lowest tier. Each program directory contains a single input, output, or posting program. Program directories have a .4gs extension.

The following graphic shows a sample directory structure:



Version Control Overview

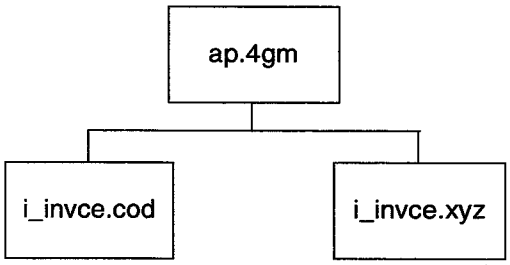
Version control lets you create multiple flavors of a program without duplicating code. Version control is useful when two or more users require different program functionality.

Version Control Directories

Version control uses custom directories that are parallel to program (*.4gs) directories. In the custom directories, you place specification or trigger files that are unique to your custom version. By default, version control recognizes *.4gc directories as custom version directories. You can have as many custom version directories as you want. For example, if you want to have a custom versions of the Enterprise Invoice Entry program, you need to create a custom directory.

Module directory.

Program and Version Control directories.



The `cust_path` variable lets you specify the order in which version control works. To merge base functionality with the new functionality you've added in i_invce.xyz, set `cust_path` as follows:

```
cust_path = xyz:cod ; export cust_path
```

The `cust_key` variable describes the starting point from which the merge utility should start on the `cust_path`. For the above example, `cust_key` should be set as follows:

```
cust_path = xyz:cod ; export cust_path
```

Building Custom Versions

You can use version control logic to build multiple versions of a base program or to build increasingly rich enhancements to a base program. Perhaps the simplest case involves modifying an input screen.

For example, suppose you are customizing the `i_invce.4gi` program, which is located in `ap.4gm/i_invce.cod` directory. You know that this program is built from a series of form specification (*.per) files, where each *.per file represents a different program screen. If, on your custom version, you want to add an input field to the main screen, you would need to complete the following steps:

1. Copy the form specification file you want to modify into the custom program directory (`i_invce.xyz`).
2. Use the Form Painter to add a field to the screen.
3. Run the *Screen* Code Generator (`fg.screen`) in the custom directory (`i_invce.xyz`).

Once initiated, the *Screen* Code Generator takes the following steps:

1. Searches your current directory (`i_invce.xyz`) and reads the modified form specification file.
2. Searches the base directory (`i_invce.cod`) for additional specification files.
3. Generates the 4GL code necessary to build your custom program.

You can then run the `fg.make` utility in the custom directory to compile the custom program.

Once complied, you can issue the following command to run the custom version:

```
fg.go *4gi
```

Table Naming Conventions

All tables follow a specific naming convention. Each table name is composed of eight characters, and the last six characters must be unique. The eight-character name is divided into four sections.

Table names are divided into four sections.

The first two characters classify the table as either a application table or a *Screen* Code Generator table:

st	Application Table
cg	Code Generator Table

The third character relates to the product, for example:

s	*Screen*
d	Database Program in Form Painter
m	User-Defined Menus
x	Non-Product Specific

The next four characters classify the type of data, for example:

eror	Error Text
help	Help Text
mssg	Messages
note	User-Defined Notes

The last character specifies the role of the table:

r	Reference (usually the same as a header)
d	Detail
h	Header

Section Summary

- uses a four-tiered directory structure. The top tier is set by the `$fg` variable. It points to the installation directory. The second tier is known as the application directory. It contains an entire suite of related modules. uses `accounting`, `codegen`, and `distribution` as application directory names.
- Beneath the application tier is the third tier or module tier. Each module directory contains a set of related input, output, and posting programs. All module directories use a *.4gm extension. For example, you might have a set of Accounts Payable programs in a `ap.4gm` directory.
- The final and fourth tier is known as the program directory. Each program directory contains a single generated program built by *Screen* or *Report*. Enterprise program directories use a *.cod extension.
- Version control lets you create multiple flavors of a program. Version control is useful when you want to customize base functionality.
- All tables follow a specific naming convention. Each table name is composed of eight characters and the last six characters must be unique. The eight-character name is divided into four sections

Exercise 16

Objective: To create a custom version of your Customer Entry program in a version control directory.

Version control is extremely useful when you need to customize a specific portion of your base program. In this exercise you will modify your Customer Entry program in a version control directory. You will end up with two versions of Customer Entry, but you will only have one code stream.

Create a i_cust.4gc Directory

By default, recognizes the *.4gc extension as a version control extension. To create a version control program, you must create a new directory parallel to your program directory. Give this new directory the same name as your program directory, but replace the *.4gs extension with a *.4gc extension.

1. **Use the `cd` command to move to your `aw.4gm` directory.**
2. **Use `mkdir` to create a new directory. Name it `i_cust.4gc`.**
3. **Use `cd` again to move to `i_cust.4gc`.**

Copy your *.per Files

1. **In your `i_cust.4gc` directory, copy over all your form specification (*.per) files from `i_cust.4gs`:**

   ```
   cp ../i_cust.4gs/cust.per .
   ```

2. **Start the Form Painter from your `i_cust.4gc` directory and open your `cust.per` file.**
3. **Change the title line to read:**

   ---------- Acme Inc Customer Entry Screen ----------

   In most cases, you will customize more than the title line. But changing this line adequately demonstrates version control.

4. **Save your changes and exit the Form Painter.**

Generate and Compile

1. **From `i_cust.4gc`, run the Screen Code Generator.**
2. **When the Generator finishes, use `ls` to view the files it created.**

   Notice that the Generator creates a whole new set of *.4gl files and a `Makefile`.

3. **Now run `fg.make` to link and compile the code.**

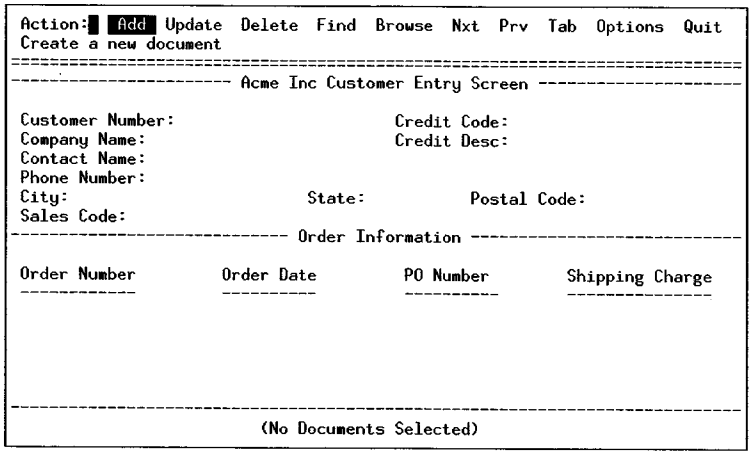
Run Your Custom Version

After you generate and compile, run your version control program:

- **`fg.go i_cust.4gi`**

The `fg.go` runner is for executing version control programs. You execute base applications using `fglgo`.

Once you initiate Customer Entry, your custom version appears:



17

Compiling Generated Code

Main topics:

- Compiling Generated Code
- Modifying Libraries
- Understanding the Library Philosophy
- Adding a Custom Library

Compiling Generated Code

Compiling generated code means turning 4GL source code and triggers into a runnable program. gives you the ability to do this for a single program or a group of programs.

You compile code using the Make Utility. This utility is run with the `fg.make` command. When you run `fg.make`, it completes the following tasks:

- Merges Custom Code: The fg.make command calls the Featurizer (fglpp) program. The Featurizer merges custom code into your program source code (see "Featurizer Overview" on page 18-2).
- Compiles Source Code and Form Specification Files: fg.make also compiles both your source code (.*4gl) files and form specification (*.per) files.
- Links Local Function Calls to Library Functions: The `fg.make` command resolves library function calls in local (i.e., source code) to their corresponding library functions.
- Produces Runnable Program File: The last task of fg.make is to construct a runnable program file. The fg.make command creates a different program file depending on the type of Informix development system you are using. If you are using the INFORMIX-4GL C Compiler, a *.4ge program file is created. If you are using the INFORMIX-4GL RDS Compiler, a *.4gi file is created.

The final three tasks are controlled by the standard UNIX `make` utility, which is called by `fg.make`. In general, the UNIX `make` utility tracks the dependencies that files have to each other.

The UNIX `make` utility uses a specification file of its own. This file, called the `Makefile`, contains all the instructions necessary for `make` to work. You do not have to create the `Makefile`, however. It is created automatically by the *Screen* Code Generator.

For the most part, you do not need a complete understanding of the UNIX `make` utility in order to use it. You should simply realize that it is called from the `fg.make` command and it produces a program file that you can run.

The `fg.make` command uses a number of command flags:

```
fg.make [-h] [-F|-R] [-L library] [-m{n|o|f|of}]
[-f] [-a]
```

`-h`	Prints an entire list of fg.make command flags.
`-F`	Forces fg.make to compile using the INFORMIX-4GL C Compiler.
`-R`	Forces fg.make to compile using the INFORMIX-4GL Rapid Development System.
`-L library`	Specifies the name of any additional libraries you want fg.make to link in.
`-mn`	Does everything except merge code. In other words, when you use the -mn flag, the Featurizer is not called.
`-mo`	Runs the Featurizer (merges code) but does not perform a compilation.
`-mf`	Overrides timestamp comparison logic and forces a custom code merge.
`-mfo`	Overrides timestamp comparison logic and forces a custom code merge. This flag does not perform a compilation, however.
`-f`	Performs a fast link. You should only use this flag in compiles where no new calls to library functions have been added.
`-a`	Causes all files to be compiled regardless of dependencies.

The Makefile

It does help, though, to have a working knowledge of the `Makefile`. The `Makefile` contains several sections. Each section supplies `make` with information about your program.

```
########################################################################
# Copyright (C)
# 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%
########################################################################
# Screen Generator version: 4.11.UD1

#  Makefile for an Informix-4GL program

#_type - Makefile type
TYPE = program

#_name - program name
NAME =     screen3.4ge

#_objfiles - program files
OBJFILES =  browse.o cust_zm.o detail.o globals.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

#_libfiles - library list
LIBFILES =  ../lib.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.m
```

As you can see, the `Makefile` lists the files necessary to create your program. For example, the `LIBFILES` section shows all the libraries used by your program (`lib.a`, `scr.a`, `user_ctl.a`, and `standard.a`).

Library Overview

A library holds code shared by multiple programs. The code is structured into functions. Each function performs a single task and works independently from other code. For example, several programs require a message that reads, "Please wait." Instead of duplicating the same lines of code in each program directory, you can simply place a call to the library function that displays the "Please wait" message.

Screen makes extensive use of libraries. These libraries are contained in the `$fg/lib` directory. Each library contains related functions. For example, the `standard` library contains functions shared by both input and output programs, such as the "Please wait" message, which is in the `pls_wait.4gl` file:

```
########################################################################
function please_wait()
########################################################################
#
    # Trap fatal errors
    whenever error call error_handler

    if mssg_prep is null
    then
        let mssg_prep = "Y"
       #1:    " Please wait..."
        let arr_mesgs[1].mssg_text =
        fg_message("standard","pls_wait",1)
    end if

    call variableText_message(arr_mesgs[1].mssg_text, 0)

end function
# please_wait()
```

Creating Custom Libraries

If you have programs that share common functions that are not in the libraries, you can create your own custom library.

Custom libraries are created at the module directory level (the *.4gs level). Just like the libraries, custom libraries contain functions that perform specific, independent tasks. These functions are placed in source code (*.4gl) file.

For example, to create a custom library called `mylib`:

1. At the program directory level, create a `mylib.4gs` directory.

   ```
   mkdir mylib.4gs
   ```

2. Move to `mlib.4gs` and create each custom function in its own source code (*.4gl) file.

3. Copy a library `Makefile` into `mylib.4gs`.

   ```
   cp $fg/lib/standard.4gs/Makefile mylib.4gs
   ```

   In order to compile your library code there must be a `Makefile` present. You can build a `Makefile` by hand or you can modify the one in the `standard.4gs` library.

4. Replace the `Makefile`'s `LIBFILES` section with your function filenames.

   For example, if `mylib.4gs` contains `wincl.4gl`, `windl.4gl`, and `winop.4gl`. The LIBFILES section should read:

   ```
   LIBFILES = \
       $(LIB)(wincl.o) \
       $(LIB)(windl.o) \
       $(LIB)(winop.o)
   ```

5. Change the `LIB=../standard.a` line to read:

   ```
   LIB= ../mylib.a
   ```

6. Finally, run `fg.make` in the `mylib.4gs` directory.

Using a Custom Library

Once you create a custom library, you can use it in your programs. You must add your custom library to the `LIBFILES` section of the program's `Makefile`. In other words, if you call custom library functions in your program code, you must tell the UNIX `make` utility where to look to find the custom library functions.

For example, if your program calls `windl`, which is in your custom `mylib.4gs` library, the `LIBFILES` section must include `mylib.a`.

You can add libraries to your program's `Makefile` using the `libraries` trigger. For example, the following `libraries` trigger adds the `mylib` library to your program's `Makefile`:

```
defaults
    libraries
        ../mylib.a
    ;
```

This trigger changes your `Makefile` to look as follows:

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

Another trigger, the `custom_libraries` trigger, also lets you add libraries to your program's `Makefile`. The `custom_libraries` trigger places your custom library above the ../lib.a \ line in the `Makefile`. For example, the following `custom_libraries` trigger places your `mylib` library first on the `LIBFILES` list.

```
defaults
    custom_libraries
        ../mylib.a
    ;
```

This trigger changes your `Makefile` to look as follows:

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

Section Summary

- Compiling generated code means turning 4GL source code into a runnable program. gives you the ability to do this for a single program or a group of programs.
- You compile code using the Make Utility. This utility is run with the `fg.make` command.
- The `fg.make` command merges custom code, compiles source code and form specification files, links local function calls to library functions, and produces a runnable program file.
- The `fg.make` command uses the standard UNIX `make` utility, which is called by `fg.make`. In general, the UNIX `make` utility tracks the dependencies that files have to each other.
- The `Makefile` contains several sections. Each section supplies `make` with information about your program. For example, the `LIBFILES` section shows all the libraries used by your program.
- A library holds code shared by multiple programs. The code is structured into functions. Each function performs a single task and works independently from other code.
- *Screen* makes extensive use of libraries. These libraries are contained in the `$fg/lib` directory.
- If you have programs the share common functions that are not in the libraries, you can create your own custom library.
- Custom libraries are created at the module directory level (the *.4gs level). Just like the libraries, custom libraries contain functions that perform specific, independent tasks. These functions are placed in source code (*.4gl) files.
- Once you create a custom library, you can use it in your programs. You must add your custom library, however, to the `LIBFILES` section of your program's `Makefile`. In other words, if you call custom library functions in your program code, you must tell the UNIX `make` utility where to look to find the custom library function.

Exercise 17A

Objective: To create a custom library and add a function to it.

Create a Library Directory

1. **Use the `cd` command to move to your `aw.4gm` directory.**
2. **Use `mkdir` to create a new directory called `mylib.4gs` and use `cd` to move to that directory.**

   This is your custom library directory. Within this directory, you can create custom functions for your programs.

Create a Custom Library Function

1. **Use vi to open a new file called `hello.4gl`.**
2. **Add the following function to your new file:**

```
function hello()
    display "hello fourgen world"
    sleep 3
end function
```

3. **Use `vi` to create a new Makefile that looks as follows:**

```
# Makefile for an Informix function library

TYPE = library

LIBFILES = \
    $(LIB)(hello.o)

FORMS=

LIB=../mylib.a

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

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

4. **While you are still in `mylib.4gs`, run `fg.make`.**

   The `fg.make` script compiles your library and creates a parallel RDS version of your library at the module directory level.

Add a libraries Trigger

To use your new `hello()` function, you must add your custom library to the `Makefile` in your `i_cust.4gs` directory. A special trigger, called `libraries` lets you do this.

1. **Use `cd` to move to your `i_cust.4gs` directory.**
2. **Use `vi` to open your `cust.trg` trigger file.**
3. **Add the following code to the defaults section of `cust.trg`:**

```
libraries
    ../mylib.a
;
```

This trigger adds your custom `mylib` library to the `LIBFILES` list in the program `Makefile`.

4. **Save and quit from `cust.trg`.**

Add a before_input Trigger

To implement your new `hello()` function, you must use it from somewhere in your program. Perhaps the simplest way to use it is with a `before_input` trigger.

1. **Use `vi` to open `cust.trg`.**
2. **In the input 1 section, add the following lines of code:**

```
before_input
    call hello();
```

This trigger simply calls your `hello()` function, which is in your custom `mylib` library.

3. **Save and quit from `cust.trg`.**

Compile the Code

- **Run `fg.make` to compile the code.**

Run Your Customer Entry Program

1. **Run your Customer Entry program.**
2. **Select Add from the ring menu.**

   What happens? Do you see the "hello world" message?

3. **Quit from your Customer Entry program.**

Exercise 17B

Objective: To call `hello()` from the Credit Entry program.

Custom libraries allow you to call custom functions from anywhere in your module directory. In other words, custom libraries work with all the programs in your module. You have already used the `hello()` function in your Customer Entry program. Now you will add a call to this function from your Credit Entry program.

Create a cred.trg Trigger File

1. **Use `cd` to move to the `i_cred.4gs` directory.**
2. **Use `vi` to create a `cred.trg` file.**
3. **Add the following `libraries` trigger to `cred.trg`:**

```
defaults
    libraries
        ../mylib.a
    ;
```

4. **Save and exit `cred.trg`.**

Add a libraries Trigger

1. **Use vi to open `cred.trg`.**
2. **Just below your `libraries` trigger, add the following code:**

```
input 1
    before_input
       call hello();
```

Your complete `cred.trg` file should look as follows:

```
defaults
    libraries
        ../mylib.a
    ;

input 1
    before_input
        call hello();
```

3. **Save and exit `cred.trg`.**

Compile the Code

- **Run `fg.make` to compile the code.**

Run Your Credit Entry Program

1. **Run your Credit Entry program.**
2. **Select Add from the ring menu.**

   What happens? Do you see the "hello world" message?

3. **Quit from your Credit Entry program.**

18

Using the Featurizer

Main topics:

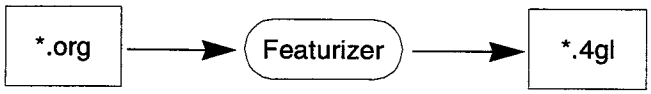
- Featurizer Overview
- Creating Block Commands
- Pluggable Feature Sets
- Triggers Versus Block Commands

Featurizer Overview

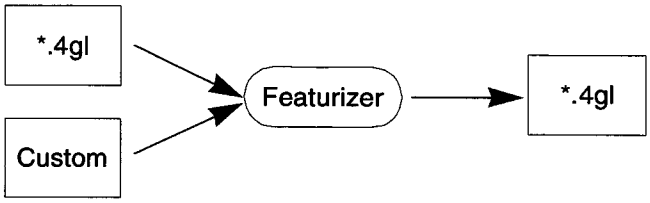
The Featurizer performs two tasks:

1. It copies *.org files, which are created by the *Screen* Code Generator, into *.4gl files.
2. After it creates the *.4gl files, the Featurizer merges the custom code into the source *.4gl files.

The Featurizer copies generated *.org files into *.4gl files.



After copying the *.org files, the Featurizer then merges custom code into source code.



Both the *Screen* Code Generator and the `fg.make` command run the Featurizer automatically. You can also run the featurizer directly with the `fglpp` command.

For instance, if you want to merge custom code into `header.4gl`, type:

```
fglpp header.4gl
```

You have already learned how to create custom modifications in trigger (*.trg) files. In addition to trigger files, though, the Featurizer also reads extension files and merges them into your source code (*.4gl) files. Extension (*.ext) files are similar to trigger files, but extension files act on physical locations in source code. Within extension files you create *block commands*.

Block Commands

Block commands let you customize physical points within generated source code. Because block commands act on physical locations, you must address where you want your block command to go. A code address contains three parts: filename, function name, and block tag.

You already know about filenames and function names, but block tags are a new concept. For example, the `mlh_cursor` function in the `midlevel.4gl` file contains eight block tags. You can easily identify block tags because they all begin with the same two characters (`#_`) followed by their block name. For example, `#_define_var` is the first block tag in the `mlh_cursor` function:

This function contains eight block tags.

```
########################################################################
function mlh_cursor()
########################################################################
#
# This function defines the table, filter, and ordering portions of
# the select statement used to build the FourGen scroller.
#

    #_define_var - define local variables

    #_curs_elements - Cursor table, hard filter, and order

    #_table - cursor table
    call put_vararg("customer")

    #_filter - filter statement
    call put_vararg("")

    #_order - order statement
    call put_vararg("")

    #_dtl_tab - detail table statement
    call put_vararg("")

    #_join - join statement
    call put_vararg("")

    #_translate - Tell upper level about translation
    call put_vararg(is_translated)
    call put_vararg(num_trans)


end function
# mlh_cursor()
```

Block tags pinpoint physical locations within generated source code. When you want to alter source code contained in a block tag, you can use block commands. Block commands use the following syntax:

```
start file "filename"

   block_command function_name block_tag
```

For example, the following block command adds a line to the `#_define_var` block tag in the `mlh_cursor` function:

```
start file "midlevel.4gl"

   after block mlh_cursor define_var

      tmp_num smallint;
```

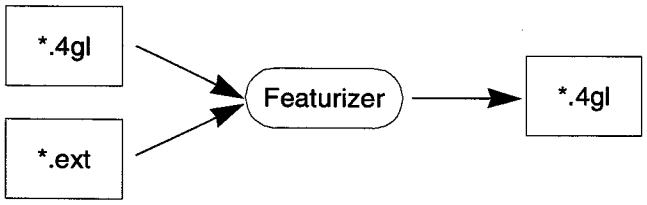
The `start file` command, on the first line, specifies the file to use (in this case it is `midlevel.4gl`).

The first argument on the second line is the name of the block command (in this case it is `after block`). The second argument specifies the function name (`mlh_cursor`). The third argument specifies the block tag minus the #_ characters (`define_var`).

The third line contains your custom code (in this case the third line defines the variable `tmp_num`).

You place block commands within extension (*.ext) files, which get read by the Featurizer and merged into your source code.

The Featurizer reads your extension files and merges them into generated source code.



Pluggable Feature Sets

Unlike trigger files, which the Featurizer reads and merges automatically, you must declare extension files within a *feature set* (`base.set`) file. A feature set file simply contains the names of the extension files you want the Featurizer to merge into your code.

Feature set files are extremely useful because they let you add custom code in a *pluggable* fashion. For example, you may have three extension files that add custom functionality to your program (`acme.ext`, `abc.ext`, and `xyz.ext`). Some departments might want the functionality added by all three extension files while others may only want the functionality in the `xyz.ext`.

For your first group of departments, your `base.set` file would contain the name of all three extension files minus the *.ext extensions:

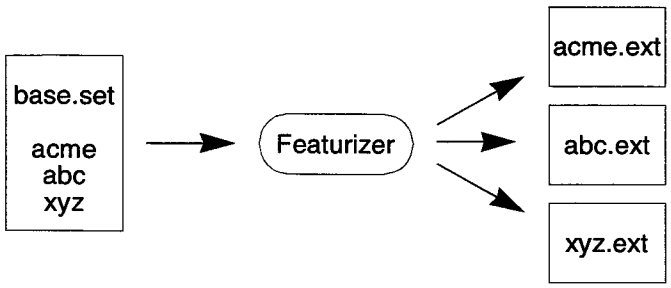
```
acme
abc
xyz
```

For your second group of departments, your `base.set` file would only contain the name of the `xyz.ext` file:

```
xyz
```

When you run the Featurizer, it looks at your `base.set` file to determine which extension files to merge into your source code.

The Featurizer reads your base.set file to determine which extension files to merge.



Section Summary

- The Featurizer performs two tasks: It copies each *.org file into a *.4gl source code file and it merges custom code into the *.4gl files.
- Both the *Screen* Code Generator and the `fg.make` command run the Featurizer automatically. You can also run the featurizer directly with the `fglpp` command.
- Block commands let you customize physical points within generated source code. Because block commands act on physical locations, you must address where you want your block command to go. A code address contains three parts: filename, function name, and block tag.
- You already know about filenames and function names, but block tags are a new concept. You can easily identify block tags because they all begin with the same two characters (#_) followed by their block name.
- When you want to alter source code contained in a block tag, you can use block commands.
- You place block commands within extension (*.ext) files, which get read by the Featurizer and merged into your source code.
- Unlike trigger files, which the Featurizer reads and merges automatically, you must declare extension files within a *feature set* (`base.set`) file. A feature set file simply contains the names of the extension files you want the Featurizer to merge into your code.

Exercise 18A

Objective: To use a block command to add a new ring menu item beneath the Options command.

On every generated input program, the ring menu contains an Options command. This command provides you, the program developer, with an "extra" space to add custom ring menu commands. When the user selects Options, the ring menu clears. With the help of block commands, you can add custom ring menu commands "beneath" the Options command.

Become Familiar with Options

1. **Start you Customer Entry program.**
2. **Select Options from the ring menu.**

As you can see, there is only a Quit command beneath the Options menu. When you press Quit, you return to the main ring menu. In this exercise, you will add a command that starts your Credit Entry program.

3. **Return to the ring menu and then Quit the program.**

Copy optnMenu.4gl to Your Local Directory

Before you can alter your Options ring menu command, you must move the `optnMenu.4gl` file to your local directory (which in your case is `i_cust.4gs`). The `optnMenu.4gl` file is located in the `scr.4gs` library directory.

1. **Use `cp` to copy `optnMenu.4gl` to your `i_cust.4gs` directory. If you are in `i_cust.4gs`, you can use the following command (you should also give this file read and write permission):**

   ```
   cp $fg/lib/scr.4gs/optnMenu.4gl .
   ```

2. **Use `ls` to verify that the copy worked correctly.**

   You should see `optnMenu.4gl` in your `i_cust.4gs` directory.

3. **Use `vi` to open `optnMenu.4gl`.**

   Notice all the lines that begin with #_.

   These characters (#_) indicate a block tag. You can use block commands to alter (i.e., customize) any code within a block tag.

4. **Quit out of `optnMenu.4gl`.**

Build an Extension (*.ext) File

Block commands are created and stored in extension (*.ext) files. In a general sense, extension files are a lot like trigger files. Extension files hold block commands whereas trigger files hold triggers. The major difference is how extension files are merged into base code. As you recall triggers get merged automatically by the *Screen* Code Generator. For extension files, however, you must specify in a feature set file, called `base.set`, which extension files to use.

1. **Use `vi` to create a new file called `menu.ext`.**

2. **Add the following block command to `menu.ext`:**

```
start file "optnMenu.4gl"

before block ring_options quit
    command key (c) "Credit Entry" "Runs the Credit Entry program"
        run "cd $HOME/labs/aw.4gm/i_cred.4gs ; fglgo *.4gi";
```

As you can see, this code modifies `optnMenu.4gl`. It adds code to run the you Credit Entry program to the `menu` block tag in the `ring_options` function.

3. **Save and quit from `menu.ext`.**

Create a base.set File

In order to incorporate your new block command into base code, you must add `menu.ext` to the base.set file.

1. **Use `vi` to create a new file called `base.set`.**
2. **Add the following line to the `base.set` file:**

   `menu`

   This is all you need to add. You do not need to include the `.ext` file extension. If you had more extension (*.ext) files to include, you would list them in the same way.

3. **Save and quit from `base.set`.**

Compile the Code

- **Run the compilation utility (`fg.make`).**

Run Your Customer Entry Program

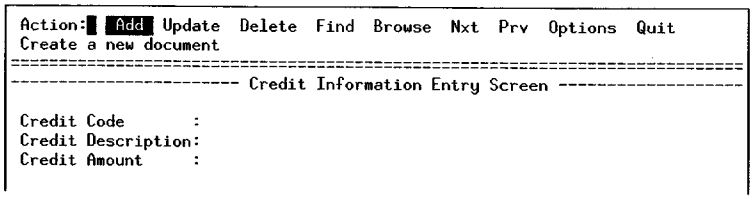
1. **Start your customer entry program.**
2. **Select Options from the ring menu.**

```
 Options: Credit Entry  Quit
Runs the Credit Entry program
================================================================================
```

As you can see, a new command (Credit Entry) has been added to your ring menu.

3. **Select Credit Entry.**

   Your Credit Entry program starts.



4. **Quit out of your Credit Information program.**

   What happens? You should be returned to the Options ring menu within your Customer Entry program.

5. **Quit out of both the Options command and your Customer Entry program.**

Exercise 18B

Objective: To demonstrate the pluggable feature set concept, you will "unplug" your `menu.ext` file.

Unplug menu.ext

1. **Use vi to open your `base.set` file.**
2. **Place a # before the word menu:**

   `# menu`
3. **Save and quit `base.set`.**

Compile the Code

- **Run `fg.make` to compile the code.**

Run Your Customer Entry Program

1. **Start your customer entry program.**
2. **Select Options from the ring menu.**

   As you can see, your custom ring menu command is gone. You can add it back by simply removing the comment mark (#) from your `base.set` file.
3. **Quit from your Customer Entry program.**

Exercise 18C

Objective: To create a generic library function that accepts multiple arguments using the "vararg" functions.

You will create a simple function that displays the names of the programmers who created the program. Since this number changes from program to program, you must use the vararg functions.

Create a Library Function

1. **Use `cd` to move to your `mylib.4gs` directory.**

   This is the library directory that you created in Exercise 17A.

2. **Use `vi` to create a new file called `shw_crd.4gl` and add the following lines of logic to it:**

```
########################################################################
function show_credits()
########################################################################
# This function gives credit to all the members of the programming
# team that created the program
#
    define
        n smallint, # generic number
        people char(30)

    # Trap fatal errors

    whenever error call error_handler

    display "This program has been brought to you by:"

    # here's the loop using num_vararg and get_vararg

    let n = num_vararg()

    while n > 0
        let people = get_vararg()
        display people
        let n = n - 1
    end while

    sleep 3
end function
# show_credits()
```

3. **Save and quit `shw_crd.4gl`.**

Update the Makefile and Compile

1. **Use vi to open the `Makefile` in your `mylib.4gs` directory.**
2. **Add `shw_crd` to the LIBFILES section in the following way:**

```
LIBFILES = \
    $(LIB)(hello.o) \
    $(LIB)(shw_crd.o)
```

3. **Save and quit from the `Makefile`.**
4. **Run `fg.make` to compile your library code.**
5. **Use `cd` to return to your `i_cust.4gs` directory.**

Add a Navigation Event

1. **Run your Customer Entry program.**
2. **Press [CTRL]-[g] to open the Navigate pop-up menu.**
3. **Select Add a navigation action.**

   The Navigation Command window appears.

4. **Complete the Navigation Commands window as follows:**

```
Update: [ESC] to Store, [DEL] to Cancel                          Help:
Enter changes into form                                          [CTRL]-[w]
================================================================================
                              Navigation Commands
--------------------------------------------------------------------------------

Action Code:  show_credits
Description:  Show who created this program
Operating system command:


Press ENTER upon return   ? Y
Access from other programs? Y
Allow access for others   ? Y

--------------------------------------------------------------------------------
Enter [Y] if you want this to appear on everybody else's navigation menu.
```

5. **Press [ESC] to save your definition and quit from your Customer Entry program.**

Create an on_event Trigger

You will add custom logic to call your `show_credit()` function using the `on_event` trigger. This trigger will be accessed via the navigation event you just created.

1. **Use `vi` to open your `cust.trg` file.**

Note

Although this section describes block commands and extension files, this exercise is best completed using the `on_event` trigger. You should note once again the main difference between triggers and block commands: triggers act on logical points within code, block commands act on physical points.

2. **Add the following lines of code to the defaults section of your trigger file:**

```
on_event show_credits
    call put_vararg("Perry Dillard")
    call put_vararg("David Hanses")
    call put_vararg("Robert Cumpston")
    call show_credits();
```

3. **Save and quit `cust.trg`.**

Compile the Code

- **Use `fg.make` to compile the code and merge in your new `on_event` trigger.**

Run Your Customer Entry Program

1. **Start Customer Entry.**
2. **Press [CTRL]-[g] to display the Navigate pop-up menu and select the "Show who created this program" option.**

   The programmer names appear at the bottom of the screen.
3. **Quit your Customer Entry program.**

19

Getting Started with Menus

Main topics:

- Benefits of *Menus*
- Files Used by *Menus*
- *Menus* Directory Structure
- Starting a *Menus* Program

Benefits of *Menus*

Menus provides an attractive environment for users to run programs you create with the FourGen *CASE* Tools. *Menus* are:

- Simple to create and modify.
- C-based and pre-compiled. All your modifications happen in *real-time.*
- Attractive and sophisticated. You can attach sibling menus or subordinate menus. You menus can cascade over each other. You can also control the placement of a menu on the screen.
- Capable of running UNIX commands. Common UNIX commands that users invoke, such as checking disk space, can be attached to a menu choice. Through menus, the user has a friendly way of executing UNIX commands.
- Able to set UNIX environmental variables.
- Securable. You can instruct *Menus* to prompt for passwords or deny people or groups access to menu choices.

Files Used by *Menus*

Menus uses several files to control how the menus look and operate. These files include image (*.img) files and item instruction files.

Image Files

The image of a menu is kept in an image file, which can be named `menu` or `menu.img`. You use a text editor to create the menu image. For example, here is a sample menu image file:

```
@       ABC COMPANY            @
~|                            |~
~|% 1 %- ABC Inquiry          |~
~|% 2 %- ABC Messages         |~
~|                            |~
~+----------------------------+~
```

The display characters that you see have special meanings:

`@ text @`	Highlights text between two @ symbols.
`|`	Creates a vertical border character.
`+`	Creates a corner character.
`-`	Creates a horizontal border character.
`% text %`	Highlights text between two % symbols when menu item is selected.
`~`	Creates a transparent space. It is important to have a transparent space around menus that overlap.

Item Instruction Files

Item instruction files hold *item instruction commands*. Item instruction commands are a series of commands specific to menus. When the user selects a choice off of the menu, the item instruction(s) in the respective item instruction file are executed.

For instance, if a user selects the first option on the following menu, an item instruction file named 1 is executed.

```
@       ABC COMPANY            @
~|                             |~
~|% 1 %- ABC Inquiry           |~
~|% 2 %- ABC Messages          |~
~|                             |~
~+-----------------------------+~
```

Menus executes the item instruction commands within this file. For example, the following shows a sample item instruction command:

```
:unix:echo "You selected ABC Inquiry":
```

The `unix` item instruction command instructs menus that the next field contains a UNIX command. Item instruction commands and their arguments are always delimited by colons.

The following list shows some commonly used item instruction commands and their meaning:

:unix:	Executes a UNIX command.
:system:	Executes an operating system command.
:ifxscreen:	Runs an input program.
:item:	Notifies the user about menu item functionality.
:show:	Shows menu arguments to user.
:submenu:	Calls up a subordinate menu.
:addmenu:	Calls up a menu at the same level.
:env:	Sets a UNIX environment variable.
:pause:	Prompts the user to press [ENTER] before continuing with the next item instruction command.
:password:	Prompts the user for a password before continuing with the next item instruction command.

:deny:	Denies all users or groups in its argument list access to the menu item.
:allow:	Allows all users or groups in its argument list access to the menu item.
:input:	Prompts the user for input and assigns the input to a UNIX variable.

Menus Files

Item instruction files:

1	Contains item instruction commands for first item on menu.
2	Contains item instruction commands for second item on menu.
3	Contains item instruction commands for third item on menu.

Image files:

menu or **menu.img**	Holds the menu image.
***.act**	Holds cosmetic item instruction commands, such as `:window:` and `:color:`. The prefix must match that of the *.img file that it corresponds to.

Help files:

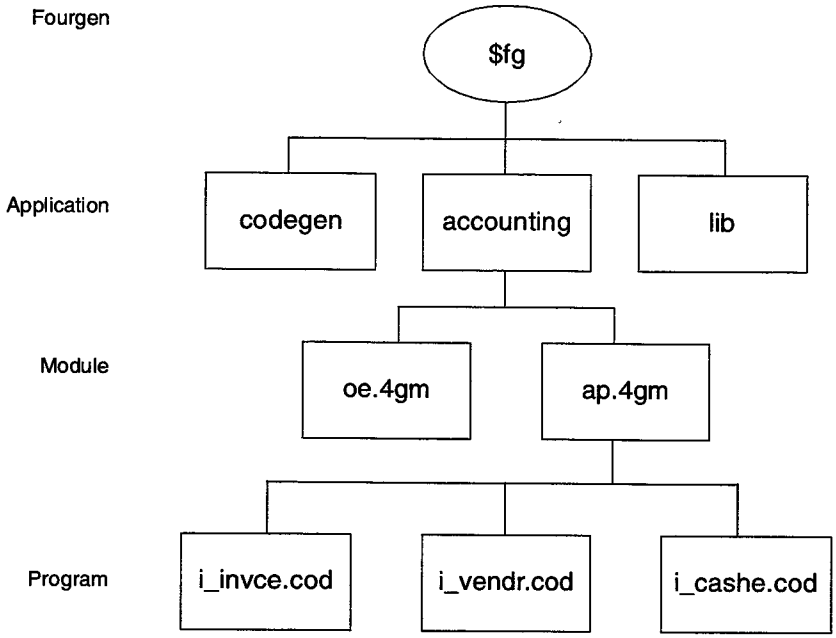
1.help	Contains help text for the first item on the menu.
menu.help	Contains general help text for all the items on the menu.

Permission files:

1.prm	Contains permission instructions for the first item on the menu.

Menus Directory Structure

As you recall, the application directory structure contains four levels: fourgen, application, module, and program. The `$fg` variable always points to the directory.



Menus uses two directory levels: `project` and `menus`.

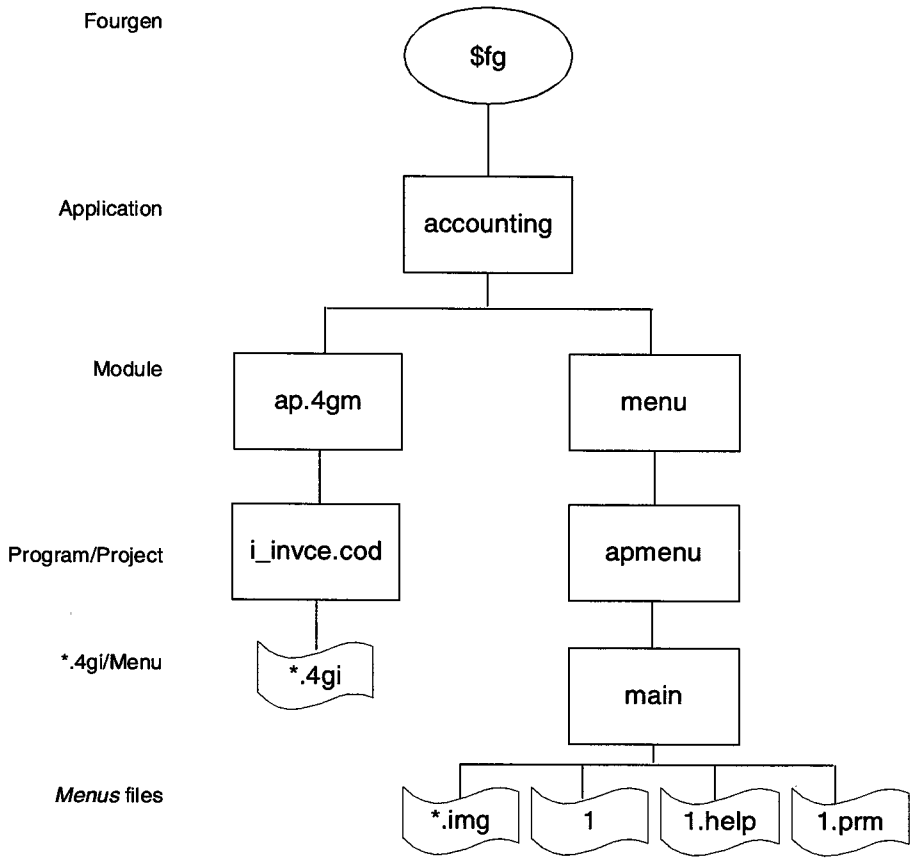
These directories parallel the application directories.

The project directory contains the module name for the menu. For example, if your module directory is `ordmnt.4gm`, your *Menus* project directory should be `ordmenu`, or some other name that uniquely identifies it.

The menus directory contains *Menus* files, such as item instruction files, image files, help files, and security files.

Directory Diagram

The following figure shows *Menus* directory structure in relation to the application structure:



The $mz Variable

The $mz variable points to your `menu` directory at the module level. Before you create or use the *Menus* program set `$mz`, for example:

```
mz=$fg/accounting/menu ; export mz
```

Starting a *Menus* Program

The `mz` command lets you run *Menus* programs. This command is found in your `$fg/bin` directory. It uses the following syntax:

`mz [project_directory] [menu_directory]`

For example, to execute the `apmenu` program, type:

`mz apmenu main`

By default, *Menus* always looks for the `main` directory first. If you there is a main directory beneath the project dirctory, you can leave off the menu directory arguement. For example:

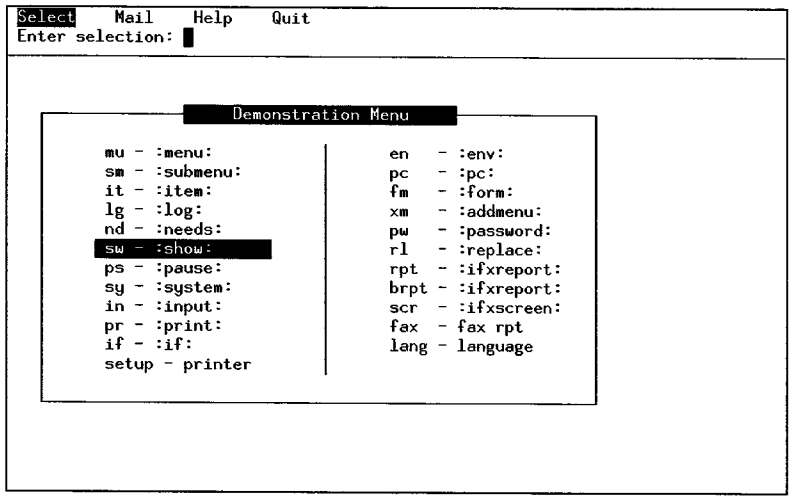
`mz apmenu`

To see a sample *Menus* program, run the *Menus* demonstration:

`mz_demo`

This command starts the *Menus* demonstration, which contains descriptions of several item instruction commands:

The mz_demo command starts a sample *Menus* program.



Section Summary

- *Menus* provides an attractive environment for users to run programs you create with the *CASE* Tools.
- *Menus* not only runs generated programs, but it can also run common UNIX commands.
- The image of a menu is kept in an image file. You use a text editor to create the menu image.
- For each item on a menu, there is a corresponding item instruction file. For example, menu item one has an item instruction file named 1, menu item two has an item instruction file named 2, and menu item three has an item instruction file named 3.
- When the user selects a menu item, the corresponding item instruction file is executed.
- Item instruction files contain short, simple instructions that tell *Menus* what to do.
- *Menus* utilizes two directory levels: project and menu.
- Before you create or run a *Menus* program, you should set the `$mz` variable.
- You can run a *Menus* program with the `mz` command.
- The `mz_demo` command starts a sample *Menus* program.

Exercise 19

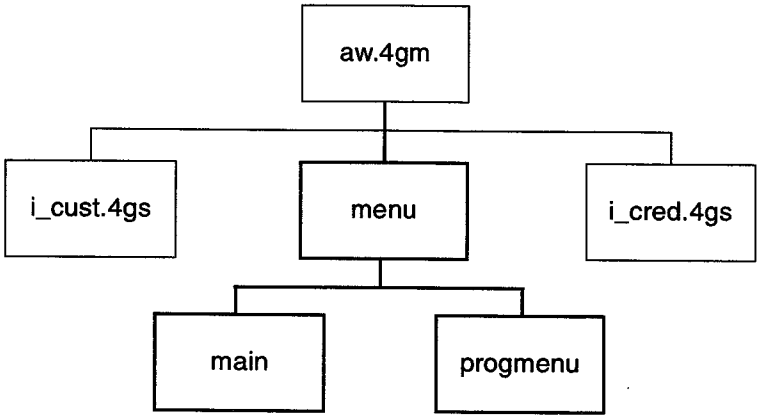
Objective: To create a simple menuing front-end that starts your Customer Entry program.

This part of the Exercises focuses on *Menus*. *Menus* gives you a flexible, easy-to-use front-end for your generated programs. In general, there are three main steps for creating a *Menus* program:

1. Create a *Menus* directory structure.
2. Create the menu image, complete with each menu option.
3. Create the command that starts your program when the corresponding option is selected by the user.

Create a *Menus* Directory

- **From within your module (`aw.4gm`) directory, add the following *Menus* directory structure in alliance with your `i_cust.4gs` and `i_cred.4gs` directories:**



The `menu` directory functions as the topmost menu directory. The other two directories, `main` and `progmenu`, hold the image files and item instruction files for your *Menus* program.

Create the Menu Image

1. **In the `main` directory, use `vi` to create a file called `menu`.**
2. **In this file, add the following character image:**

```
@         ACME INC.          @
~|                           |~
~|% 1 %- Customer Info       |~
~|% 2 %- Credit Entry        |~
~|                           |~
~+---------------------------+~
```

3. **Save and quit `menu`.**

Run Your Menu

You now have a "hollow" menu image that you can run.

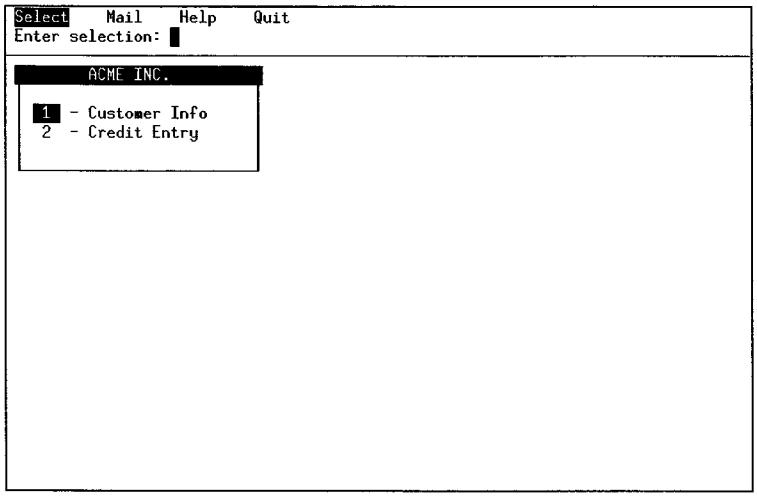
1. **Set your `mz` variable so that it points to your module directory:**

   `mz=$HOME/labs/aw.4gm ; export mz`

2. **Run your *Menus* program with the following command:**

   `mz menu`

   A simple *Menus* program appears:



This program reflects the image you created in the `menu` file. As you will see, it does not yet initiate either of your input programs.

3. **Select item 1.**

   What happens? You should see the following message because you still must attach each menu item to its respective program.

   ```
   Menu item '1' not found.
   ```

4. **Press [DEL] and Q to quit your *Menus* program.**

Attach Your Programs to the *Menus* Front End

1. **In the `main` directory, use `vi` to create a new file called `1`:**

   ```
   vi 1
   ```

   This is the item instruction file for menu item number one. Item instruction files contain the logic that attaches programs to menu options.

2. **Add the following `ifxscreen` command to this file:**

   ```
   :ifxscreen:aw:i_cust::x:
   ```

   The `ifxscreen` command takes four arguments. The first represents your module directory less the .4gm extension. The second contains your program directory less the .4gs directory. The third argument (empty in this example) holds command flags. The last field, which is optional, prevents abnormal exits during processing.

3. **Save and quit from `1`.**

4. **Use `vi` to create a second file and name it `2`.**

5. **Add the following `ifxscreen` command to this file:**

   ```
   :ifxscreen:aw:i_cred::x:
   ```

6. **Save and quit from `2`.**

Run Your *Menus* Program Again

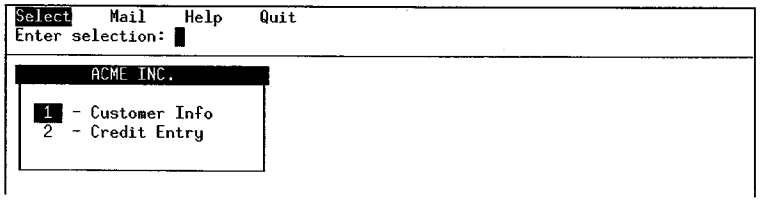
1. **Set `$ifxproject` to point to your application directory:**

   ```
   ifxproject=$HOME/labs ; export ifxproject
   ```

2. **Run your *Menus* program again:**

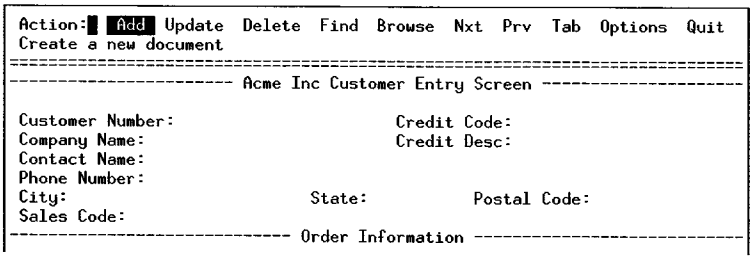
   ```
   mz menu
   ```

   Your *Menus* program appears.



3. **Select item 1.**

   Your Customer Entry program appears.



4. **Quit from Customer Entry and from *Menus*.**

You have now successfully created a *Menus* program. In the next exercise, you will modify this program to include a submenu.

20

Building a Menuing System

Main topics:

- Linking Input Programs to *Menus*
- Setting the `$ifxproject` Variable
- Creating Menu Security Files
- Using *Menus* with Version Control

Linking Input Programs to *Menus*

Menus gives you a special item instruction command (`:ifxscreen:`) to link input programs to *Menus*. You can use this command to associate the programs you build with the Form Painter to a *Menus* program.

The `:ifxscreen:` command uses the following syntax:

`:ifxscreen:`*`module_name`*`:`*`program_name`*`:`*`flags`*`:x:`

In general, to link an input program to a *Menus* program:

1. Build your input program using the Form Painter. Make sure to follow the standard application directory structure (see "The Directory Structure" on page 16-2).

2. Create a *Menus* image file that contains a menu item corresponding to your input program. For example, if your input program is a customer information entry program, let the menu item read "Start Customer Entry" or some other descriptive phrase.

3. Create an item instruction file that corresponds to the position of the menu item on the menu image. For instance, if the "Start Customer Entry" item is the first item on the menu image, create an item instruction file named `1`.

4. Use the `:ifxscreen:` command in the item instruction file. For example, if your program is in `ap.4gm/i_invce.cod`, use the following `:ifxscreen:` command:

   `:ifxscreen:ap:i_invce::x:`

Setting the $ifxproject Variable

The `$ifxproject` variable instructs *Menus* where to look for an input program. For example, consider the following `:ifxscreen:` command:

```
:ifxscreen:ap:i_invce:
```

Menus knows that this program is in `ap.4gm/i_invce.cod`, but *Menus* doesn't know where `ap.4gm` resides. To resolve this problem, the `$ifxproject` variable is set. This variable provides *Menus* with the full directory path to your application directory. For example, if your application directory is `$fg/accounting`, the `$ifxproject` variable should be set to:

```
ifxproject=$fg/accounting ; export ifxproject
```

Other Useful *Menus* Variables

Besides `$ifxproject`, *Menus* uses other useful variables. The following list contains some of these variables and a short description of each:

hot	Determines how many keystrokes the user must enter to uniquely identify a menu. Typically set to 1 or 2.
ifx	Contains the full path name of the last program run.
ifxdebug	Allows you to run Informix programs using the Informix debugger.
md	Contains the name of the menu directory plus the full pathname of the selected menu item.
company	Holds the database name.

Creating Menu Security Files

You can also assign security restrictions to menu items. For example, if you want to assign user restrictions to menu item number one, you can create a `1.prm` file. Inside this file, use the `:deny:` and `:allow:` commands to set security values. These commands use the following syntax:

```
:allow:user_id,...:
:deny:user_id,...:
```

If you want to set security on a group of users, use:

```
:allow:group_name,...:group:
:deny:group_name,...:group:
```

For example, if you want to deny two groups (sales and managers) access to a menu item, type:

```
:deny:sales,managers:group:
```

You can also assign a password to a menu item. Unlike the `:allow:` and `:deny:` commands (which go into *.prm files), the `:password:` command goes into an item instruction file.

The `:password:` command uses the following syntax:

```
:password:literal_password:[p:]
```

Where *literal_password* is the actual password value. For example, if you want failsafe to be your password, type

```
:password:failsafe:p:
```

Once you save this item instruction file, the password line gets encrypted.

The `:p:` argument logs failed password attempts into an administrative file that you can review. Use the `$passfail` variable to point to this log file.

Using *Menus* with Version Control

In addition to running the base version of an input program, you can also instruct *Menus* to run a version control version (see "Version Control Overview" on page 16-3).

Recall that the `:ifxscreen:` item instruction command doesn't require the module and program directory extensions, *.4gm and *.4gs/*.cod respectively. To run a program in `ap.4gm/i_invce.cod`, you specify:

```
:ifxscreen:ap:i_invce::x:
```

You do not include the directory extensions.

When *Menus* encounters an `:ifxscreen:` item instruction, it looks in the base program *.4gs/*.cod directory to find the input program.

If you want *Menus* to look in a version control directory, you need to set the `$cust_path` variable to point to that directory. For example, if you want to run a custom version of enterprise `i_invce` program located in the `i_invce.abc` directory, set `$cust_path` as follows:

```
cust_path=abc:cod ; export cust_key
```

Section Summary

- *Menus* gives you a special item instruction command (`:ifxscreen:`) to link input programs to *Menus*. You can use this command to associate the programs you build with the Form Painter to a *Menus* program.
- The `$ifxproject` variable instructs *Menus* where to look for an input program (i.e., it points to the program's application directory).
- You can assign security restrictions to menu items. For example, if you want to assign user restrictions to menu item number one, you can create a `1.prm` file. Inside this file, you can use the `:deny:` and `:allow:` commands to set security values.
- In addition to running the base version of an input program, you can also instruction *Menus* to run a version control version.
- If you want *Menus* to look in a version control directory, you need to set the `$cust_path` variable to point to that directory.

Exercise 20A

Objective: To create a submenu from your original menu.

Although the programs you have built do not require it, *Menus* gives you the ability to create "submenus." There are three main steps for building submenus:

1. Create a submenu directory.
2. Create the submenu image file.
3. Create the logic to open the submenu.

Create a Submenu Directory

If you created the *Menus* directory structures shown in Exercise 19, you have already created a submenu directory, which you named `progmenu`. If you haven't, see "Create a Menus Directory" on page 19-10 and create that directory structure on your own system (including the `progmenu` directory).

Create the Submenu Image File

Instead of starting from scratch, copy your original `image.img` file into your `progmenu` directory. It will now become your submenu image file.

1. **From your menueze directory, type:**

   ```
   cp main/menu progmenu/menu
   ```

   At this point, you should have two `menu` files, one in your `main` directory and the other in your `progmenu` directory.

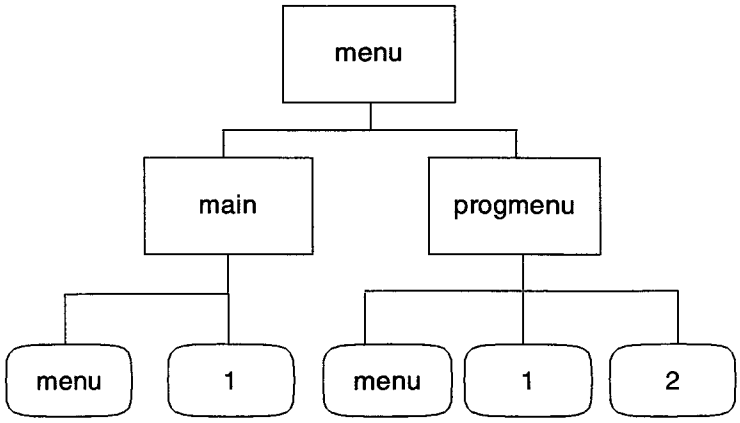
2. **In addition, copy over both of your item instruction files:**

   ```
   cp main/1 progmenu/1
   cp main/2 progmenu/2
   ```

3. **Finally, delete file `2` in the `main` directory:**

   ```
   rm main/2
   ```

Your `menu` directory should now contain the following files:



Note that both `menu` files and `1` files are identical. In most cases, it is simpler to modify image and item instruction files than it is to create new ones.

Create the Logic to Open the Submenu

1. **Use `vi` to edit the menu file in the `main` directory. Make it look as follows:**

```
@         ACME INC.           @
~|                            |~
~|% 1 %- Input Programs       |~
~|                            |~
~+----------------------------+~
```

2. **Save and quit `menu`.**

3. **Use `vi` again to replace the `ifxscreen` line in `main/1` as follows:**

   `:submenu:menu:progmenu`

4. **Save and quit `1`.**

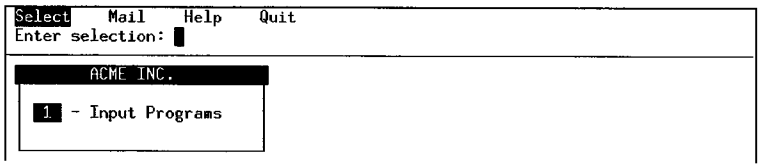
In affect, you have moved your original menu to a submenu and created a "higher-order" menu above it. As it was stated earlier, your sample programs do not require such an elaborate structure. This exercise is just intended to show you how to create sub-menus.

Run Your *Menus* Program

1. **From the UNIX prompt, type:**

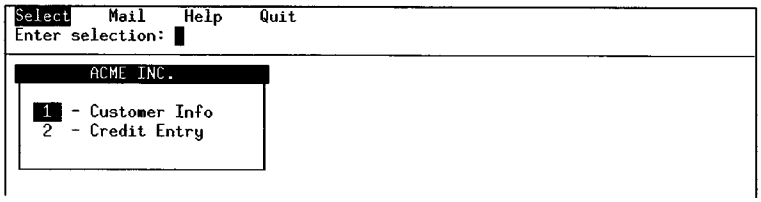
   `mz menu`

   Notice that mz command remains unchanged, even though you've added a submenu. Your *Menus* program appears and the Input Programs menu appears first:.



2. **Select item 1 (the only item available).**

   Your Customer Entry/Credit Entry menu appears:



3. **To start your Customer Entry program, select item 1. To select your Credit Entry program, select item 2.**
4. **Quit out of your input program to return to the *Menus* screen.**
5. **Press [DEL] to return to the Input Programs menu and Q to exit the application.**

Exercise 20B

Objective: To add a password to your Customer Entry menu item.

Menus lets you apply password protection on each menu item.

Create a password Instruction

1. **In your `progmenu` directory, use `vi` to open file `1`.**
2. **Add the following `password` command *above* the `ifxscreen` line:**

   ```
   :password:eatBreakfast::
   ```

3. **Save and quit file `1`.**

Run Your *Menus* Program

1. **Run your *Menus* program:**

   ```
   mz menu
   ```

2. **Select Input Programs from the first menu and the Customer Entry.**

   Before the Customer Entry program loads, a password prompt appears:

3. **Enter `eatBreakfast` as the password.**

   If you make a mistake typing in `eatBreakfast`, don't worry, *Menus* gives you three chances to type in a password correctly.

   Once you enter the password, your Customer Entry program starts.

Check Your Password

1. **Quit out of both your Customer Entry and *Menus* program.**
2. **Return to your `progmenu` directory and use `vi` to open file `1`.**

   What does the password command look like now? Instead of seeing the word eatBreakfast, an encrypted password appears. Encryption takes place once you run your *Menus* program:

   ```
   :password:+!1aX2/!n1~%1rX{:
   ```

3. **If you feel truly inspired, you can password protect the other items on your menu.**

21

Security

Main topics:

- Security Overview
- The Security Programs

Security Overview

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

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

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

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

Security Programs

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

Program Name	Description
Module and Program Information	This program lists the modules and programs on your system. By default, this information comes pre-loaded in Security.
Security Events	This program lists the events used by the modules and programs on your system. Like modules and programs, event information is pre-loaded.
Security Groups	This program lets you define which individual users belong to which user group.
User & Group Permissions	This program provides a complete method for identifying users and groups on your system. In addition, it links information in the Module and Event programs with user and group definitions, and it allows you to set explicit user and group permissions. Most of the work you do with Security is done in this program.
Group Security Control	This program provides an easy-to-use interface for setting up group permissions on events. It does not contain all the features and flexibility of the User & Group Permissions program, but it is a simplistic alternative.

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

Determining Precedence

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

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

User Level Search Order

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

Application Level Search Order

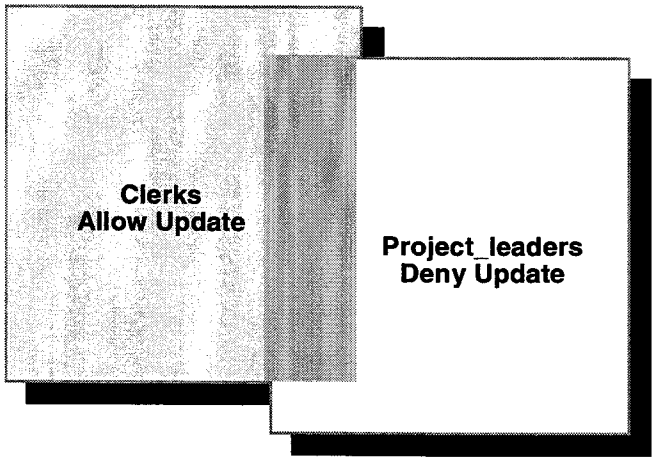
Overlapping Group Permissions

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

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

You can place users into more than one group.

If a user is in two groups that have conflicting security permissions, allow permission is granted.



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

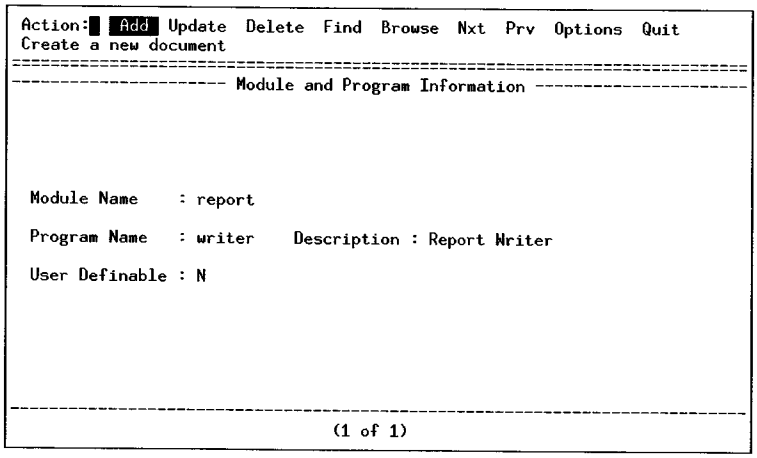
The Security Programs

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

Module and Program Information

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

The Module and Program Information Program.



Adding Custom Programs to Module and Program Information

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

To add a custom report to Module and Program Information:

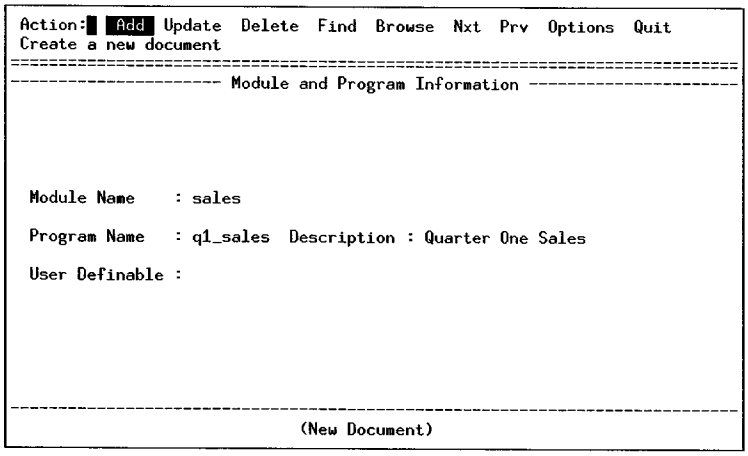
1. **Select Add from the ring menu.**
2. **In the Module Name field, enter the module directory of the custom program.**

   For example, if your custom report is in `sales.4gm`, enter `sales` in the Module Name field.
3. **In the Program Name field, enter the program directory that contains your custom report.**

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

   The User Definable field is a non-entry field.
5. **Press [ESC] to store your entry.**

The Module and Program Information program lets you make custom programs "eligible" to secure.

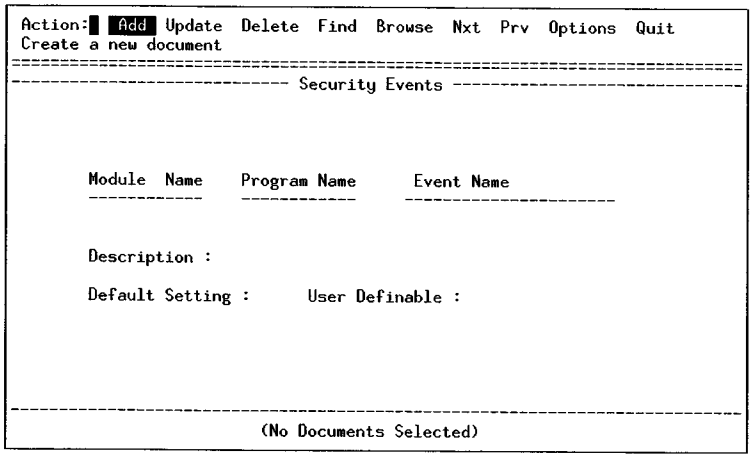


Security Events

This input program is similar to Module and Program Information. It too comes pre-loaded with events used in FourGen programs, such as add, delete, and update. As well, Security Events lets you define cus-

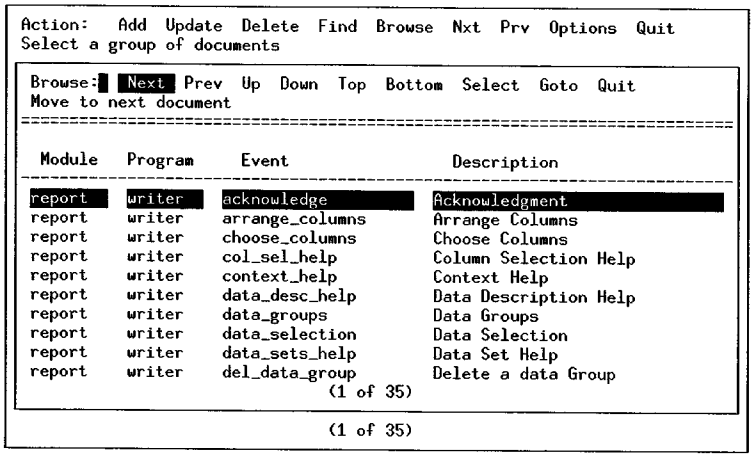
tom events in custom programs. Similar to Module and Program Information, Security Events just lets you define events that are eligible to secure.

The Security Events Program



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

There are 35 events associated with the *Report* Writer.



Adding Custom Events to Security Events

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

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

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

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

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

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

   For example, if the module directory is `sales.4gm`, enter `sales`.

3. **In the Program Name field, enter the program directory of your custom program.**

   For example, if the program directory is `q1_sales.4gs`, enter `q1_sales`.

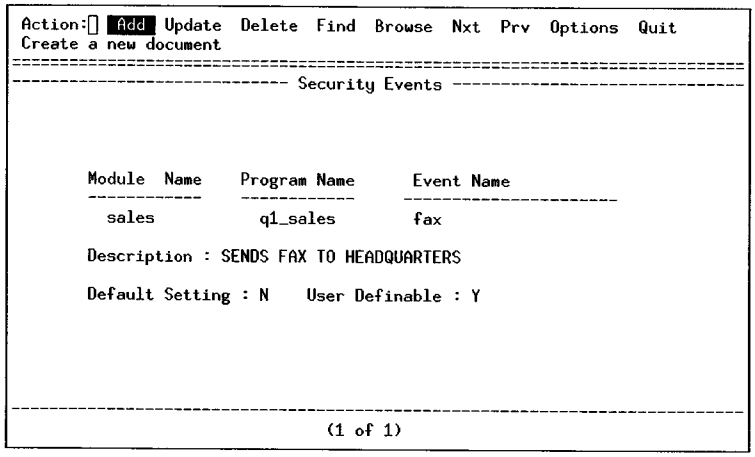
4. **In the Event Name field, enter the name of your custom event.**

   For example, if the event name is `fax`, enter `fax`.

5. **In the Description field, enter a description of your event.**
6. **In the Default Setting field, enter the default permission for the event.**

   The User Definable field is a non-entry field.
7. **Press [ESC] to store your entry.**

Use the Security Events program to make custom events securable.



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

Security Groups

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

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

The Security Groups program lets you define groups of users who share the same permission settings.

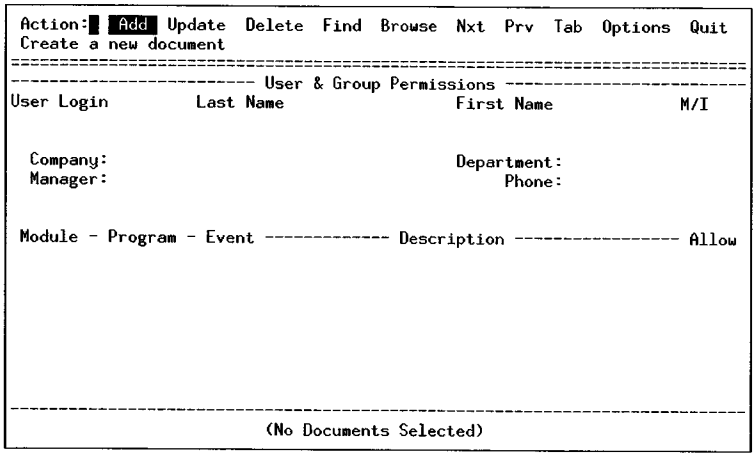
```
Add:    [ESC] to Store, [DEL] to Cancel, [TAB] Next Window      Help:
Enter changes into form                                         [CTRL]-[w]
=============================================================================(Zoom)==
---------------------------- Security Groups -------------------------------

  Group Code  : sales
  Description : SALES PERSONNEL


- User Login --- User Login --- User Login --- User Login --- User Login -
   donw           lynnf          jamesp         thomasr        ralpho
                  ▒▒▒▒▒▒

------------------------------------------------------------------------------
Enter the user login.
```

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

User and Group Permissions

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

The User and Group Permissions program.



Setting Individual User Permissions

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

To set permission for an individual user:

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

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

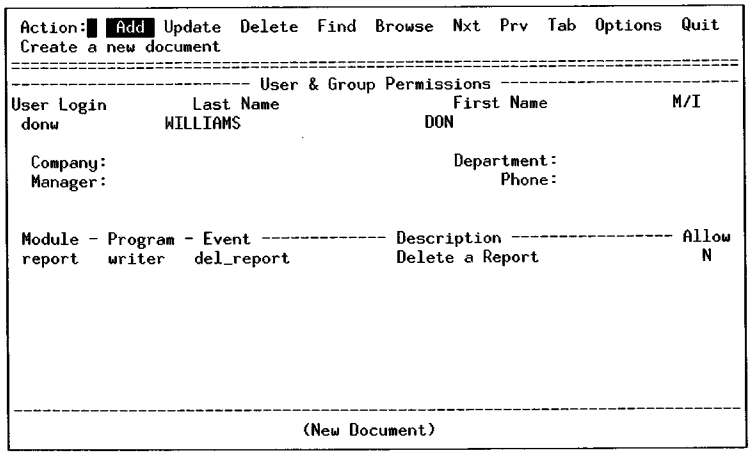
   The User Login and Last Name fields are the only required fields. The other fields in the header section are optional, such as the Department and Phone fields.

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

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

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

This entry denies donw the ability to delete reports.



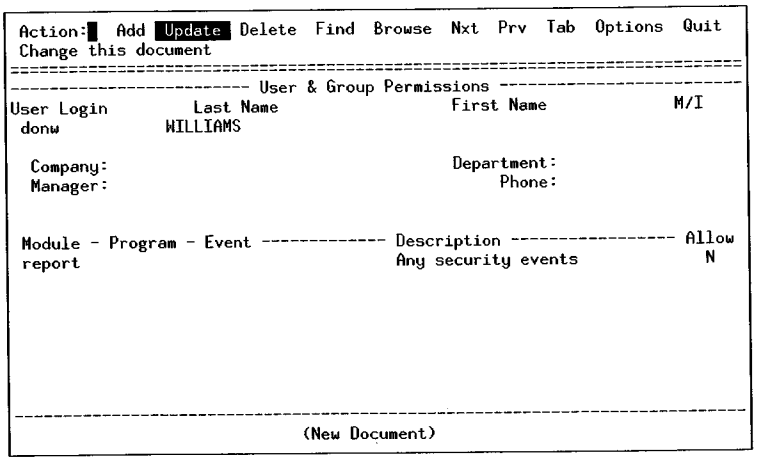
4. **Once you finish entering permission data, press [ESC] to store your entry.**

Setting Permission for an Entire Module

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

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

This entry denies donw access to all the programs in the report module.



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

Setting Group Permissions

You can also set permissions for groups that you have defined in the Security Group program (see "Security Groups" on page 21-10). In the same way you set permissions for individual users, you also set permissions for groups.

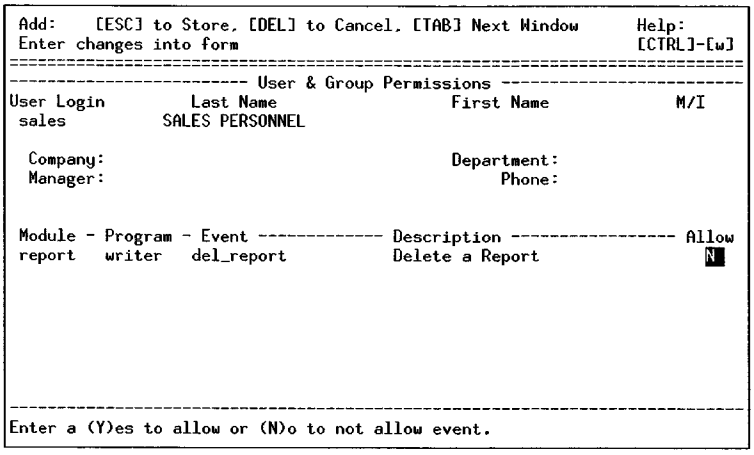
To set permissions for a group:

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

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

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

This entry sets permissions for the sales group.



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

Setting Defaults Permission

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

To set Defaults permission:

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

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

Caution The Defaults permission affects all users on the system.

Group Security Control

Group Security Control is a simplified version of the User and Group Permissions program. With Group Security Control, common program events are already listed. Group Security Control has a matrix type interface, which helps you assign permission settings.

This entry sets permissions for the account group on the report programs.

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

  Group  : account    ACCOUNTANTS GROUP
  Module : report     Report Module

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

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

The following describes the events available in Security Control.

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

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

Section Summary

- Security is based on a hierarchy. You design your security system around three levels of users. In addition, applications are divided into three levels. The key to setting up a quality security system depends on your understanding of these levels and how they relate to each other.
- Security is a collection of five input programs. You use all of these programs to define Security on each level of user and application.
- The Module and Program Information program lets you enter the modules and programs eligible to secure. All modules and programs come pre-loaded. You only need to use Module and Program Information when you create custom programs or modules.
- Security Events is similar to Module and Program Information. It too comes pre-loaded with events used in programs, such as add, delete, and update. As well, Security Events lets you define custom events in custom programs. Similar to Module and Program Information, Security Events just lets you define events that are eligible to secure.
- Security Groups lets you assign individual users to groups. By creating groups of users, from individuals users who require similar system access, you can simplify your security configuration.
- User and Group Permissions is where most of your security work gets done. It is the program that relates the information set in Module and Program Information, Security Events, and Security Groups with actual permission settings.
- Group Security Control is a simplified version of the User and Group Permissions program. With Group Security Control, common program events are already listed. Group Security Control has a matrix type interface, which helps you assign permission settings.

Exercise 21

Objective: To use : Security to deny yourself the ability to update records in your Customer Entry program.

Security lets you control how a program is used and by whom. In this exercise, you will set a security restriction on yourself. You will deny yourself access to the Update ring menu command in your Customer Entry program.

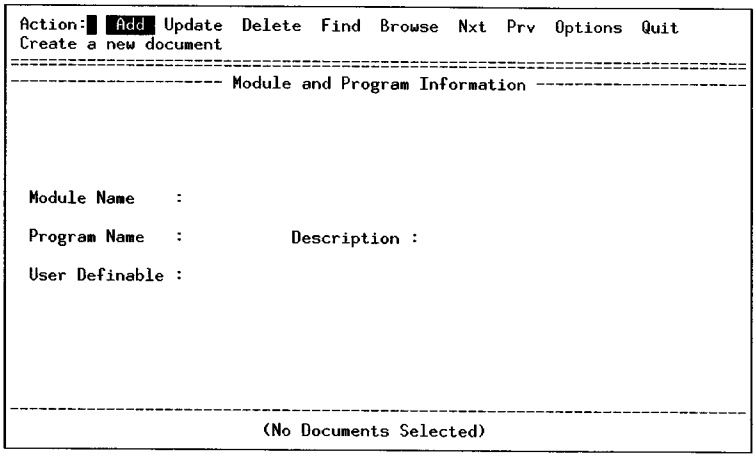
Start the Module Information Program

This program adds your Customer Entry program to a "roster" in the database. The roster is simply a listing of all the modules and programs that are "securable" or eligible to secure.

1. **From the UNIX prompt, type:**

   `fg.modules`

   The Module Information program appears:



2. **Select Add from the ring menu and enter aw in the Module Name field.**

3. **Save this record.**

4. **Select Add again and enter aw in the Module Name field and `i_cust` in the Program Name field.**

5. **Enter a description for `i_cust` then save and quit this program.**

Start the User Permissions Program

User Permissions assigns different security permission values to individual users or groups of users. You will use this program on yourself.

1. **From the UNIX prompt, type:**

   ```
   fg.users
   ```

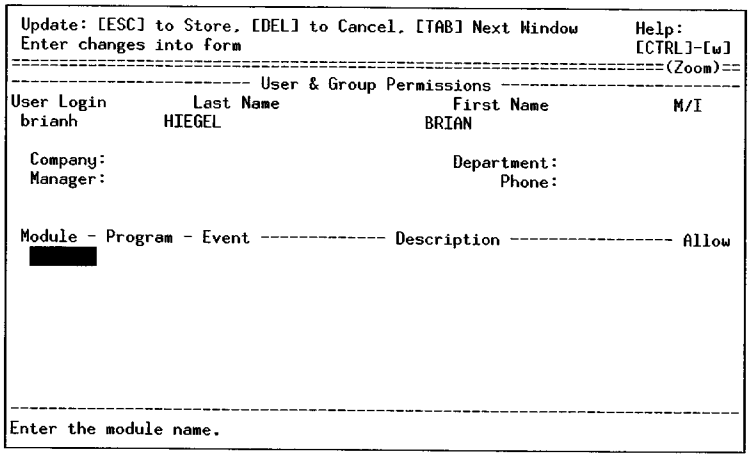
   The User Information program appears. This programcontains both a header and a detail section. The header section contains information about the user, which in this case will be you. The detail section contains information about the module, program, event, and permission setting.

2. **Select Add from the ring menu to create a new user record.**

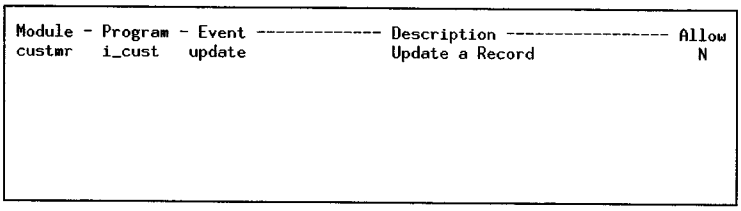
3. **Place your user login in the User Login field.**

4. **Enter your first and last name in the Name fields.**

5. **Press [TAB] to move to the detail section:**



6. **Fill in the detail fields as follows and press [ESC] to save this record:**



7. **Quit from User Permissions.**

Start Your Customer Entry Program

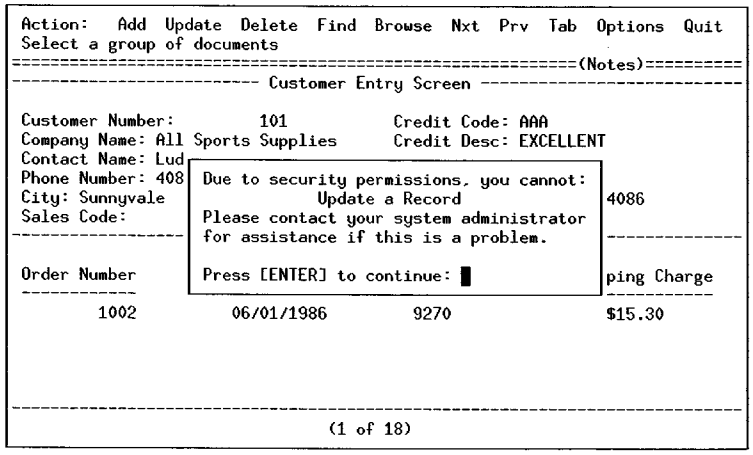
1. **Use `cd` to move to your `i_cust.4gs` directory and start your Customer Entry program.**

   If you would rather, you can also start it from the *Menus* program you created in Exercise 19.

2. **Use Find to select a record or group of records.**

3. **Press Update to alter the record.**

   A message appears denying you access to update:



4. **Press [ENTER] to return to the ring menu.**

   Notice that you can still use the other ring menu commands, you only restricted access to the Update command.

5. **Quit from Customer Entry.**