

InterBase Forms Guide

Disclaimer

Borland International, Inc. (henceforth, Borland) reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should, in all cases, consult Borland to determine whether or not any such changes have been made.

The terms and conditions governing the licensing of InterBase software consist solely of those set forth in the written contracts between Borland and its customers. No representation or other affirmation of fact contained in this publication including, but not limited to, statements regarding capacity, response-time performance, suitability for use, or performance of products described herein shall be deemed to be a warranty by Borland for any purpose, or give rise to any liability by Borland whatsoever.

In no event shall Borland be liable for any incidental, indirect, special, or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Borland has been advised, knew, or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Borland.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

© **Copyright 1993** by Borland International, Inc. All Rights Reserved. InterBase, GDML, and Pictor are trademarks of Borland International, Inc. All other trademarks are the property of their respective owners.

Corporate Headquarters: Borland International Inc., 100 Borland Way, P. O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom.

Software Version: V3.0

Current Printing: October 1993

Documentation Version: v3.0.1

Reprint note

This documentation is a reprint of InterBase V3.0 documentation. It contains most of the information from *InterBase Previous Versions Documentation Corrections* and *InterBase Version 3.2 Documentation Corrections* and a new index. For information on features added since InterBase Version V3.0, consult the appropriate release notes.

Table Of Contents

Preface

Who Should Read This.	ix
Using this Book	x
Text Conventions	xi
Syntax Conventions	xii
InterBase Documentation	xiii

1 Introduction

Overview.	1-1
The Forms System	1-3
The Forms Editor	1-4
How to Approach Forms	1-5
The Sample Database	1-6
Accessing the Sample Database	1-6
Invoking Fred	1-7
Fred's Top-level Menu Options	1-8
About the Examples in this Manual	1-9
For More Information	1-10

2 Creating a Form

Overview.	2-1
Creating a Form	2-1
Navigating in a Form.	2-4
Editing Fields.	2-5
The Edit Type Tag Line	2-7
The Reformat Option.	2-7
The Size Option	2-7

The Exit Option	2-8
Saving the Example Form	2-10
For More Information	2-11
3 Editing a Form	
Overview	3-1
The Edit Tag Line	3-3
The Select Option	3-4
The Move Option	3-4
The Add Option	3-6
The Change Option	3-11
The Reverse Option	3-12
The Delete Option	3-13
The Save Option	3-14
The Exit Option	3-15
For More Information	3-16
4 Editing Subforms	
Overview	4-1
Selecting a Subform	4-3
The Change Option	4-4
The Characteristics Option	4-4
The Region Option	4-4
The Sub_item Option	4-5
The Size Option	4-6
The Exit Option	4-6
For More Information	4-7
5 The Ski Directory Example: A Fred Tutorial	
Overview	5-1
Starting the Tutorial	5-2
The Application Forms	5-3
The NEW_SKI_AREA Form	5-3
The SKI_TRAILS Form	5-5

The NE_SKI_DIR Form	5-7
Creating the SKI_TRAILS Subform	5-9
Completing the fred Session	5-11
Using the Forms in an Application	5-12
For More Information	5-13
6 Using Forms with Qli	
Overview	6-1
Using Forms in Qli	6-2
Invoking Forms in Qli	6-3
Invoking Forms Automatically	6-3
Invoking Forms Explicitly	6-4
Displaying Limited Fields in a Form	6-5
Using Forms to Enter and Modify Data	6-5
Formatting a Form in Qli	6-7
For More Information	6-8
7 Using Forms with GDML	
Overview	7-1
Displaying a Form	7-2
Creating a Window	7-4
Deleting a Window	7-5
Using Attributes	7-6
The .State Attribute	7-6
The .Terminator Attribute	7-8
The .Terminating_field Attribute	7-9
Creating Menus	7-10
Defining Static Menus	7-10
Defining Dynamic Menus	7-12
Using Subforms in GDML	7-13
The New England Ski Directory Application	7-13
Error Handling	7-17
For More Information	7-18

8 Using Blobs with Forms

9 Forms Reference

Overview 9-1
Case_Menu Statement 9-2
Display 9-5
For Form 9-9
For_Item 9-12
For_Menu 9-14
Put_Item 9-17

A Platform Specific Notes

Apollo Notes A-1
 Mouse Support A-1
 Editing Keys A-2
Sun Notes A-4
Keyboard Diagrams A-5

B The Atlas Database

About the Atlas Database B-1

C Sample Forms Programs

Overview C-1
Sample Program 1 C-2
Sample Program 2 C-5

Preface

This book contains information on the InterBase forms facility.

Who Should Read This

The audience for this book is anyone who wants to create an end-user interface that uses forms for data display or input.

This book assumes that you have read the *Getting Started with InterBase* book provided with your documentation set.

Using this Book

This manual is organized in the following way:

Chapter One	Introduces forms and the forms editor, fred .
Chapter Two	Describes creating forms based on existing relations using fred .
Chapter Three	Describes editing an existing form using fred .
Chapter Four	Describes how to select and edit subforms in fred .
Chapter Five	Presents a hands-on tutorial for creating the forms for a complete application.
Chapter Six	Discusses using forms in qli to store, retrieve, and modify data interactively.
Chapter Seven	Discusses manipulating forms using GDML statements.
Chapter Eight	Describes using blobs with forms.
Chapter Nine	Provides syntax and usage descriptions for all forms-specific GDML statements.
Appendix A	Describes platform-specific implementation notes.
Appendix B	Describes the sample database provided with Inter-Base.
Appendix C	Provides two additional sample programs that use GDML statements to manipulate forms.
Index	

Text Conventions

The following section explains how to interpret special type treatments within the text:

boldface	<p>Indicates a command, option, statement, or utility. For example:</p> <ul style="list-style-type: none"> • Use the commit command to save your changes. • Use the sort option to specify record return order. • The case_menu statement displays a menu in the forms window. • Use gdef to extract a data definition.
<i>italic</i>	<p>Indicates chapter and manual titles; identifies file-names and pathnames. Also used for emphasis, or to introduce new terms. For example:</p> <ul style="list-style-type: none"> • See the introduction to SQL in the <i>Programmer's Guide</i>. • <i>/usr/interbase/lock_header</i> • Subscripts in RSE references <i>must</i> be closed by parentheses and separated by commas. • C permits only <i>zero-based</i> array subscript references.
fixed width font	<p>Indicates user-supplied values and example code:</p> <ul style="list-style-type: none"> • <code>\$run sys\$system:iscinstall</code> • <code>add field population_1950 long</code>
UPPERCASE	<p>Indicates relation names and field names:</p> <ul style="list-style-type: none"> • Secure the RDB\$SECURITY_CLASSES system relation. • Define a missing value of X for the LATITUDE_COMPASS field.

Syntax Conventions

This book uses the following syntax conventions:

<code>{braces}</code>	Indicates an alternative item: <ul style="list-style-type: none">• <code>option ::= {vertical horizontal transparent}</code>
<code>[brackets]</code>	Indicates an optional item: <ul style="list-style-type: none">• <code>dbfield-expression[not]missing</code>
<code>fixed width</code>	Indicates user-supplied values and example code: <ul style="list-style-type: none">• <code>\$run sys\$system:iscinstall</code>• <code>add field population_1950 long</code>
<code>commalist</code>	Indicates that preceding word can be repeated to create an expression of one or more words, with each word pair separated by one comma and one or more spaces. For example, <code>field_def-commalist</code> resolves to: <code>field_def[,field_def[,field_def]...]</code>
<i>italics</i>	Indicates syntax variable: <ul style="list-style-type: none">• <code>create_blob blob-variable in dbfield-expression</code>
	Separates items in a list of choices.
↓	Indicates that parts of a program or statement have been omitted.

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

Getting Started with InterBase (INT0032WW2179A) provides an overview of InterBase components and interfaces.

Database Operations (INT0032WW2178D) describes how to use InterBase utilities to maintain databases.

Data Definition Guide (INT0032WW2178F) describes how to create and modify InterBase databases.

DDL Reference (INT0032WW2178E) describes the function and syntax for each of the data definition language clauses and statements. It also lists the standard error messages for **gdef**.

DSQL Programmer's Guide (INT0032WW2179C) describes how to program with DSQL, a capability for accepting or generating SQL statements at runtime.

Forms Guide (INT0032WW2178A) describes how to create forms using the InterBase forms editor, **fred**, and how to use forms in **qli** and GDML applications.

Programmer's Guide (INT0032WW2178I) describes how to program with GDML, a relational data manipulation language, and SQL, an industry standard language.

Programmer's Reference (INT0032WW2178H) describes the function and syntax for each of the GDML and InterBase supported SQL clauses and statements. It also lists the standard error messages for **gpre**.

Qli Guide (INT0032WW2178C) describes the use of **qli**, the InterBase query language interpreter that allows you to read to and write from the database using interactive GDML or SQL statements.

Qli Reference (INT0032WW2178B) describes the function and syntax for each of the data definition, GDML, and SQL clauses and statements that you can use in **qli**.

Sample Programs (INT0032WW2178G) contains sample programs that show the use of InterBase features.

Master Index (INT0032WW2179B) contains index entries for the entire InterBase Version 3.0 documentation set.

In addition, platform-specific installation instructions are available for all supported platforms.

Chapter 1

Introduction

This chapter introduces InterBase forms and the interfaces you use to display and manipulate forms.

Overview

A successful database depends on the effective collection and retrieval of data. To simplify both processes, InterBase provides *forms*, screen images used for the collection and display of data.

The InterBase forms facility provides you with the tools for creating an end-user interface to a database application. You can create forms that enable a user to view or enter information from a database. The forms facility is comprised of:

- A forms editor (**fred**) that provides menu support for building forms.
- **qli** statements for manipulating existing forms and for using default forms.
- GDML statements for incorporating and manipulating forms and menus in GDML applications.

Overview

A *form* is a fill-in template for data. A form usually corresponds to a database relation. Elements of a form are:

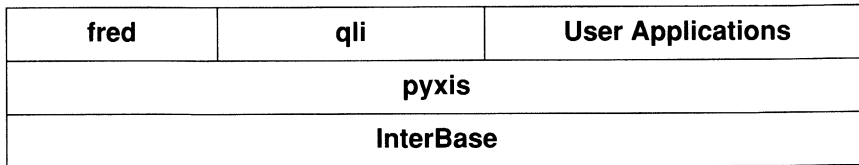
- Text
- Labels
- Fields
- Subforms

Form elements are described in the next chapter, *Creating a Form*.

The Forms System

InterBase forms are built with a software layer named **pyxis**. **Pyxis** is used in turn to create the interfaces to **fred** (the forms editor), **qli**, and end-user applications created using GDML. The relationship between these components is illustrated in Figure 1-1.

Figure 1-1. Forms System Architecture



The following table describes the different ways of creating and using forms in an application.

Table 1-1. Facilities for Using Forms

With this facility...	You can ...
fred	Create, edit, and delete forms and sub-forms with menu support. You cannot store database values.
qli	Store and retrieve database values. Does not provide menu support.
GDML	Create form-based applications. Edit, delete, and store values in pre-defined forms.

The Forms Editor

The InterBase forms editor, known as **fred**, provides an interactive way to define forms. It is characterized by the:

- Use of menus. Rather than requiring you to learn a forms definition language, **fred** provides menus that let you generate a new form or revise an existing form.
- Automatic generation of forms. A single menu choice generates a new form.
- Ease of editing forms to your application's requirements. **fred** lets you add fields to a form, move field labels and data input areas, and change the appearance of labels and input areas.
- Ease of generating forms that reference multiple relations. Menu choices let you choose fields from another relation to include in the form.
- Storage of forms in the same database as the relations they reference. When you finish creating or editing a form, **fred** automatically stores the form in the database (or discards it if you so desire).

Note

You cannot store data through **fred**. To enter values using a form you must use **qli**, GDML or SQL.

The next chapter describes creating a form using **fred**.

How to Approach Forms

This manual provides several ways of learning forms and the **fred** interface. We suggest one of the following:

- *Learn by doing.* Turn to Chapter 5, *The Ski Directory Example: A Fred Tutorial*, and get to know **fred** by building forms following step-by-step instructions.
- *Learn by example.* Read through the chapters sequentially to get a complete look at **fred** and forms.

The Sample Database

Interactive examples are used extensively in this manual to illustrate forms concepts. The examples all refer to data stored in the InterBase sample database, *atlas.gdb*. For information on *atlas.gdb*, refer to Appendix B, *The Atlas Database*.

Accessing the Sample Database

If you plan to try any of the examples in this manual, you must use the *atlas.gdb* database. Copy the database to a local directory so that you can make changes without affecting the original sample database. The location of the sample database file depends on the operating system you are using:

- VMS systems. Copy the sample database from *interbase\$ivp* to a file in your directory:

```
$ copy interbase$ivp:atlas.gdb atlas.gdb
```

- UNIX systems. Copy the sample database from */usr/interbase/examples/atlas.gdb* to a file in your directory:

```
% cp /usr/interbase/examples/atlas.gdb atlas.gdb
```

- Apollo AEGIS systems. Copy the sample database from */interbase/examples/atlas.gdb* to a file in your directory:

```
% cpf /interbase/examples/atlas.gdb atlas.gdb
```

Invoking Fred

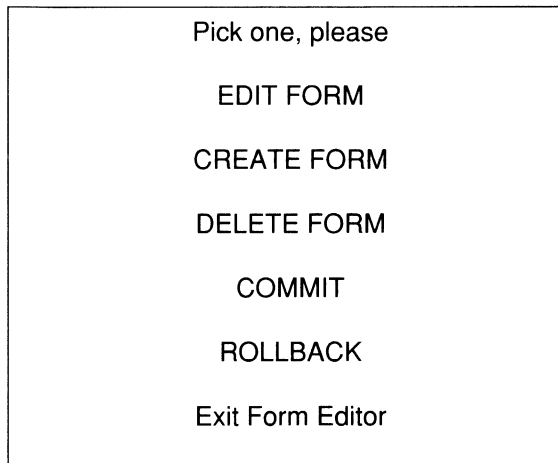
To invoke **fred**, type the editor name, **fred**, followed by the name of a database:

```
% fred atlas.gdb
```

The database name can be specified as a full or relative pathname if it is not in the current directory when you invoke **fred**.

Fred puts up a menu titled " Pick one, please" that lists the top level options:

Figure 1-2. The "Pick one, please" Menu.



```
Pick one, please
EDIT FORM
CREATE FORM
DELETE FORM
COMMIT
ROLLBACK
Exit Form Editor
```

Fred's Top-level Menu Options

The **edit form** option lets you edit an existing form. You can add or delete fields, reformat the form, or save it as a new form. Editing a form is described in Chapter 3.

The **create form** option lets you create a new form, possibly based on an existing relation. Creating a form is described in Chapter 2.

If you select **delete form** from the menu, **fred** pops up a menu listing all of the forms in the current database. Select the name of the form you want to delete using the arrow keys, and press Enter. The form is deleted from the database.

The **commit** option writes all operations since the last commit or rollback to the database. Operations you perform in **fred** are not entered in the database until they are committed.

For example, if you use the **delete form** option to remove a form, the form will no longer be displayed in the **fred** menu that lists forms, but it still exists in the database. If you commit the changes the form is deleted.

The **rollback** option lets you undo changes to the database if they have not yet been committed. For example, if you remove a form using the delete form option, the form will no longer be displayed in the **fred** menu that lists forms, but it still exists in the database. If you roll back the changes the form is restored.

About the Examples in this Manual

This guide uses interactive examples to illustrate concepts. Chapter 5 is a complete tutorial providing instructions for building the forms for a New England Ski Directory application. This application uses forms to display and accept data for ski areas in the New England states. The application code is presented in Chapter 6.

All of the examples in this manual use data from the *atlas.gdb* database supplied with your software.

For More Information

For More Information

Refer to the *Qli Reference* for the syntax for:

- **commit**
- **rollback**

Chapter 2

Creating a Form

This chapter describes how to create forms based on relations, and how to navigate in a form.

Overview

This chapter describes how to create a form based on a relation in the *atlas.gdb* database. Most concepts in this chapter are illustrated using an example.

The forms examples demonstrate creating a form based on the `SKI_AREAS` relation.

Creating a Form

To create a form:

1. Invoke **fred** for the atlas database if you have not already done so by typing:

```
% fred atlas.gdb
```
2. Select **create form** from the “Pick One, Please” menu.

Fred now displays the “Select Relation” menu shown partially in Figure 2-1. This menu lists all the relations from the *atlas.gdb* database you specified when you invoked **fred**.

3. Select the SKI_AREAS relation from the list using the arrow keys to move the selection box, and press the Return key.

Figure 2-1. Select Relation Menu

```

BASEBALL_TEAMS
CITIES
CROSS_COUNTRY
MAYORS
POPULATIONS
PROVINCES
RIVERS
RIVER_STATES
SKI_AREAS
STATES
TOURISM
    
```

Once you have chosen a relation and pressed the Return key, **fred** paints a *default form* on the screen. A default form displays field names and input areas that correspond to the field names of the source relation, and the top level tag line. Figure 2-2 shows the default form for the SKI_AREAS relation.

Figure 2-2. Elements in a Default Form

```

NAME      XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Label — TYPE
CITY      XXXXXXXXXXXXXXXXXXXXXXXXXXXX
STATE     XXXX ————— Input area

Edit type: EDIT REFORMAT SIZE Exit —Tag line
    
```

Default Form Elements

In the default form, **fred** automatically defines form fields to match the characteristics of the database fields for the relation. The form fields consist of two parts:

1. A *label*, such as “STATE.” The label matches the database field name.
2. An *input area*, such as “XXXX.” The input area reflects the datatype of the database field.

fred treats the label and the input area as separate objects. For example, if you choose the “STATE” field label as the object you want to manipulate, the input area “XXXX” that follows is not be affected by operations on the “STATE” label.

Input Area

The input area accepts a field value in accordance with the datatype specified for the field. Valid datatype representations in **fred** are:

- **X** for alphanumeric data
- **9** for fixed integer data, with or without scale
- **F** for single or double floating data
- **D** for date data
- An outlined box for blob data

Datatypes are fully described in the chapter on defining fields in the *Data Definition Guide*.

Changing the forms and appearance of field labels and input areas is discussed in the next chapter, *Editing a Form*.

The Tag Line

At the bottom of the form is the *tag line*. The tag line lists options for editing and manipulating the form. The tag line is updated to display operations you can perform on the form. The tag line options are discussed later in this chapter.

Navigating in a Form

When defining a form using **fred**, you can be in one of two cursor modes:

- Navigation mode
- Edit mode

Navigation mode enables you to move from field to field within a form.

Edit mode enables you to modify the editable contents of a field within a form.

When you enter the forms editor you are by default in navigation mode. The arrow, tab, delete, and return keys move the cursor from field to field within a form.

Note

If you are using InterBase on an Apollo, you can navigate in a form using a mouse. Using a mouse with forms is described in Appendix A, *Apollo Specific Notes*.

Practice using the cursor movement keys to move around the SKI_AREAS form.

Editing Fields

To enter edit mode while navigating a form, move to a field and press an edit key. The edit keys are marked in bold typeface in the following table. The other keys control cursor movement and edit functions within a field.

Figure 2-3. Edit Keys Table

Function	Description	Key on Apollo	Key on all other platforms
Edit	Toggles between edit mode and navigation modes	EDIT	Ctrl-G
Insert/overstrike	Toggles between insert and overstrike modes	INS	Ctrl-A
Erase	Deletes the contents of entire field	LINE DEL	Ctrl-U
Insert	Inserts any printable character into field	Any char	Any char
Right	Moves cursor one character to right	Right arrow	Right arrow
Left	Moves cursor one character to left	Left arrow	Left arrow
Delete	Deletes character to left of cursor	BACKSPACE	Delete
Delete next character	Deletes the current character	CHAR DEL	Ctrl-F
Go to start	Moves cursor to start of field	Left Bar Arrow	Ctrl-H
Go to end	Moves cursor to end of field	Right Bar Arrow	Ctrl-E

Once a field is selected for editing, if the first key you press is the erase key or the insert key (any printable character) the contents of the field are deleted and new characters are inserted. However, if the first key you press is the edit key or the insert/overstrike

Editing Fields

key, the existing contents remain and new characters are inserted. If you try typing characters into a field that is full and cannot accept more characters, the bell rings to warn you that characters will be lost.

The Edit Type Tag Line

When you open a form in **fred**, whether it is newly created or opened for editing, your first options for altering the form are presented in the Edit type tag line. The top level options on the tag line are:

- Edit
- Reformat
- Size
- Exit

The **edit** option is discussed in the next chapter.

The Reformat Option

The **reformat** option is only useful if you are editing a form, not creating a new form. Reformatting restores the default characteristics of that form.

Caution

Take caution when using this option; it undoes all of your formatting changes and reformats the form so that it looks like a default form. A default form has vertical orientation, right justification for labels, and left justification for input areas. To avoid losing work if you reformat accidentally, save your work frequently. Saving a form is described in the section *The Exit Option*.

The Size Option

The **size** option on the Edit type tag line allows you to change the width and height of a form. When you select **size** from the Edit menu, **fred** puts up a subform that displays the current dimensions of the form, as shown in Figure 2-4.

Figure 2-4. Resizing the SKI_AREAS Form

F o r m S i z e

WIDTH HEIGHT

OUTLINE_FORM

The Edit Type Tag Line

To change the width or height of a form, move the cursor to the appropriate box using the TAB or cursor keys. Type in a new value and press Return. To display the form in a box, set the value in the `outline_form` box to Y. Press Enter when you have completed your changes.

If you choose a size that is too small to display the entire form, the form becomes scrollable. If there is one element too wide for the form, **fred** truncates the display.

The Exit Option

If you select **Exit** from the Edit type tag line, the tag line changes to Retention Options and displays the following options:

- Save
- Rename
- Discard
- External file

These options are described below.

The **save** option prepares to write the form to the database. If this is the first time you have saved this form from the top level tag line, you are prompted to supply a form name. The form name can be up to 31 characters and can include any printable character.

If you are editing an existing form, **fred** does not prompt you for a filename, but overwrites the current version of the form stored in the database and returns you to the “Pick one, please” menu.

Note

Changes made in **fred** are not written to the database until you explicitly commit the changes by selecting **commit** from the “Pick one, please” menu, or you exit **fred**, which executes a commit. Even after you save a form, the changes can be undone by selecting the **rollback** command from the “Pick one, please” menu. The commit and rollback operations are described in Chapter 1.

Renaming a Form

The **rename** option is only available if you are editing a form, not creating a new form. If you do not want to overwrite the original form, use the **rename** option to assign a new name to a form. You can then save your changes to this new form.

Discarding a Form

If you do not want to save the new form you have created, or the changes you have made to an existing form, select **discard** from the Retention Options tag line.

If you are in a form and you do not want to make any changes, use the **discard** option to close the form.

Saving a Form to an External File

The **external file** option writes the form to an operating system file that you specify. **fred** prompts you for a filename. Assign a filename or a full pathname, following the conventions for your operating system. Press Enter to assign the filename. **fred** displays the “Pick one, please” menu so that you can commit or rollback your changes, or select another operation.

Saving the Example Form

To save the SKI_AREAS form, follow these steps:

1. Select **exit** from the Edit Type tag line.
2. Select **save** from the Exit tag line. **fred** prompts you to enter a form name.
3. Enter the name SKI_AREAS. **fred** displays the “Pick one, please” menu.
4. Select **commit** from the menu.

The SKI_AREAS form is now stored as part of the atlas database. Chapter 3 describes editing existing forms.

For More Information

For more information about:

- Keyboard commands, refer to Appendix A, *Platform Specific Notes* for machine-specific keyboard variations.
- Datatypes, refer to the chapter on field attributes in the *Data Definition Guide*.
- The *atlas.gdb* database, refer to Appendix B, *The Atlas Database*.

Chapter 3

Editing a Form

This chapter describes how to edit the contents and layout of an existing form.

Overview

The following sections describe customizing an existing form. The example form is a form provided with the atlas database and is based on the STATES relation.

To edit the STATES form:

1. Select **edit form** from the “Pick one, please” menu. **fred** displays the list of existing forms.
2. Select STATES from the list of forms. The STATES form is displayed.
3. The cursor is positioned on the tag line at the bottom of the form. Using the arrow keys, move to the **edit** option on the Edit type tag line.
4. Press Enter to select the **edit** option. **fred** prompts you to select a form element, as shown in Figure 3-1.

Figure 3-1. The STATES Form

```
STATE      XXXX

STATE_NAME  XXXXXXXXXXXXXXXXXXXXXXXX
          AREA  9999999999
STATEHOOD   DDDDDDDDDDD
          CAPITAL XXXXXXXXXXXXXXXXXXXXXXXX

Edit type: EDIT REFORMAT SIZE Exit
```

The Edit Tag Line

All editing operations are performed using options from the tag line displayed at the bottom of the form. The Edit type tag line has these options:

- Edit
- Reformat
- Size
- Exit

To edit a form, use the arrow keys to move the selection box to the Edit option and press Return. Press Enter to select the STATE field. The tag line changes to display the following options for editing a form:

This option...	allows you to...
select	select an object for editing.
move	reposition form elements.
add	add text, fields, or subforms to a form.
change	modify label text or input field characteristics.
delete	delete elements from a form.
reverse	change the display characteristics of input fields and labels.
save	save a form.
exit	exit to the top level tag line.

Each of the editing operations is described in the following sections.

The Select Option

To select elements in the form move the cursor to the **select** option and press Return. You are now in navigation mode, as described in Chapter 2. Move to the STATEHOOD field and press Return to select the field.

Note

Apollo users can use the mouse to select menu options and elements. Other Apollo platform-specific variations are described in Appendix A.

The Move Option

The **move** options enable you to move all, or some, of the elements in your form. Move the cursor to **move** and press Enter. The tag line changes to display options for reordering the form. These options and the operations they perform are listed in the following table.

This option...	allows you to...
item	move the selected field or label.
some	select multiple items by selecting one, pressing Return, and repeating until the desired group is selected. Press Enter when done selecting. (Pressing Return also deselects a selected item.)
most	select many form elements quickly. All elements are selected. To deselect, move cursor to an item and press Return. Repeat until only the group you want to move is selected and then press Enter. (Pressing Return also reselects a deselected item.)
all	move all elements in a form.
exit	return to the Edit tag line.

Once you have selected the item or items you want to move, you can move them in the following ways:

- Enter a number, then press an arrow key to move the object that number of spaces in the direction indicated.
- Press any cursor movement key to move an object or group of objects in the direction indicated. Continue to press the key until the object is positioned where you would like it.

For example, to reformat the STATES form so the STATE label and field are more prominent, do the following:

1. Select **edit** from the tag line.
2. Move the cursor to the STATE label and press Enter.
3. Select **move** from the Edit tag line and press Enter.
4. Select **most** from the Move tag line. All the elements are selected and displayed in reverse video.
5. Move the cursor to the STATE label field and press the Return key. The field is deselected.
6. Move the cursor to the STATE input field and press Return. The input field is deselected.
7. Press the Enter key.
8. Enter 10, then press the right arrow key.

The STATES form is now formatted as shown in Figure 3-2.

Figure 3-2. Moving Fields in the STATES Form

STATE	XXXX
STATE_NAME	XXXXXXXXXXXXXXXXXXXXXXX
AREA	9999999999
STATEHOOD	DDDDDDDDDD
CAPITAL	XXXXXXXXXXXXXXXXXXXXXXX
Edit options:SELECT MOVE ADD CHANGE DELETE	
REVERSE SAVE Exit	

Note

As soon as you press the Enter key, the tag line reverts to the Edit tag line.

The Add Option

The **add** option enables you to add text, a form field that corresponds to a program field, or a database field from another relation. When you select **add** from the Edit tag line, the tag line changes to display the following **add** options:

- Text
- Field
- Database fields
- Repeating sub-form
- Exit

Each of these options is described in this section.

Adding Text to a Form

To add text to a form, select **text** from the Add tag line. The tag line prompts you to:

```
Enter text, terminate with a <cr>
```

For example, to add explanatory text to the STATES form, follow these steps:

1. Select **text** from the Add tag line.
2. Move the cursor to the right of the State input field using the arrow keys to position the cursor.

Note

If you use the space bar to position the cursor, the spaces are included as part of the text you add.

3. **Type:** (Enter 2-letter state code)
4. Press the Return key.

Figure 3-3 displays the updated STATES form.

Figure 3-3. Adding Text to a Form

```

STATE      XXXX (Enter 2-letter state code)

STATE_NAME  XXXXXXXXXXXXXXXXXXXXXXXX
AREA        9999999999
STATEHOOD   DDDDDDDDDDD
CAPITAL     XXXXXXXXXXXXXXXXXXXXXXXX

Edit options:SELECT MOVE ADD CHANGE DELETE REVERSE
SAVE Exit
    
```

Adding External Fields to a Form

The Add menu also enables you to add a field that is not part of the database. For example, suppose you have information about each state stored in an external file; you can add that information to the STATES form using the **field** option from the Add tag line.

When adding a non-database field to a form, you must supply field attributes such as datatype and length. **fred** supplies a form through which you can enter the field characteristics.

For example, to add a GUIDEBOOK field to the STATES form:

1. Select **add** from the Edit tag line.
2. Select **field** from the Add tag line.
3. **fred** displays the field definition form, shown in Figure 3-4.

Figure 3-4. The Field Definition Form

FIELD_NAME	<input type="text"/>		
FILL_STRING	<input type="text"/>		
EDIT_STRING	<input type="text"/>		
UPCASE	<input type="checkbox"/>	DATATYPE	<input type="checkbox"/>
WIDTH	<input type="checkbox"/>	SCALE	<input type="checkbox"/>
ALIGN_RIGHT	<input type="checkbox"/>	LENGTH	<input type="checkbox"/>

You are required to supply a field name and datatype; the remaining fields are optional. Field characteristics are described in the section on field attributes in the *Data Definition Guide*. The following table briefly describes the optional fields.

Table 3-1. Options for Defining Non-Database Fields

This option...	allows you to...								
edit_string	<p>describe the format of the string when it is displayed.</p> <p>For example, an edit string of (xxx)bxxx-xxxx describes how a phone number should be printed, where x represents a numeric value and b represents a space.</p>								
fill_string	specify what characters appear by default in the field when no value is explicitly entered.								
upcase	direct InterBase to translate all input to uppercase. For example, change Ma to MA . The default is to accept input exactly as given.								
width	<p>specify how much of the field should be displayed. Many datatypes, such as long, have default widths.</p> <p>For example, the AREA field in the STATES form can accept 10 numeric characters. Although it is not practical in this example, you could specify the field to display only four characters.</p>								
align_right	<p>right-justify the display of data. This is especially helpful for displaying numeric data. While numeric fields are right-justified by default, you may have numeric data in character fields. For example, suppose you want travel expenses stored in the character form 104.12, 2,193.21, 17.99. Unless you specify align-right they will be left-justified:</p> <table data-bbox="484 1494 852 1622"> <thead> <tr> <th>Default</th> <th>Align-right</th> </tr> </thead> <tbody> <tr> <td>104.12</td> <td>104.12</td> </tr> <tr> <td>2193.21</td> <td>2193.21</td> </tr> <tr> <td>17.99</td> <td>17.99</td> </tr> </tbody> </table>	Default	Align-right	104.12	104.12	2193.21	2193.21	17.99	17.99
Default	Align-right								
104.12	104.12								
2193.21	2193.21								
17.99	17.99								

Table 3-1. Options for Defining Non-Database Fields continued

This option...	allows you to...
datatype, scale, and length	assign datatype attributes for the field. Datatypes, scale and length are described in the <i>Data Definition Guide</i> .

Adding Database Fields to a Form

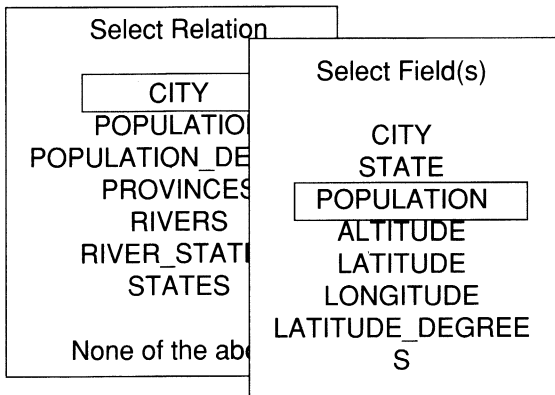
You may want a form to reference other relations involved in a query. You can do this by adding fields from other relations. Use this feature if you plan to use a form to express a join between two or more relations in your database. Expressing joins using forms is discussed in Chapter 6, *Using Forms with Qli*.

To add a field from the CITIES relation to the STATES form, follow these steps:

1. Select **add** from the Edit menu.
2. Select **database fields** from the Add tag line. A menu pops up listing all of the available relations.
3. Select the CITIES relation. A menu pops up listing the fields. See Figure 3-4.
4. Select the POPULATION field by moving the cursor to POPULATION, pressing <CR>, then pressing Enter. This selection sequence makes it possible for you to select multiple fields to add. In this case, however, you simply want to select POPULATION.

Figure 3-5 displays the selected database fields.

Figure 3-5. Selecting a Database Field



The Edit Tag Line

The POPULATION field is added to the bottom of the STATES form. Use the **move item** command to align the POPULATION field with the other fields on the form. Figure 3-6 shows the STATES form with the POPULATION field added.

Figure 3-6. Adding the POPULATION Field to the STATES Form

```
STATE      XXXX (Enter 2-letter state code)
STATE_NAME XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
AREA       9999999999
STATEHOOD  DDDDDDDDDD
CAPITAL    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          POPULATION 9999999999

Edit options:SELECT MOVE ADD CHANGE DELETE REVERSE
              SAVE Exit
```

When you use this form in **qli** or in a program, a value for the POPULATION field does not show up unless the record selection expression includes a join over the STATE field that searches for a city equal to the capital city. For an example of how to refer to the field of a second relation in **qli**, refer to Chapter 6 of this manual.

Adding Repeating Subforms

Forms use *subforms* to deal with repeating groups. Subforms, as the name implies, are forms within a form. They are useful for entering or retrieving data that has a one-to-many relationship.

For example, if you modify the STATES form to include information about ski areas in a specified state, you must either expand the form greatly to accommodate all of the ski area data, or display the ski area data more efficiently, using a subform.

A subform can be scrolled independently within the form, so that you can create a display window showing a specified amount of data at one time.

To create the subform described in the previous paragraph, follow these steps:

1. Select **add** from the Edit tag line.
2. Select **repeating subform** from the Add tag line. A menu pops up listing forms in the database.
3. Select **SKI_AREAS** from the list.

The **SKI_AREAS** subform is added to the STATES form, as shown in Figure 3-7.

Figure 3-7. Adding the SKI_AREAS Subform to the STATES Form

```

STATE          XXXX (Enter Uppercase only)
STATE_NAME    XXXXXXXXXXXXXXXXXXXXXXXX
  AREA        9999999999
STATEHOOD     DDDDDDDDDDD
  CAPITAL     XXXXXXXXXXXXXXXXXXXXXXXX

POPULATION    9999999999
SKI_AREAS

```

```

NAME  XXXXXXXXXXXXXXXXXXXXXXXX
TYPE  X
CITY  XXXXXXXXXXXXXXXXXXXXXXXX
STATE XXXX

```

```

Edit options:SELECT MOVE ADD CHANGE DELETE REVERSE
SAVE Exit

```

Editing subforms is described in the next chapter.

The Change Option

The **change** option lets you change label text, explanatory text, or the characteristics of an input area. The change options depend on what form element is selected. For example, if you select a label or explanatory text, then select **change** from the Edit tag line, you are prompted to:

Enter replacement text, terminate with <CR>

If you select a field, then select **change** from the Edit tag line, a new tag line is displayed with these options:

This option...	allows you to...
characteristics	edit the defining properties of afield, such as the datatype and display width.
enumerations	specify or edit a list of acceptable values for the field.
exit	return to the Edit tag line.

The Edit Tag Line

The **characteristics** menu allows you to edit the properties of a field. To invoke the menu, select a field, then select **characteristics** from the Change tag line. **Fred** displays the menu shown in Figure 3-8.

Figure 3-8. The Characteristics Menu

FIELD_NAME	NAME		
EDIT_STRING	X (25)		
FILL_STRING	XXXXXXXXXXXXXXXXXXXXXXXXXX		
UPCASE	N	DATATYPE	VARYING
WIDTH	25	SCALE	
ALIGN_RIGHT	N	LENGTH	27

The attributes on the Characteristics menu are described earlier in this chapter, in the section *Adding External Fields to a Form*.

The **enumerations** option limits values for a field to those you specify. In order to limit the states to those in New England, you can specify an enumerated list of states that define the valid domain for the STATE field.

To limit the field:

1. Select the STATE field.
2. Select **change** from the Edit option tag line.
3. Select **enumerations** from the Change tag line. **Fred** displays a form in which you can enter the valid values for the field.
4. Enter **CT, NH, MA, ME, RI, and VT** in the **values** input area.
5. Press Enter.

The Reverse Option

The **reverse** option lets you change the display of field labels and input areas. When you select **reverse** from the Edit tag line, the behavior differs, depending on what is selected.

If text is selected, the **reverse** option toggles the video display between white text on a black background and black text on a white background.

If an input area is selected when you choose the **reverse** option, the tag line changes to display the following options:

This option...	allows you to...
invert	toggle the normal display between black text on a white background and white text on a black background.
reverse on update	display updatable fields in reverse video. This is the default behavior.
normal	remove any distinction between updatable and non-updatable input areas.
exit	return to the Edit tag line.

Note

The result of the invert operation is immediately visible to you in **fred**. The **reverse** and **normal** cases are not apparent until you use the form in **qli** or in a program.

The Delete Option

The **delete** option on the Edit tag line enables you to delete elements from a form.

Since the STATES form has several fields that are not relevant to the example, follow these steps to delete extraneous fields:

1. Move the cursor to the AREA label and press Enter.
2. Select **delete** from the tag line. The AREA label disappears.
3. Move the cursor to the AREA field and press Enter.
4. Select **delete** from the tag line.
5. Repeat these steps to delete the STATEHOOD, CAPITAL, and POPULATION labels and fields.

The STATES form should now look as it appears in Figure 3-9.

Figure 3-9. Revising the STATES form

```
STATE   XXXX (Enter 2-letter state code)
STATE_NAME   XXXXXXXXXXXXXXXXXXXXXXXX
SKI_AREAS
  NAME       XXXXXXXXXXXXXXXXXXXXXXXX
  TYPE       X
  CITY       XXXXXXXXXXXXXXXXXXXXXXXX
  STATE      XXXX
Retention options:  SAVE  RENAME  DISCARD
EXTERNAL FILE
```

The Save Option

The **save** option on the Edit tag line allows you to save a form intermittently without going to the top level tag line. When you select the **save** option, **fred** prompts you for a form name. The form name can be up to 31 characters in length.

To assign a name to the form created in this chapter, enter the name `STATE_SKI`.

The Exit Option

When you are done editing a form, select **exit** from the Edit tag line. The tag line changes to the Edit type tag line:

Figure 3-10. Exiting Edit Mode

```

STATE   XXXX (Enter 2-letter state code)

STATE_NAME   XXXXXXXXXXXXXXXXXXXXXXXX

SKI_AREAS
  NAME       XXXXXXXXXXXXXXXXXXXXXXXX
  TYPE       X
  CITY       XXXXXXXXXXXXXXXXXXXXXXXX
  STATE      XXXX

Edit options:SELECT MOVE ADD CHANGE DELETE
REVERSE SAVE Exit
    
```

The exit options are described in Chapter 2, in the section, *The Exit Option*.

The STATE_SKI form is now complete. Follow these steps to exit from the STATE_SKI form:

1. Select **save** from the Exit options tag line.
2. Select **commit** from the "Pick one, please" menu.
3. Select **exit form editor** to end the **fred** session.

For More Information

For More Information

For more information on:

- Writing a record selection expression that joins relations, see the *Qli Guide*.
- Datatypes, refer to the section on field attributes in the *Data Definition Guide*.
- Creating and using external relations, see the *Programmer's Guide*.
- Editing Subforms, see Chapter 4.

Chapter 4

Editing Subforms

This chapter describes editing the content and layout of a subform.

Overview

The following sections describe editing the `SKI_AREAS` subform in the `STATE_SKI` form. Creating the `SKI_AREAS` subform and adding the subform to the `STATE_SKI` form were described in the previous chapter.

To invoke the `STATE_SKI` form, defined in Chapter 3:

1. Move the cursor to the **Edit form** option on the “Pick one, please” menu, and press Enter. **fred** displays a list of existing forms in the active database.
2. Select `STATE_SKI` from the list of relations. **fred** displays the form shown in Figure 4-1.

Figure 4-1 shows the `STATE_SKI` form with the `SKI_AREAS` subform from the Chapter 3 example.

Figure 4-1. STATE_SKI form with SKI_AREAS Subform

```
STATE      XXXX (Enter 2-letter state code)
STATE_NAME  XXXXXXXXXXXXXXXXXXXXXXXX

SKI_AREAS

  NAME      XXXXXXXXXXXXXXXXXXXXXXXX
  TYPE      X
  CITY      XXXXXXXXXXXXXXXXXXXXXXXX
  STATE     XXXX

Edit options:SELECT MOVE ADD CHANGE DELETE
REVERSE SAVE Exit
```

Selecting a Subform

To select the SKI_AREAS subform:

1. Select **edit** from the Edit type tag line.
2. Move the cursor to the subform to the right of the SKI_AREAS label and press Enter. The SKI_AREAS subform is selected.
3. Select **change** from the Edit tag line. The tag line changes to the Change tag line

All changes to the subform are made using the options on the Change tag line.

Note

Making changes to a subform does not affect the original form from which the subform is created. Thus, if you edit the SKI_AREAS subform in the STATE_SKI form, the changes you make are not reflected in the SKI_AREAS form in the database.

If you want to use the same edited subform in more than one form, edit the original subform, delete the subform in the STATE_SKI form, and add the newly modified subform.

The Change Option

The Change options apply to the subform when the subform is selected. They are:

- Characteristics
- Region
- Sub_item
- Size
- Exit

The Characteristics Option

When you select the **characteristics** option from the Change tag line, a menu prompts you to rename the subform. To rename a subform:

- Type a new name and press Enter to change the name, or
- Press Enter to close the menu and keep the existing name

The Region Option

The **region** option lets you change the size of the subform. When you select **region** from the Change tag line, **fred** displays a Form Size menu, as shown in Figure 4-2.

The region determines how many instances of the subform are displayed on the form. For example, if the height of a subform is 6 and you set a region size to have a height of 24, you could have four instances of the subform displayed at once.

Figure 4-2. The Form Size Menu

```

STATE      XXXX (Enter Uppercase only)
STATE_NAME XXXXXXXXXXXXXXXXXXXXXXXXXX
  AREA     9999999999
STATEHOOP  DDDDDDDDDDD
  CAPITAL  F o r m   S i z e
  CAPITOL  WIDTH 34      HEIGHT 15
POPULATION
SKI_AREAS  OUTLINE_FORM  Y
          NAME  XXXXXXXXXXXXXXXXXXXXXXXX
          TYPE  X
          CITY  XXXXXXXXXXXXXXXXXXXXXXXX
          STATE XXXX

Edit options:SELECT MOVE ADD CHANGE DELETE REVERSE
SAVE Exit

```

To change the size of the subform, use the arrow keys to select a value field in the Form Size menu and change the value. Press Enter to close the menu. Fred redisplayed the form, changing the subform size as specified. The outline form option dictates whether the subform is outlined or not.

The Sub_item Option

The **sub_item** option lets you select items in the subform. When you select **sub_item** from the Change tag line, the tag line prompts you to select an item. You select fields or labels within the subform in the same way you select elements in a form. Move the cursor to the element you wish to select and press the Enter key.

Once a subform item is selected, you can perform all form operations on that item such as deleting or editing the item.

Once you have selected **sub_item**, you get a menu that applies only to the elements of the subform. You must exit from this menu to get back to a menu that affects the form.

For example, if you edit the NAME label on the SKI_AREAS subform, and then select **change** from the Edit tag line, instead of getting options for changing the size or characteristics of the subform, you are prompted to enter new text for the NAME label. To

The Change Option

select the subform, you must select **exit** from the Edit tag line. Now when you select **change**, **fred** displays the options for editing the SKI_AREAS subform.

The Size Option

When a subform is selected, the **size** option from the Change tag line affects the display of the contents of the subform. For example:

1. With the SKI_AREAS subform selected, select **size** from the Change tag line. A Form Size menu similar to the one used for resizing the whole subform appears.
2. Select the field next to **height** and type **5** for the height value.
3. Press Enter. The subform is redisplayed so that only the NAME field is visible.
4. Select **size** again and change the height value to **10** to redisplay the entire contents of the subform.

The Exit Option

Selecting **exit** from the Change tag line returns you to the Edit options tag line.

For More Information

For more information on using subforms in applications, refer to Chapter 7.

Chapter 5

The Ski Directory Example: A Fred Tutorial

This chapter describes how to build the forms for an end-user application.

Overview

The following sections provide step-by-step instructions you can use to create the forms for an application that displays and accepts information for ski areas in New England. The application consists of the following:

- Three forms based on relations in the *atlas.gdb* database.
- GDML code to display the forms and insert values into the database.

The forms are created in this chapter. The code is presented in Chapter 7, *Using Forms with GDML*.

Starting the Tutorial

To begin the tutorial, you should have InterBase installed and loaded. You should have a copy of the sample database, *atlas.gdb*, available. For information on accessing *atlas.gdb*, refer to Chapter 1 of this book.

You should also be familiar with basic **fred** operations. For information on navigating and editing in a form, refer to Chapter 2 of this book.

The Application Forms

This section describes how to edit three forms that serve as the basis for the New England Ski Directory application. The forms do the following:

- `NEW_SKI_AREA` accepts information about ski areas.
- `SKI_TRAILS` is used as a subform of `NE_SKI_DIR`.
- `NE_SKI_DIR` structures the display of information for the application.

The `NEW_SKI_AREA` Form

This section describes creating the `NEW_SKI_AREA` form, based on the `SKI_AREAS` relation in the `atlas.gdb` database.

To begin the tutorial, invoke **fred** for `atlas.gdb`, as follows:

```
% fred atlas.gdb
```

To create the `NEW_SKI_AREA` form:

1. Move the cursor to the **create form** option on the “Pick one, please” menu, and press Enter. **Fred** displays a list of relations in the `atlas.gdb` database.
2. Select `SKI_AREAS` from the list of relations. **Fred** displays the form shown in Figure 5-1.

Figure 5-1. The `NEW_SKI_AREA` form

NAME	XXXXXXXXXXXXXXXXXXXXXXXXXX
TYPE	X
CITY	XXXXXXXXXXXXXXXXXXXXXXXXXX
STATE	XXXX
Edit type: EDIT REFORMAT SIZE Exit	

The `NEW_SKI_AREA` form is for entering information about ski areas into the database. To make the form more useful, reformatting the form and adding instructions at the top of the form, follow these steps:

1. Select **edit** from the Edit type tag line at the bottom of the form.
2. Select any field in order to invoke the Edit options tag line.

The Application Forms

3. Select **move** from the Edit options tag line.
4. Select **all** from the Move options tag line.
5. Press the down arrow four times to move all of the form elements down four spaces.
6. Press Enter.
7. Now select the field next to NAME .
8. Select **add** from the Edit options tag line.
9. Select **text** from the Add options tag line.
10. Use the right arrow key to move the cursor to the right side of the field, type **ADD NEW SKI AREA** and press Return.
11. Select the text.
12. Select **move** from the Edit type tag line.
13. Select **item** from the Move options tag line.
14. Use the arrow keys to move the new label to the top and center of the form. The form should look as shown in Figure 5-2.

Figure 5-2. Adding a Header to the Form

```
          ADD NEW SKI AREA
NAME      XXXXXXXXXXXXXXXXXXXXXXXX
TYPE      X
CITY      XXXXXXXXXXXXXXXXXXXXXXXX
STATE     XXXX
Edit type: EDIT  REFORMAT  SIZE  Exit
```

Use the steps described above to append the following text to the TYPE field:

(N = Nordic, A = Alpine, B = Both)

The form should now look as shown in Figure 5-3.

Figure 5-3. The NEW_SKI_AREA Form

```

      ADD NEW SKI AREA

NAME      XXXXXXXXXXXXXXXXXXXXXXXX
TYPE      X (N = Nordic, A = Alpine, B = Both)
CITY      XXXXXXXXXXXXXXXXXXXXXXXX
STATE     XXXX

Edit type:  EDIT  REFORMAT  SIZE  Exit

```

Now the form has greater significance to an end-user who encounters the form as part of the Ski Directory application. Follow these steps to name and save the form:

1. Select any field or label and press Enter.
2. Select **exit** from the Edit options tag line.
3. Select **exit** from the Edit type tag line.
4. Select **save** from the Retention options tag line. **fred** prompts you to enter a name for the form.
5. Type in NEW_SKI_AREA and press Return.
6. Select **commit** from the “Pick one, please” menu.

The SKI_TRAILS Form

This section gives step-by-step instructions for creating the SKI_TRAILS form. The form is based on the SKI_AREAS relation in the *atlas.gdb* database. The subform is used to display information about existing ski areas within the NE_SKI_DIR form.

To create the SKI_TRAILS form:

1. Move the cursor to the **create form** option on the “Pick one, please” menu, and press Enter. **fred** displays a list of relations in the *atlas.gdb* database.
2. Select SKI_AREAS from the list of relations. **fred** displays the form shown in Figure 5-1.

The Application Forms

The `SKI_TRAILS` form is for displaying the information existing in the database about ski areas. For this application, you don't need the `STATE` label and field, so you can delete them, following these steps:

1. Select **edit** from the Edit type tag line at the bottom of the form.
2. Move the cursor to the `STATE` field label and press Enter.
3. Select **delete** from the Edit options tag line.
4. Repeat the previous three steps to delete the `STATE` input field.

The next step is formatting the `SKI_TRAILS` form so that it fits suitably within the `NE_SKI_DIR` form. The labels for the `SKI_TRAILS` form should be removed and added to the `NE_SKI_DIR` form since the information in the subform is repeated and it is inefficient to repeat the labels. The fields will be rearranged so data is presented horizontally for easier scrolling. When the formatting is complete, the form will appear as shown in Figure 5-4.

Figure 5-4. The `NE_SKI_DIR` Form

```
STATE   XXXX
SKI_TRAILS
      NAME           TYPE           CITY
XXXXXXXXXXXXXXXXXXXXX X XXXXXXXXXXXXXXXXXXXXXXXX

TAG XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Edit type: EDIT  REFORMAT  SIZE Exit
```

Follow these steps to reformat the form:

1. Delete the three labels on the `SKI_TRAILS` form, using the instructions from the previous section.
2. Using the steps described in the section *Adding a header to the `NEW_SKI_AREA` form*, move the fields so that they line up horizontally, with the `NAME` field on the left, the `TYPE` field in the middle, and the `CITY` field on the right.

The `SKI_TRAILS` form should appear as shown in Figure 5-5.

Figure 5-5. The `SKI_TRAILS` Form

```
XXXXXXXXXXXXXXXXXXXXX X XXXXXXXXXXXXXXXXXXXXXXXX
```

The next step is to name, save, and close the `SKI_TRAILS` form. To do this:

1. Select any field or label and press **Enter**.
2. Select **exit** from the Edit options tag line.
3. Select **exit** from the Edit type tag line.
4. Select **save** from the Retention options tag line. **fred** prompts you to enter a name for the form.
5. Type in `SKI_TRAILS` and press **Enter**.
6. Select **commit** from the "Pick one, please" menu.

The `NE_SKI_DIR` Form

Now you create the `NE_SKI_DIR` form based on the `STATES` relation. To create the form:

1. Move the cursor to the **create form** option on the "Pick one, please" menu, and press **Enter**. **fred** displays a list of relations in the *atlas.gdb* database.
2. Select `STATES` from the list of relations. **fred** displays the form based on `STATES` shown in Figure 5-6.

Figure 5-6. The `STATES` Form

STATE	XXXX
STATE_NAME	XXXXXXXXXXXXXXXXXXXXXXXXXXXX
AREA	9999999999
STATEHOOD	DDDDDDDDDD
CAPITAL	XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Edit type: EDIT REFORMAT SIZE Exit	

The `STATES` form is the basis for the `NE_SKI_DIR` form you use for displaying the information existing in the database about ski areas in specified states. For this application, the only field you need is the `STATE` field, so you can delete the others, as follows:

1. Select **edit** from the Edit type tag line at the bottom of the form.
2. Move the cursor to the `STATE_NAME` field label and press **Enter**.
3. Select **delete** from the Edit options tag line.
4. Repeat the previous three steps to delete all other labels and fields except for the `STATE` label and field.

The Application Forms

Since this form serves as the basis of the application, it should be as informative as possible. Add a TAG field that displays messages for the user at runtime. To add the TAG field:

1. Select **Add** from the Edit options tag line.
2. Select **field** from the Add options tag line.
3. Select **characteristics** from the Add options tag line.
4. Enter the following in the Characteristics menu:

FIELD_NAME	TAG		
EDIT_STRING	X (25)		
FILL_STRING	XXXXXXXXXXXXXXXXXXXXXXXXXXXX		
UPCASE	N	DATATYPE	CHAR
WIDTH	50	SCALE	
ALIGN_RIGHT	N	LENGTH	50

The NE_SKI_DIR form should now look as shown in Figure 5-7.

Figure 5-7. Adding a Field to NE_SKI_DIR

STATE	XXXX
TAG	XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Edit type: EDIT REFORMAT SIZE Exit	

Creating the SKI_TRAILS Subform

The SKI_TRAILS form is used as a subform of the NE_SKI_DIR to display a many-to-one relationship of ski areas to states. For each state specified, you can use a subform to display all ski areas in that state. To add SKI_TRAILS as a subform to NE_SKI_DIR:

1. Select the STATE field and press Enter.
2. Select **add** from the Edit options tag line.
3. Select **repeating subform** from the Add tag line. Fred displays a menu listing all of the forms in the database.
4. Select SKI_TRAILS from the list. **fred** displays the NE_SKI_AREA form with the subform, as shown in Figure 5-8.

Figure 5-8. Adding a Subform to NE_SKI_DIR

```

STATE      XXXX

TAG XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

SKI_TRAILS
  XXXXXXXXXXXXXXXXXXXXXXXX X XXXXXXXXXXXXXXXXXXXXXXXX

Edit type: EDIT  REFORMAT  SIZE Exit
    
```

To complete the reformatting of the NE_SKI_DIR form, follow these steps:

1. Using the instructions in the section *Formatting the NEW_SKI_AREA Form*, move the TAG label and field to the bottom right of the NE_SKI_DIR form.
2. Add text so that the fields for the SKI_TRAILS output are labeled with their correct names. The form should look as shown in Figure 5-9.

Figure 5-9. Formatting the NE_SKI_DIR Form

```
STATE      XXXX
SKI_TRAILS
          NAME          TYPE          CITY
XXXXXXXXXXXXXXXXXXXXXXXXX  X  XXXXXXXXXXXXXXXXXXXXXXXX

TAG XXXXXXXXXXXXXXXXXXXXXXXXXXXX

Edit type: EDIT  REFORMAT  SIZE Exit
```

Finally, name, save, and close the NE_SKI_DIR form. To do this:

1. Select any field or label and press Enter.
2. Select **exit** from the Edit options tag line.
3. Select **exit** from the Edit type tag line.
4. Select **save** from the Retention options tag line. **Fred** prompts you to enter a name for the form.
5. Type in NE_SKI_DIR and press Enter.
6. Select **commit** from the "Pick one, please" menu.

Completing the fred Session

All of the forms for the New England Ski Directory are defined and formatted. End your **fred** session after committing the changes to the database. To commit the changes and exit **fred**:

1. Select **commit** from the “Pick one, please” menu.
2. Select **Exit Form Editor** from the same menu.

Using the Forms in an Application

The next chapter describes how forms are used in **qli** and GDML. Refer to the section on the New England Ski Directory Application for the GDML code that uses the forms created in this chapter as an application interface.

For More Information

For more information on:

- Keyboard commands, refer to Appendix A, *Platform Specific Notes* for machine-specific keyboard variations.
- GDML statements for use with forms, see Chapters 7 and 8 of this manual.

Chapter 6

Using Forms with Qli

This chapter describes storing and retrieving data from an InterBase database using forms through **qli**.

Overview

Forms you define using **fred** can be incorporated into applications or accessed using **qli**. This chapter describes using forms interactively in **qli**.

Using Forms in Qli

When using forms in **qli**, you can store values in preexisting or system-generated forms using **qli**. **Qli** allows you to use forms in conjunction with the following operations:

- **print**
- **store**
- **modify**

qli's support of forms is limited to simple forms. **qli** *does not* support subforms. Thus, the STATE_SKI example form created in Chapter 3 or the NE_SKI_DIR example form described in the previous chapter does not work in **qli**. You can, however, view and store data using the SKI_AREAS form.

The following sections describe manipulation of values in **qli** using the SKI_AREAS form to illustrate operations.

Invoking Forms in Qli

There are two ways of working in a form environment in **qli**. You can:

- Invoke forms automatically using the **set form** command
- Invoke forms explicitly using the **using form** clause in a **print**, **store**, or **modify** statement

The two options are described in the following sections.

Invoking Forms Automatically

The **set form** command allows you to turn on the forms facility for a **qli** session, or for as long as you want to use forms in **qli**. Once you have executed **set form**, a form is displayed for each **print**, **store**, or **modify** statement invoked. You can turn off the automatic forms facility by issuing the **set no form** command.

For example, to display the existing values stored in the **SKI_AREAS** relation in **qli**, type:

```
QLI> print ski_areas sorted by state
```

NAME	TYPE	CITY	STATE
Birchwood Acres	N	Groton	MA
Great Farm	N	Carlisle	MA
Bretton Woods	B	Mt. Washington	NH
Waterville Valley	B	Waterville Valley	NH
Windblown	N	New Ipswich	NH
Wilderness	B	Dixville Notch	NH
Epson Hills	N	Stowe	VT
Trapp Family Lodge	N	Stowe	VT
Mt. Mansfield	B	Stowe	VT

To view this data in a form, use the **set form** statement. When you select the **set form** option, **qli** looks for an existing form for each **print**, **store**, or **modify** operation you invoke. If there is no existing form, **qli** puts up a default form, using the same default format that **fred** uses.

Thus, to display the relation in a form, type:

```
QLI> set form
QLI> print ski_areas sorted by state
```

Figure 6-1. Invoking the SKI_AREAS Form

NAME	Birchwood Acres
TYPE	N
CITY	Groton
STATE	MA
<enter> to continue, <pf1> to stop	

Note

The set form option only works for GDML statements.

The ski areas are displayed in alphabetical order by state. To see the next ski area and its corresponding data, press **Enter**. To quit the form and return to the **qli** prompt, press PF1.

Note

The termination key may be different according to the platform on which you are working. Refer to Appendix A for platform-specific notes.

When you no longer want forms invoked automatically, type:

```
QLI> set no form
```

Invoking Forms Explicitly

The other option is to invoke a form as needed with the **using form** clause. For example, if you want to invoke a form for the SKI_AREAS relation, you can call the form explicitly in the following way:

```
QLI> print ski_areas sorted by state using form
```

Qli displays the form shown in Figure 6-1.

Note

If you do not specify a form name, qli paints a default form, just as **fred** does.

If there is a corresponding form in the database, you can specify the form name.

In Chapter 5, you created a form named `SKI_TRAILS` based on the `SKI_AREAS` relation. To invoke the `SKI_TRAILS` form:

```
QLI> print ski_areas sorted by state using
CON> form ski_trails
```

Figure 6-2. Invoking the SKI_TRAILS Form

Birchwood Acres	N	Groton
<p><enter> to continue, <pfl> to stop</p>		

Displaying Limited Fields in a Form

To express a record selection expression that joins relations or limits the fields to be displayed, use a view.

For example, suppose you only wanted the `NAME` and `STATE` fields of the `SKI_AREAS` relation displayed in a form. You can create a view containing only the fields you want, then invoke the view in a form. The following example uses the SQL **create view** statement to define a view called `SKI_STATE`:

```
QLI> create view ski_state as
CON> select name, state from ski_areas order by state
QLI> print ski_state using form
```

NAME	Birchwood Acres
STATE	MA

<enter> to continue, <pfl> to stop

Using Forms to Enter and Modify Data

In addition to displaying data, you can use forms to store and modify data. For example, in the `SKI_AREAS` relation, you may want to add new ski area records. To do so, combine a **store** statement with a **using form** clause, as follows:

```
QLI> store ski_areas using form ski_areas
```

Invoking Forms in Qli

```
NAME                XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
TYPE                X
CITY                XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
STATE               XXXX

<enter> to continue, <pf1> to stop
```

Refer to Chapter 2 for information on navigating the form and editing fields.

Similarly, you can use the **using form** clause with a **modify** statement to identify fields that can be modified, as follows:

```
QLI> for ski_areas with name = "Bretton Woods"
CON> modify using form
```

```
NAME                
TYPE                
CITY                
STATE               

<enter> to continue, <pf1> to stop
```

The updatable fields are displayed in reverse video. Refer to Chapter 2 for information on navigating the form and editing fields.

Formatting a Form in Qli

You can call the forms editor from **qli** using the **edit form** statement followed by a form or relation name.

For example, the following invokes the form editor for the form based on the **RIVERS** relation:

```
QLI> edit form rivers
```

RIVER	XX
SOURCE	XXXX
OUTFLOW	XX
LENGTH	9999999999

For details on the edit options, refer to Chapter 3, *Editing a Form*.

When you have finished editing a form, exit the form editor by pressing the PF1 key. To save the edited version of the form, commit the current transaction:

```
QLI> commit
```

To revert to the last saved version of the form, roll back the current transaction:

```
QLI> rollback
```

For More Information

For More Information

For information on using forms in **qli**, refer to:

- The chapter on forms in the *Qli Guide*.
- The *Qli Reference* for the syntax of the following statements:
 - **set forms**
 - **print using form**
 - **store using form**
 - **modify using form**
 - **edit form**

Chapter 7

Using Forms with GDML

This chapter describes creating and using forms interfaces in programs.

Overview

You can use form manipulation statements in GDML or SQL programs.

GDML supports both forms and subforms. GDML provides form manipulation statements for displaying forms and using forms to gather and display data. There is also a menu facility that lets you build and display menus to provide users with options that execute program code.

Note

If you use GDML with forms and you have your code in more than one file, you must include `WINDOW_SCOPE GLOBAL` in the file containing the main routine and `WINDOW_SCOPE EXTERN` in all other files. If you do not include these statements, link errors will occur.

Displaying a Form

The form manipulation capabilities for GDML center on the **for_form** statement. The **for_form** statement is terminated by the **end_form** statement. Within the scope of the **for_form** block, you can include commands to:

- Assign values to form fields
- Display the form
- Write values to the database

The **for form** statement (the statement can also be written **for_form**) binds a form to a window, associates the form with the program, and prepares the form for display.

Aside from the **for_form** statement, some of the statements you can use to manipulate a form are:

- The **for** loop inside the **for_form** statement. This creates the record selection you want displayed. You do not need to display every field on the predefined form. Placing the **for** loop inside the **for_form** loop also avoids reinitializing the form for each record retrieved.
- *Context variables* allow you to identify the target (that is, the form you want displayed), and the source (the database relation and records). In the following program, the variable representing the target is *x*, and the variable representing the source is *s*.
- The **display** statement displays the form fields specified. The sample program lists all of the fields. You can display selected fields, or use the option **displaying *** to display all fields.
- The **if** statement sets up a condition, using the **TERMINATOR** field to test for the terminating action (in this case pressing the F1 key) that breaks out of the **for** loop and returns control to the application. InterBase assigns to the terminator field the last key passed by the user during the **display** statement. For example, the terminator can show the user pressed the Enter key, a function key, or updated a field that has a **waking_on** attribute.
- The ending statements **end_for**, **end_form**, **commit**, and **finish** close off the actions started at the beginning of the program.

The following program displays the **SKI_AREAS** form. To run the example, you must first create the **SKI_AREAS** form, as described in Chapter 2, *Creating a Form*:

```
database db = "atlas.gdb";
main ()
{
  ready;
```

```
start_transaction;

for_form f in ski_areas
  for s in ski_areas sorted by s.state
    strcpy (f.name, s.name);
    strcpy (f.type, s.type);
    strcpy (f.city, s.city);
    strcpy (f.state, s.state);
    display f displaying name, type, city,
            state;
    if (f.terminator == PYXIS_$KEY_PF1)
      break;
  end_for;
end_form;
commit;

finish;
}
```

Creating a Window

The **create_window** statement lets you override the dimensions for a window that the **display** statement provides for a form. You specify dimensions in a program by assigning values to the variables **gds_\$width** and **gds_\$height** and including a **create_window** statement.

gds_\$width is measured in character cell units. It can have between 1 and 80 columns.

gds_\$height is also measured in character cell units. It can have between 1 and 24 rows. Using these variables you can position a form in a window.

The following code fragment uses the **create_window** statement to set the height of the window to 20 rows and the width to 80 columns:

```
DATABASE DB = 'atlas.gdb'

main()
{
char answer;
short found;

gds_$height = 20;
gds_$width = 80;
CREATE_WINDOW;

READY;
START TRANSACTION;
```

Deleting a Window

If you want to close a window that you have previously opened by displaying a form or by explicitly creating a window for one, include a **delete_window** statement in an appropriate place in your program. The program at the end of this chapter includes a **delete_window** statement.

Using Attributes

GDML has three attributes unique to forms that you can use to set aspects of a form, or to determine what action has occurred. The attributes you use for this are the *.state* attribute, the *.terminator* attribute, and the *.terminating_key* attribute. By using attributes you can determine such things as when control should return to an application, whether an entered value is valid, or whether a user has entered a value for a field that requires a value.

The .State Attribute

Each field in a form has attributes associated with it. For example, its datatype or its length. Another attribute is the field's state, or condition. In forms, GDML allows you to manipulate this condition on input (before executing a display) and on output (after executing a display).

The *.state* attribute is a field-specific attribute. The values you can assign to it on input are similar to the options you can use in the display statement. Although this is a less common usage of *.state*, it is valuable if you want the application to determine at run time how to treat the field. If you do this, then you need to include the **overriding** option in the **display** statement.

To use the *.state* attribute, you assign **pyxis** constant values to fields. They are listed in the following table.

Table 7-1. Input values for *.state* attribute

Mnemonic	Numeric	Function
PYXIS_\$OPT_DISPLAY	1	Displays the field with the value given in the program
PYXIS_\$OPT_UPDATE	2	Allows the user to update this field
PYXIS_\$OPT_WAKEUP	4	Returns control to the application program immediately if the user updates this field
PYXIS_\$OPT_POSITION	8	Places the cursor on this field when the display executes

If you want the application to determine what actions have occurred against a particular field, you can use the `.state` attribute on output. Since this usage of `.state` concerns output, the values are meaningful only after a **display** statement. The values on output in the `.state` field are numeric values, so you can use relational operators to compare a `.state` value to a **pyxis** constant value. Those values are listed in the following table.

Table 7-2. Output values for .state attribute

Mnemonic	Numeric	Function
PYXIS_\$OPT_USER_DATA	4	The user has changed the value in this field
PYXIS_\$OPT_INITIAL	3	The value in this field has not changed since the previous display statement
PYXIS_\$OPT_DEFAULT	2	This field has default values
PYXIS_\$OPT_NULL	1	The field has no default values, nor has the user changed it

In the following example, the `.state` attribute is used to determine whether the user has entered data, and if so what action to take:

```
FOR_ITEM FC IN F.CITY_POP_LINE
  if (FC.POPULATION.STATE == PYXIS_$OPT_USER_DATA)
    FOR C IN CITIES WITH C.CITY = FC.CITY
      AND C.STATE = F.STATE
      MODIFY C USING
        C.POPULATION = FC.POPULATION;
      END_MODIFY;
    END_FOR;
  END_ITEM;
```

Since these values are also numeric, you can make comparisons using relational operators.

The .Terminator Attribute

The .terminator attribute is a form-specific attribute. It contains a value that reflects the last key pressed by a user during a form display. You have read-only access to this value, and you use it to determine why control was returned to the application. With this information the application determines what action to take. For instance, did the user press Enter, a function key, or enter data into a **waking_on** field. The .terminator attribute can have the following values:

Table 7-3. Possible Values for .terminator Attribute

Mnemonic	Numeric
PYXIS_\$KEY_DELETE	127
PYXIS_\$KEY_UP	128
PYXIS_\$KEY_DOWN	129
PYXIS_\$KEY_RIGHT	130
PYXIS_\$KEY_LEFT	131
PYXIS_\$KEY_PF1	132
PYXIS_\$KEY_PF2	133
PYXIS_\$KEY_PF3	134
PYXIS_\$KEY_PF4	135
PYXIS_\$KEY_PF5	136
PYXIS_\$KEY_PF6	137
PYXIS_\$KEY_PF7	138
PYXIS_\$KEY_PF8	139
PYXIS_\$KEY_PF9	140
PYXIS_\$KEY_ENTER	141
PYXIS_\$KEY_SCROLL_TOP	146
PYXIS_\$KEY_SCROLL_BOTTOM	147

The following code fragment uses the `.terminator` attribute:

```
for_form x in states_form
  for s in states sorted by s.statehood
    strcpy (x.state_name, s.state_name);
    x.statehood = s.statehood;
    x.area = s.area;
    strcpy (x.state, s.state);
    strcpy (x.capital, s.capital);
    display x displaying statehood, area, state,
      state_name, capital;
    if (x.terminator == PYXIS_$KEY_PF1)
      break;
  end_for;
end_form;
```

The `.Terminating_field` Attribute

This form-specific attribute contains the name of the field the cursor was in when the user hit a terminating key. You use this attribute to determine which in a list of **waking_on** fields the user updated to return control to the application. The following example shows one way to use the `.terminating_field`:

```
FOR FORM X IN TEST_FORM
  DISPLAY X ACCEPTING FIELD1, FIELD2, WAKING_ON FIELD1, FIELD2
  if (!strcmp (X.TERMINATING_FIELD, "FIELD2"))
    X.FIELD3 = X.FIELD2 * 3;
  else if (!strcmp (X.TERMINATING_FIELD, "FIELD1"))
    X.FIELD3 = X.FIELD1 * 5;
END_FORM
```

Creating Menus

GDML supports two kinds of menus:

- static
- dynamic

Static menus are menus defined and displayed as you would a form, with all menu options specified.

Dynamic menus are menus created at runtime, with only the menu orientation (horizontal or vertical) specified.

The following sections describe defining static and dynamic menus.

Defining Static Menus

For creating and displaying static menus, GDML provides the **case_menu** statement, which resembles a Pascal **case** statement.

Here are some points about the **case_menu** statement used in the following programming example:

- The first display statement, which displays the SKI_AREAS form, uses the **displaying *** option to display all fields for the SKI_AREAS form.
- The **case_menu** statement is terminated by an **end_menu** statement. The block of code controlled by the **case_menu** statement is divided into smaller blocks by **menu_entree** statements that offer choices to the user. When a user selects a **menu_entree**, only that block of code is executed.
- The **case_menu** statement is qualified by the **transparent** option. This option displays the menu without obscuring the form beneath it.
- “Update Ski Area Type” is the title-string argument that names the menu.
- The **menu_entree** keyword is followed by the text string that explains the choice.
- The second display statement, under the “Change type” menu_entree, allows the TYPE field to be updated, and redisplay the SKI_AREAS form.

The following sample program also performs some default checking to determine whether a new value has been entered for the TYPE field before updating the database record.

The following program illustrates a **case_menu** statement. Once again, the SKI_AREAS form is displayed, but this time a menu prompts the user to modify the type of ski area, or leave the existing value. Enter the following program to display a menu for updating the TYPE field for the SKI_AREAS relation using the predefined

SKI_AREAS form. For information on compiling and executing a program in InterBase, refer to the chapter on preprocessing programs in the *Programmer's Guide*.

To run the example, you must first create the SKI_AREAS form, as described in Chapter 2, *Creating a Form*:

```

database db = "atlas.gdb";
#define CONTINUE 0
#define STOP 1
main ()
{
short    flag;

flag = CONTINUE;
for form x in ski_areas
    for s in ski_areas sorted by s.state
        strcpy (x.name, s.name);
        strcpy (x.type, s.type);
        strcpy (x.city, s.city);
        strcpy (x.state, s.state);
        display x displaying *;

    case_menu (transparent) "Update Ski Area Type?"
        menu_entree "Leave current value" ;;
        menu_entree "Change Type" :
            display x accepting type
            cursor on type waking on type;
            if (x.type.state == PYXIS_$OPT_USER_DATA)
                modify s
                strcpy (s.type, x.type);
                end_modify;
            menu_entree "Exit" :
                flag = STOP;
        end_menu;

    if (flag == STOP)
        break;
    end_for
end_form;
delete_window;
finish;
}

```

Defining Dynamic Menus

For creating and displaying dynamic menus, GDML provides the **for_menu** statement, similar in construct to the **for_form** statement. A dynamic menu is useful for creating a menu the contents of which are not determined until run-time. For example, the Edit Form menu **fred** displays is a dynamic menu since the available forms are not known until run-time.

You build dynamic menus by using the **for_menu** statement and its terminator, the **end_menu** statement. Within these delimiters, you use the substatements **put_item** to write information to the menu and **for_item** to read information from the menu.

Within a **put_item** block, you establish the menu title and menu entrees by giving values to **entree_text**, **entree_length**, and **entree_value**.

Here are some points about the **for_menu** statement:

- You define the menu by assigning a title, title length, and menu orientation (the default orientation is vertical).
- To establish the contents of the menu, you use a **put_item** statement that specifies the source for the menu entree, the display length, and a value to assign to each entree instance.
- You use a **display** statement similar to the display statement used for a form to display the menu.

The **for_menu** statement is used in the sample program at the end of this chapter to display a menu of states in New England plus an **exit** option.

Using Subforms in GDML

As described in Chapter 3, subforms are commonly used for displaying repeating values within a form. The example for subforms uses a subform to present information about ski areas in each of the New England states.

To display a subform in GDML, you use a nested **for form** statement. The subform is controlled by its own loop construct to prevent the primary form from being updated and redisplayed for every change made within the subform.

The following example program displays the NE_SKI_DIR form defined in Chapter 5. The NE_SKI_DIR form includes the SKI_TRAILS form as a subform.

If you plan to run this example program, be sure you have followed the steps in Chapter 5 to create the NEW_SKI_AREAS form and the NE_SKI_DIR form with the SKI_TRAILS subform.

The New England Ski Directory Application

If you have created the forms for the New England Ski Directory application following the steps in Chapter 5, and you want to enter the application code, type the sample program into an editor.

The code for the following program is included with your software in the *examples* directory. The filename is *forms_ski.e*.

To run the tutorial program, you must first preprocess, then compile the program. Refer to the chapter on preprocessing with **gpre** in the *Programmer's Guide*.

```

DATABASE DB = 'atlas.gdb';

static char state_list[7][5] = {"CT", "MA", "ME", "NH", "RI",
"VT", "Exit"};

main()

/*****

*   m a i n
*
*   Prompt the user for a state, then display menu
*   of options
*   until user wants to quit.
*
*****/
```

Using Subforms in GDML

```
{
int state_idx;

READY;
START_TRANSACTION;

state_idx = get_state();

/* Select a state */

while (state_idx != 7)
{
CASE_MENU "Choose one"
MENU_ENTREE "View Ski Areas":
    view_ski_areas (state_list[state_idx]);
MENU_ENTREE "Store New Ski Area":
    store_ski_area (state_list[state_idx]);
MENU_ENTREE "Pick New State":
    state_idx = get_state();
MENU_ENTREE "Exit New England Directory":
    state_idx = 7;
END_MENU;
}

COMMIT;
FINISH;
}

get_state ()
/*****
*   g e t _ s t a t e
*
*   Create a dynamic menu containing the 6 New England
*   states plus an "Exit" option.  Return the state or
*   signal to quit.
*
*****/
{
int i;

FOR_MENU M

    strcpy (M.TITLE_TEXT, "Choose a state");
```



```

M.TITLE_LENGTH = strlen (M.TITLE_TEXT);

for (i = 0; i < 7; i++)
{
    PUT_ITEM E IN M
        strcpy (E.ENTREE_TEXT, state_list[i]);
        E.ENTREE_LENGTH = 4;
        E.ENTREE_VALUE = i;
    END_ITEM;
}

/* User selects a state, or "exit" */

    DISPLAY M;
    return M.ENTREE_VALUE;
END_MENU;
}

view_ski_areas (state)
char *state;
/*****
*   v i e w _ s t a t e
*
*   Display the ski areas in the state selected.
*
*****/
{
    int count = 0;

    FOR FORM F IN NE_SKI_DIR

        strcpy (F.STATE, state);

/*   Fill in the Subform */

    FOR SK IN SKI_AREAS WITH SK.STATE = state
        PUT_ITEM P IN F.SKI_TRAILS
            strcpy (P.NAME, SK.NAME);
            strcpy (P.TYPE, SK.TYPE);
            strcpy (P.CITY, SK.CITY);
        END_ITEM;
        count++;
    END_FOR;
}

```

Using Subforms in GDML

```
    if (!count)
        strcpy (F.TAG, "No ski areas listed.");
    else
        sprintf (F.TAG, "%d ski areas listed.",      count);

    DISPLAY F DISPLAYING *;

END_FORM;
}

store_ski_area (state)
char *state;
/*****
*   s t o r e _ s k i _ a r e a
*
*   Store a ski area record in the state selected.
*
*****/

{

FOR FORM F IN NEW_SKI_AREA

    strcpy (F.STATE, state);

    DISPLAY F DISPLAYING STATE
    ACCEPTING NAME, TYPE, CITY;

    STORE SK IN SKI_AREAS USING
        strcpy (SK.NAME, F.NAME);
        strcpy (SK.TYPE, F.TYPE);
        strcpy (SK.CITY, F.CITY);
        strcpy (SK.STATE, F.STATE);
    END_STORE;

END_FORM;
}
```

Error Handling

Error handling in this release of the forms package is characterized by the following behavior:

- The form statements do not currently support the **on_error** clause. An error, such as **gpre**'s failure to find a referenced form, results in an error message and the termination of the program.
- Conversion errors result in an error message, but do not cause the program to terminate. However, the program has no way of knowing that an error occurred.

For More Information

For More Information

For syntax for the following statements, refer to the entry in Chapter 8:

- **for_form**
- **case_menu**
- **display**
- **for_menu**
- **for_item**
- **put_item**

Chapter 8

Using Blobs with Forms

This chapter illustrates how to use blobs with forms. It requires that you create a form called `VFORM` on the `VARIED_XC` relation. The *atlas.gdb* sample database is used.

```
DATABASE DB = 'atlas.gdb'
/*
 * Here is an example of editing a blob while using forms.
 */
main()
{
  READY;
  START_TRANSACTION;
  view_form();
  COMMIT;
  FINISH;
  exit (0);
}

view_form ()
{
```

```

FOR FORM F IN VFORM

  FOR R IN VARIED_XC
    strcpy (F.area_name, R.area_name );
    strcpy (F.state, R.state );
    F.comments = R.comments;

  DISPLAY F DISPLAYING * ACCEPTING area_name, state;

  FOR S IN VARIED_XC WITH S.area_name = F.area_name
    MODIFY S USING
      strcpy ( s.state, f.state );
      BLOB_edit ( GDS_REF ( s.comments ), DB, gds_$trans,
        "comments" );
    END_MODIFY;
  END_FOR;

END_FOR;
END_FORM;
DELETE_WINDOW;
}

```

Chapter 9

Forms Reference

This chapter describes the syntax and usage of GDML statements used for form manipulation.

Overview

This section provides syntax and examples for the following forms-specific GDML statements:

- **case_menu**
- **display**
- **for form**
- **for_item**
- **for_menu**
- **put_item**

Chapter 7 gives examples for the statements described in the following sections.

Case_Menu Statement

Function The `case_menu` statement displays a menu in the forms window and executes the code associated with the user's choice.

Syntax

```

case_menu [(options)]title-string menu-entrees
end_menu
menu-entrees::= {menu_entree entree-string}
options::= {vertical|horizontal|transparent}

```

title-string

A quoted string that provides the title line for the menu.

menu_entree

Establishes a line that appears in a menu and introduces a block of code that executes if the line is chosen:

- All code between the keywords `case_menu` and `end_menu` must be introduced by `menu_entree` labels.
- To specify an option to continue without taking any action, include a null statement under the `menu_entree` label.

Because the `case_menu` statement is like a Pascal `case` statement, and not like a C `switch` statement, choosing a menu item executes only the code between that item and the next item or `end_menu`.

entree-string

A quoted string that becomes a line in a vertical menu or a selection item in a horizontal menu.

vertical

Displays the menu choices in a vertical format. This display option is the default. A vertical menu obscures the contents of the current form with its menu choices.

horizontal

Displays the menu choices in a horizontal format. A horizontal menu, also called a "tag-line menu," displays the menu choices on the bottom line of the current form.

transparent

Displays the menu choices, obscuring only those parts of the form directly behind the menu.

Example

The following example cycles through the *atlas.gdb* database, displaying a menu for each displayed state and prompts the user to update the state's capital or exit from the iteration:

```

database db = "atlas.gdb";

#define CONTINUE 0
#define STOP 1

main()
{
short  flag;

ready db;

start_transaction;

flag = CONTINUE;
for form x in show_state
  for s in states sorted by s.statehood
    strcpy (x.state_name, s.state_name);
    x.statehood = s.statehood;
    x.area = s.area;
    strcpy (x.state, s.state);
    strcpy (x.capital, s.capital);
    display x displaying *;

    case_menu (transparent) "Alter State?"
      menu_entree "No Changes":
        ;
      menu_entree "Change Capital":
        display x accepting capital
        cursor on capital waking on capital;
        if (x.capital.state ==
PYXIS_$OPT_USER_DATA)
          modify s
            strcpy (s.capital, x.capital);
          end_modify;

```

Case_Menu Statement

```
        menu_entree "Exit" :
            flag = STOP;
        end_menu;

        if (flag == STOP)
            break;
        end_for;
    end_form;
    delete_window;
    commit;
    finish;
}
```

Troubleshooting See the appendix about error handling in the *Programmer's Reference* for a discussion of errors and error handling.

See Also See the entries in this chapter for:

- **display**
- **for_form**

Display

Function

The **display** statement displays a form or a menu on the user's screen. In a form, it also:

- Controls the fields that are displayed, those that can be updated, the cursor position, and other characteristics of the form.
- In a menu, it controls the orientation of the display and how the menu appears in relation to other menus on the screen.
- In a form, each display attribute can appear at most once per **display** statement.

A **display** statement must occur inside a **for_form - end_form** block or inside a **for_menu - end_menu** block.

Syntax

Form format:

```
display form-context-variable[display-attribute...]

display-attribute::=
accepting field-list
cursor on field-name/
displaying field-list/
no_wait |
overriding field-list/
waking on field-list
field-list::=
{*|field-commalist}::=
{field-name|subform.subform-field-name}
```

Menu format:

```
display menu-context-variable
[horizontal|vertical]
[transparent|opaque]
```

form-context-variable

Provides a name associated with this instance of the form in the **for_form** statement.

field-list

Specifies an asterisk (*) indicating that all fields are listed, a commalist of form field names without any qualifiers, or a field in a subform. The subform variant allows you to both read and write a field from a subform, a capability not available in the **for_item** and **put_item** statements by themselves.

accepting

Specifies which fields can be updated.

cursor on

Specifies the field on which the cursor is positioned when the form appears.

displaying

Lists the fields for which values established in the program should replace the fill characters established in the form definition. If you want to update the value between **display** statements, you must signal the change by including the field in the **displaying** list of the second *display* statement.

no_wait

Updates the information on the screen, but does not pause for user input.

overriding

Lists the fields whose display attributes are controlled at runtime by the program.

waking on

Lists the fields that cause control to return to the program if the user changes their value. If you supply more than one field in the **waking on** list, you should test the special field **TERMINATING_FIELD** when control returns to your program to see which field caused the wake-up.

If the wake-up is on a repeating group item, you can reference other items from the repeating group.

menu-context-variable

A qualifier that references the context of the menu in the **for_menu** statement.

horizontal | **vertical**

Specifies the orientation of the menu on the screen. The default is vertical.

transparent | **opaque**

Transparent specifies the menu displays on the screen without obscuring what is already there. Opaque specifies the menu displays on the screen and covers what is already there. The default is opaque.

Examples

The following code fragment displays records from the **STATES** relation through a form:

```
for form x in states
  for s in states sorted by s.statehood
    strcpy (x.state_name, s.state_name);
    x.statehood = s.statehood;
    x.area = s.area;
    strcpy (x.state, s.state);
    strcpy (x.capital, s.capital);
    display x displaying statehood, area,
state,
        state_name, capital;
    if (x.terminator == PYXIS_$KEY_Pf1)
      break;
  end_for;
end_form;
```

The following code fragment creates a dynamic menu displaying the six New England states plus an **exit** option:

```
FOR_MENU M
strcpy (M.TITLE_TEXT, "Choose a state");
M.TITLE_LENGTH = strlen (M.TITLE_TEXT);
for (i = 0; i < 7; i++)
{
  PUT_ITEM E IN M
  strcpy (E.ENTREE_TEXT, state_list [i]);
```

Display

```
        E.ENTREE_LENGTH = 2;  
        E.ENTREE_VALUE = i;  
        END_ITEM;  
    }  
    DISPLAY M;  
    return M.ENTREE_VALUE;  
END_MENU
```

Troubleshooting

See the appendix about error handling in the *Programmer's Reference* for a listing of errors and error codes.

See Also

See the entries in this chapter for:

- **case_menu**
- **case form**
- **for_menu**

For Form

Function

The **for form** statement binds a form definition to a window and creates a context in which form fields can be referenced. This statement does not cause a form to appear on the screen. Use the **display** substatement to make the form appear on your screen.

For form statements can be nested. As the forms are displayed, they overlay each other. Unless a form is specified as **tag** or **transparent**, it completely covers the previously displayed form.

Syntax

```

for form [(options)] form-context-variable in
[database-handle.] form-name
form-context-variable.field-name [.state]
    statement end_form
options::= {transparent| tag|
    form-handle form-handle-variable |
    transaction-handle transaction-handle-
        variable

```

transparent

Pushes a transparent form over the current form, covering only those portions that are actually behind text on the top form.

tag

Displays a one-line tag form horizontally in the bottom line of the form.

form-handle

Specifies a variable by which **gpre** can refer to the form in its calls to **pyxis**. If you do not specify a form-handle, **gpre** assigns it a unique name. If you do specify a form-handle, you can use the variable to invoke the form in different routines.

transaction-handle

Specifies the transaction you want to commit. If the transaction you want to commit has a transaction handle associated with it, you must use that handle when you commit the transaction. If you do not specify a transaction handle on a **commit** statement, InterBase commits the “default” transaction. The default transaction is what InterBase starts when you use a **start_transaction** statement without a handle.

form-context-variable

The context variable qualifies references to the form fields to distinguish them from database fields or program variables.

form-name

Specifies the form to bind. The form name must be the name of a form already defined in a database. If you include a database handle, the form must be in that database. Otherwise, **gpre** searches databases referenced by the program, beginning with the most recently declared database.

statement

Any host language statement or a GDML **display**, **for_item**, or **put_item** statement. See the entries in this chapter for these statements. The **for form** statement allows free reference to form fields inside the **for form** and **end_form** structure. If your program performs a statement, such as a return from a subprogram, that cause it not to drop through to the **end_form** terminator, it first executes a call to **pyxis_\$pop_window**. **gpre** automatically provides the context of **gds_\$window**. The syntax for this call follows.

C:
pyxis_\$pop_window (&gds_\$window)

All other languages:

pyxis_\$pop_window (gds_\$window)

Example

The following code fragment displays records from the STATES relation through a form:

```

for form x in states
    for s in states sorted by s.statehood
        strcpy (x.state_name, s.state_name);
        x.statehood = s.statehood;
        x.area = s.area;
        strcpy (x.state, s.state);
        strcpy (x.capital, s.capital);
        display x displaying statehood, area,
state,
            state_name, capital;
        if (x.terminator == PYXIS_$KEY_PF1)
            break;
    end_for;
end_form;

```

Troubleshooting

See the *Programmer's Reference* for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **case_menu**
- **display**
- **for_item**
- **put_item**

For_Item

Function

The **for_item** statement is used inside a **for_form** statement to read items from a repeating group. The **for_item** statement allows only *read* access to the fields in its substatements.

Syntax

Form format:

```
for_item subform-context-variable in
           form-context-variable.subform-name
           statement
end_item
```

Menu format:

```
for_item entree-context-variable in menu-
           context-variable entree-assignment-
           statements
end_item
```

subform-context-variable

Specifies a context variable for the subform. This context variable must uniquely identify the subform in the form.

form-context-variable.subform-name

Specifies the subform name qualified with the context variable associated with the form in which the subform exists.

entree-context-variable

A qualifier that references the context of the entree in the **for_item** statement.

menu-context-variable

A qualifier that references the context of the menu in the **for_menu** statement.

entree-assignment-statements

Host language statements that read the values of *entree-context-variable.entree_text*, *entree-context-variable.entree_length*, and *entree-context-variable.entree_value*.

Example

The following code fragment modifies database records appearing in a subform:

```
FOR FORM F IN CITY_STATES
FOR_ITEM FC IN F.CITY_POP_LINE
  IF (FC.POPULATION.STATE ==
PYXIS_$OPT_USER_DATA)
    FOR C IN CITIES WITH C.CITY = FC.CITY
    AND C.STATE = F.STATE
    MODIFY C USING
      C.POPULATION = FC.POPULATION;
    END_MODIFY;
  END_FOR;
END_ITEM;
END_FORM
```

Appendix C of this book contains the program from which this extract was taken.

Troubleshooting

See Appendix A of the *Programmer's Reference* for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **case_menu**
- **display**

For_Menu

Function

The **for_menu** command lets you create a *dynamic menu*.

A dynamic menu obtains the specifications for its title, entries and format at runtime. These specifications most often come from dynamic user input or from database values. This differs from the **case_menu** command which requires you to specify all of these characteristics before the application is compiled.

Syntax

```
for_menu [(menu_handle menu-handle)] menu-
context-variable
menu-title-assignment-statements
entree-assignment-statements
display statement
menu-result-statements
end_menu
```

menu_handle

Specifies a variable by which **gpre** can refer to the menu in its calls to **pyxis**. If you do not specify a menu-handle, **gpre** assigns it a unique name. If you do specify a menu-handle, you can use the variable to invoke the menu in different routines.

menu-context-variable

A qualifier that references the context of the menu in the **for_menu** statement.

menu-title-assignment-statements

Host language statements in which you assign values to *menu-context-variable.title_text* and *menu-context-variable.title_length*. These statements must appear between the **for_menu** statement and the **end_menu** statement, and before the **display** statement.

entree-assignment-statements

Host language statements in which, within a **put_item** statement, you assign values to *entree-context-variable.entree_text*, *entree-context-variable.entree_length*, and *en-*

tree-context-variable.entree_value. These statements must appear between the **for_menu** statement and the **end_menu** statement, and before the **display** statement.

display statement

The display statement displays a menu on the user's screen. This statement must appear between the **for_menu** statement and the **end_menu** statement and after all title and entree assignment statements.

menu-result-statements

Host language statements that use the values of *menu-context-variable.entree_text*, *menu-context-variable.entree_length*, and *menu-context-variable.entree_value* for the entree selected from the menu.

The *menu-result-statement* also reads the *menu-context-variable.terminator* to determine what key was pressed to terminate the menu selection. This statement must appear between the **for_menu** statement and the **end_menu** statement, and after the **display** statement.

Example

The following C code fragment creates a menu consisting of the first ten cities in the CITIES relation. Once the user chooses a city, the program displays the selected city name and its population:

```
FOR_MENU M
    strcpy (M.TITLE_TEXT, "Choose a City");
    M.TITLE_LENGTH = strlen(M.TITLE_TEXT)
    FOR FIRST 10 C IN CITIES SORTED BY DESCENDING
    POPULATION
        PUT_ITEM E IN M
            strcpy (E.ENTREE_TEXT, C.CITY);
            E.ENTREE_LENGTH =
strlen(E.ENTREE_TEXT);
            E.ENTREE_VALUE = C.POPULATION;
        END_ITEM
    END_FOR
for (;;)
{
    DISPLAY M VERTICAL
    if (M.TERMINATOR == PYXIS_$KEY_Pf1)
```

For_Menu

```
        break;
        printf ("You chose %s, population %d\n",
               M.ENTREE_TEXT, M.ENTREE_VALUE);
    }
END_MENU
```

Troubleshooting See the appendix about error handling in the *Programmer's Reference* for a listing of errors and error codes.

See Also See the entries in this chapter for:

- **case_menu**
- **display**
- **for_item**
- **put_item**

Put_Item

Function

The **put_item** statement is used inside a **for_form** statement to write items to a repeating group. Each **put_item** statement adds one row (that is, one group) to a subform. It is used inside a **for_menu** statement to add instances of entrees to a menu. You terminate a **put_item** statement with an **end_item**.

Syntax

Form format:

```
put_item subform-context-variable in form-context-variable.subform-name
    statement
end_item
```

Menu format:

```
put_item entree-context-variable in menu-context-variable
    entree-assignment-statements
end_item
```

subform-context-variable

Specifies a context variable for the subform. This context variable must uniquely identify the subform in the form.

form-context-variable.subform-name

Specifies the subform name qualified with the context variable associated with the form in which the subform exists.

entree-context-variable

A qualifier that references the context of the entree in the **for_item** statement.

entree-assignment-statement

Host language statements in which you assign values to *entree-context-variable.entree_text*, *entree-context-variable.entree_length*, and *entree-context-variable.entree_length*.

Examples

The following program adds records to a subform's repeating groups:

```

database db = "atlas.gdb";

main()
{
ready;
start_transaction;
for s in states
    for form f in city_states
        strcpy (f.capital, s.capital);
        f.statehood = s.statehood;
        strcpy (f.state_name, s.state_name);
        f.area = s.area;
        for c in cities with c.state = s.state
            put_item cs in f.cities
                strcpy (cs.city, c.city);
                cs.altitude = c.altitude;
                cs.population = c.population;
            end_item;
        end_for;
        display f displaying *
    end_form;
end_for;
COMMIT;
FINISH;
}

```

The following code fragment creates a dynamic menu displaying the six New England states plus an "Exit" option:

```

FOR_MENU M
strcpy (M.TITLE_TEXT, "Choose a state");
M.TITLE_LENGTH = strlen (M.TITLE_TEXT);
for (i = 0; i < 7; i++)
{
    PUT_ITEM E IN M
    strcpy (E.ENTREE_TEXT, state_list [i]);
    E.ENTREE_LENGTH = 2;
    E.ENTREE_VALUE = i;
}

```



```
        END_ITEM;  
    }  
    DISPLAY M;  
    return M.ENTREE_VALUE;  
END_MENU
```

Troubleshooting

See the appendix about error handling in the *Programmer's Reference* for a listing of errors and error codes.

See Also

See the entries in this chapter for:

- **case_menu**
- **display**
- **for_form**
- **for_item**

Appendix A

Platform Specific Notes

Apollo Notes

This section describes differences between **fred**'s behavior documented in previous chapters and **fred**'s behavior on Apollo workstations.

Some of the differences are:

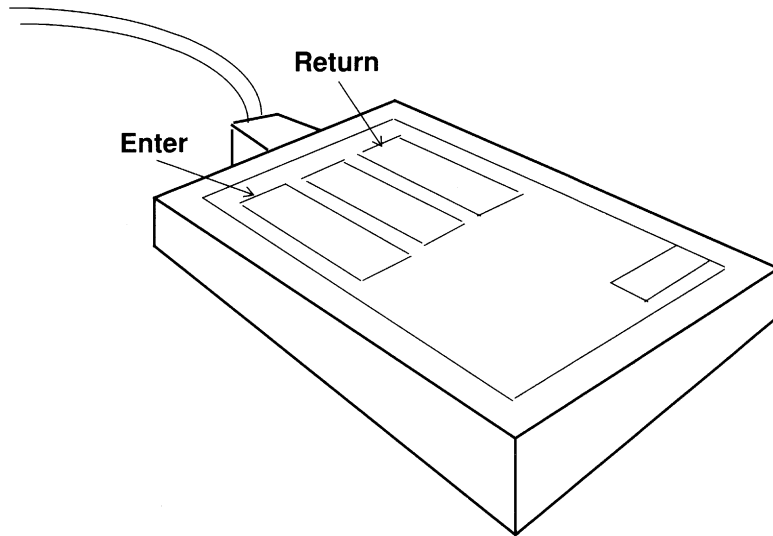
- Mouse support with an Apollo workstation.
- Keyboard commands used for editing and navigation.

Mouse Support

InterBase supports the mouse if you are using forms on an Apollo. You can use the mouse to navigate in forms and menus, and use the mouse buttons in the place of the Return and Enter keys.

Mouse actions are mapped to keystrokes as shown in Figure A-1.

Figure A-1. Mouse Support in Fred



Moving the mouse corresponds to the arrow keys. For example, using the mouse to move the cursor toward the top of the screen is the same as using the up arrow key.

Editing Keys

Apollo workstations have different key assignments for some forms editing functions. The following table lists the editing functions described in Chapter 2, *Editing a Form*, and points out key command variations on Apollo keyboards.

The keys in bold typeface are edit keys; the others are cursor movement keys.

Figure A-2. Apollo Editing Key Commands

Edit Function	Non-Apollo Key	Apollo Key
Edit	Ctrl-G	EDIT
Insert/overstrike	Ctrl-A	INS
Erase	Ctrl-U	LINE DEL
Insert	any printable character	any printable character

Figure A-2. Apollo Editing Key Commands continued

Edit Function	Non-Apollo Key	Apollo Key
Right	right arrow	right arrow
Left	left arrow	left arrow
Delete	Delete	Backspace
Delete next character	Ctrl-F	CHAR DEL
Go to start	Ctrl-H	left bar arrow
Go to end	Ctrl-E	right bar arrow

Sun Notes

If you are using forms on a Sun workstation, you may encounter some unexpected behavior. Here are some things to beware of:

- *Forms require a shell tool window.* If you invoke a form in any other type of window (for example, a console), InterBase forces the window to behave like a shell tool window. Thus, don't be surprised if you see window scroll bars disappearing.
- *Forms might invert the video display.* For example, if you are using forms in **qli**, you might find that all of your screen display is shown in reverse video.

To fix this problem, type:

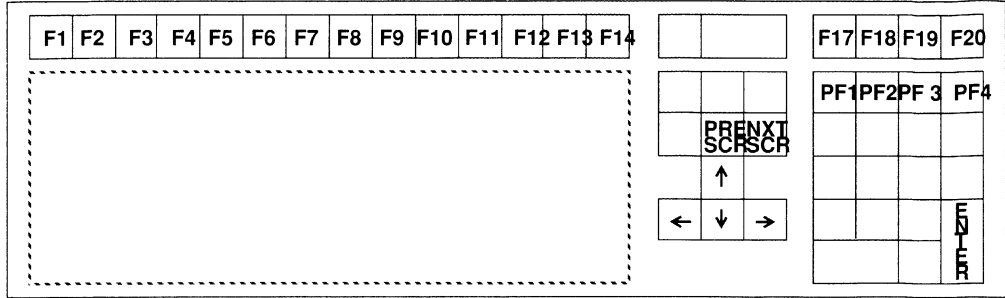
```
QLI> spawn "reset"
```

Keyboard Diagrams

The following diagrams show the key value that each programmable key produces. The diagrams show key-mapping for:

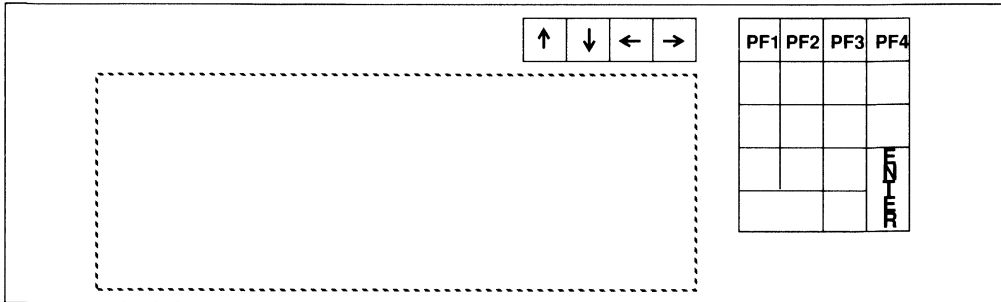
- VT220
- VT100
- Sun 3 and 4
- Sun 386i
- Apollo DNxxx

Figure A-3. The VT220 Keyboard



InterBase Constant	Value Returned	Generated by
PYXIS_\$KEY_DELETE	127	Delete
PYXIS_\$KEY_UP	128	Up-arrow
PYXIS_\$KEY_DOWN	129	Down-arrow
PYXIS_\$KEY_RIGHT	130	Right-arrow
PYXIS_\$KEY_LEFT	131	Left-arrow
PYXIS_\$KEY_PF1	132	PF1
PYXIS_\$KEY_PF2	133	PF2
PYXIS_\$KEY_PF3	134	PF3
PYXIS_\$KEY_PF4	135	PF4
PYXIS_\$KEY_PF5	136	F17
PYXIS_\$KEY_PF6	137	F7
PYXIS_\$KEY_PF7	138	F8
PYXIS_\$KEY_PF8	139	F9
PYXIS_\$KEY_PF9	140	F10
PYXIS_\$KEY_ENTER	141	ENTER
PYXIS_\$KEY_SCROLL_TOP	146	PREV SCR
PYXIS_\$KEY_SCROLL_BOTTOM	147	NEXT SCR

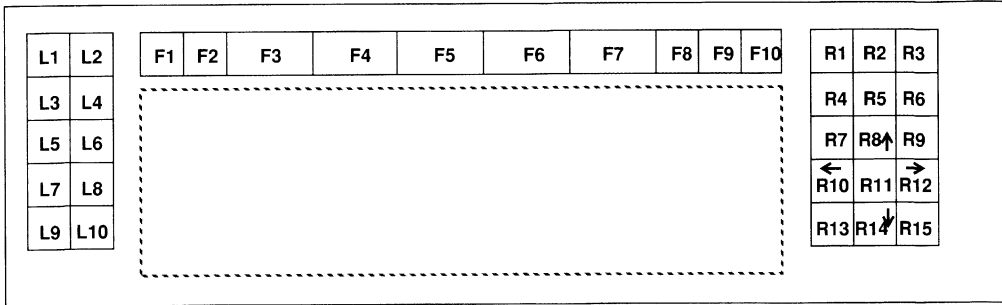
Figure A-4. The VT100 Keyboard



InterBase Constant	Value Returned	Generated by
PYXIS_\$KEY_DELETE	127	Delete
PYXIS_\$KEY_UP	128	Up-arrow
PYXIS_\$KEY_DOWN	129	Down-arrow
PYXIS_\$KEY_RIGHT	130	Right-arrow
PYXIS_\$KEY_LEFT	131	Left-arrow
PYXIS_\$KEY_PF1	132	PF1
PYXIS_\$KEY_PF2	133	PF2
PYXIS_\$KEY_PF3	134	PF3
PYXIS_\$KEY_PF4	135	PF4
PYXIS_\$KEY_PF5	136	
PYXIS_\$KEY_PF6	137	
PYXIS_\$KEY_PF7	138	
PYXIS_\$KEY_PF8	139	
PYXIS_\$KEY_PF9	140	
PYXIS_\$KEY_ENTER	141	ENTER
PYXIS_\$KEY_SCROLL_TOP	146	
PYXIS_\$KEY_SCROLL_BOTTOM	147	

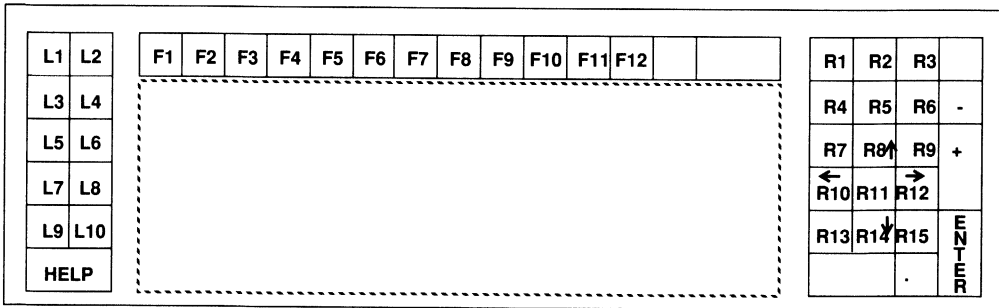
Keyboard Diagrams

Figure A-5. The Sun-3 and Sun-4 Keyboards



InterBase Constant	Value Returned	Generated by
PYXIS_\$KEY_DELETE	127	Delete
PYXIS_\$KEY_UP	128	R8
PYXIS_\$KEY_DOWN	129	R14
PYXIS_\$KEY_RIGHT	130	R12
PYXIS_\$KEY_LEFT	131	R10
PYXIS_\$KEY_PF1	132	R1
PYXIS_\$KEY_PF2	133	R2
PYXIS_\$KEY_PF3	134	R3
PYXIS_\$KEY_PF4	135	R4
PYXIS_\$KEY_PF5	136	R5
PYXIS_\$KEY_PF6	137	R6
PYXIS_\$KEY_PF7	138	R7
PYXIS_\$KEY_PF8	139	R9
PYXIS_\$KEY_PF9	140	R11
PYXIS_\$KEY_ENTER	141	R15
PYXIS_\$KEY_SCROLL_TOP	146	Control-T
PYXIS_\$KEY_SCROLL_BOTTOM	147	Control-B

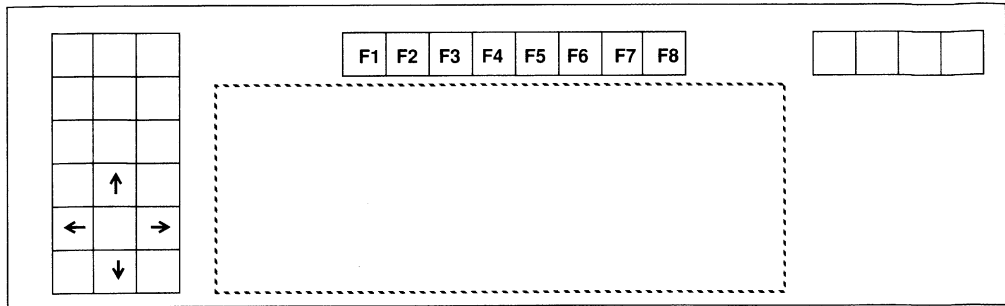
Figure A-6. The Sun 3861 Keyboard



InterBase Constant	Value Returned	Generated by
PYXIS_\$KEY_DELETE	127	Delete
PYXIS_\$KEY_UP	128	R8
PYXIS_\$KEY_DOWN	129	R14
PYXIS_\$KEY_RIGHT	130	R12
PYXIS_\$KEY_LEFT	131	R10
PYXIS_\$KEY_PF1	132	R1
PYXIS_\$KEY_PF2	133	R2
PYXIS_\$KEY_PF3	134	R3
PYXIS_\$KEY_PF4	135	R4
PYXIS_\$KEY_PF5	136	R5
PYXIS_\$KEY_PF6	137	R6
PYXIS_\$KEY_PF7	138	R7
PYXIS_\$KEY_PF8	139	R9
PYXIS_\$KEY_PF9	140	R11
PYXIS_\$KEY_ENTER	141	R15
PYXIS_\$KEY_SCROLL_TOP	146	Control-T
PYXIS_\$KEY_SCROLL_BOTTOM	147	Control-B

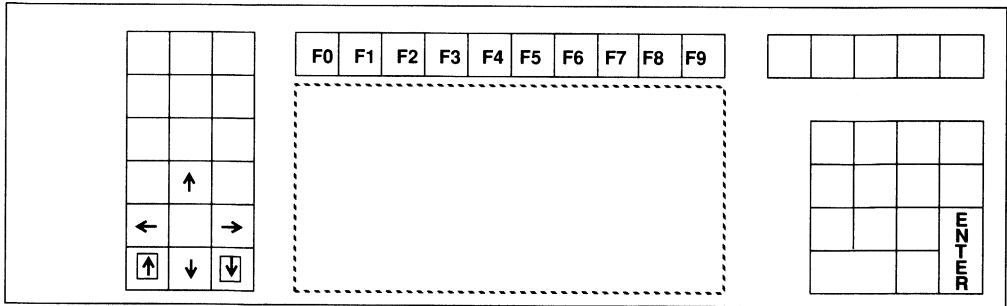
Keyboard Diagrams

Figure A-7. The Apollo DN3xx Keyboard



InterBase Constant	Value Returned	Generated by
PYXIS_\$KEY_DELETE	127	Delete
PYXIS_\$KEY_UP	128	Up-arrow
PYXIS_\$KEY_DOWN	129	Down-arrow
PYXIS_\$KEY_RIGHT	130	Right-arrow
PYXIS_\$KEY_LEFT	131	Left-arrow
PYXIS_\$KEY_PF1	132	F1
PYXIS_\$KEY_PF2	133	F2
PYXIS_\$KEY_PF3	134	F3
PYXIS_\$KEY_PF4	135	F4
PYXIS_\$KEY_PF5	136	F5
PYXIS_\$KEY_PF6	137	F6
PYXIS_\$KEY_PF7	138	F7
PYXIS_\$KEY_PF8	139	Shift-F1
PYXIS_\$KEY_PF9	140	Shift-F2
PYXIS_\$KEY_ENTER	141	F8
PYXIS_\$KEY_SCROLL_TOP	146	Control-T
PYXIS_\$KEY_SCROLL_BOTTOM	147	Control-B

Figure A-8. The Apollo DN3xxx Keyboard



InterBase Constant	Value Returned	Generated by
PYXIS_\$KEY_DELETE	127	Delete
PYXIS_\$KEY_UP	128	Up-arrow
PYXIS_\$KEY_DOWN	129	Down-arrow
PYXIS_\$KEY_RIGHT	130	Right-arrow
PYXIS_\$KEY_LEFT	131	Left-arrow
PYXIS_\$KEY_PF1	132	F1
PYXIS_\$KEY_PF2	133	F2
PYXIS_\$KEY_PF3	134	F3
PYXIS_\$KEY_PF4	135	F4
PYXIS_\$KEY_PF5	136	F5
PYXIS_\$KEY_PF6	137	F6
PYXIS_\$KEY_PF7	138	F7
PYXIS_\$KEY_PF8	139	Shift-F1
PYXIS_\$KEY_PF9	140	F9 or Shift-F2
PYXIS_\$KEY_ENTER	141	ENTER or F8
PYXIS_\$KEY_SCROLL_TOP	146	Control-T or Boxed-up-arrow
PYXIS_\$KEY_SCROLL_BOTTOM	147	Control-B or Boxed-down-arrow

Appendix B

The Atlas Database

About the Atlas Database

An InterBase database consists of an arbitrary number of relations, each containing an arbitrary number of fields. The database can also contain views, indexes, security classes, triggers, their supporting clauses, and the metadata for each of those objects.

The sample database used in the InterBase documentation is based on a North American atlas and gazetteer. The database consists of relations that represent:

- U.S. states (STATES) and Canadian provinces (PROVINCES).
- A sampling of North American cities (CITIES).
- Tourism offices for each of the states and provinces (TOURISM).
- Ski areas (SKI_AREAS).
- State populations (POPULATIONS).
- A selection of North American rivers (RIVERS) and some of the states through which they meander (RIVER_STATES).
- Mayors for cities in the CITIES relation (MAYORS), as of 1985. This relation includes the name, party affiliation, date of next election or date of original appointment for mayors and city managers in approximately one hundred cities.
- The population center for the United States every ten years since 1790 (POPULATION_CENTER).

About the Atlas Database

- Information about cross-country skiing areas in Massachusetts, Maine, and Vermont (**CROSS_COUNTRY**). This relation contains facts about trails, availability of various amenities, and a comment field that describes the ski area.
- Baseball teams and their stadiums, from both the American and National Leagues (**BASEBALL_TEAMS**).
- Population density for states (**POPULATION_DENSITY**), a view that divides the area of a state by its population for each of the last four censuses.
- Geographical data for cities (**GEO_CITIES**).
- The sample database also contains two views, subsets of one or more relations and other views.

Appendix C

Sample Forms Programs

Overview

This appendix provides two sample GDML programs that illustrate forms features. The features are highlighted and described. For information on specific syntax usage, refer to Chapter 7.

Sample Program 1

The following program illustrates the use of subforms and static menus. It displays a form, and the user types a state code. The program then displays the cities in that state along with their populations. The user can update the populations if so desired, and go on to choose another state. If the user does not enter a state code, the program asks whether it should commit the updates, and then exits.

The code for the following program is included with your software in the *examples* directory. The filename is *forms_city_pops.e*.

To run the example program, you must first preprocess, then compile the program. Refer to the chapter on preprocessing with **gpre** in the *Programmer's Guide*.

```
DATABASE DB = 'atlas.gdb'

main()
{
char answer;
short found;

/* Create forms window */
gds_$height = 20;
gds_$width = 80;
CREATE_WINDOW;

/* Open database and start transaction */
READY;
START_TRANSACTION;

/* Loop until user leaves form without filling in a state code */
found = 1;
while (1)
{
FOR FORM F IN CITY_POPULATIONS

/* Set instructional message to be displayed in form.
* If user just entered state code for a non-existent * state,
say so in the message.
*/
if (found)
{
strcpy (F.TAG,
"Enter State Code (enter nothing to
exit)");
```

```

        found = 0;
    }
else
    strcpy (F.TAG,
        "State not found; Enter State Code
        (enter nothing to exit)");

/* Display form and await entering of state code */
DISPLAY F DISPLAYING TAG ACCEPTING STATE
    if (F.STATE.STATE == PYXIS_$OPT_NULL) break;

/* Look for state */
FOR S IN STATES WITH S.STATE = F.STATE

    /* Note that state was found */
    found = 1;

    /* Put city information into subform */
    FOR C IN CITIES WITH C.STATE = S.STATE
        SORTED BY C.CITY
        PUT_ITEM FC IN F.CITY_POP_LINE
            strcpy (FC.CITY, C.CITY);
            FC.POPULATION = C.POPULATION;
        END_ITEM;

    END_FOR;

    /* Put state information into form */
    strcpy (F.STATE, S.STATE);
    strcpy (F.STATE_NAME, S.STATE_NAME);
    strcpy (F.TAG, "Update populations if
        needed");

    /* Display current form and allow
    * populations to be updated */
    DISPLAY F DISPLAYING STATE, STATE_NAME,
        CITY_POP_LINE.CITY,
        CITY_POP_LINE.POPULATION, TAG
        ACCEPTING CITY_POP_LINE.POPULATION;

    /* Perform modifications for any
    * updated populations */
    FOR_ITEM FC IN F.CITY_POP_LINE
        if (FC.POPULATION.STATE ==
            PYXIS_$OPT_USER_DATA)

```

Sample Program 1

```
        FOR C IN CITIES WITH C.CITY =
            FC.CITY
        AND C.STATE = F.STATE
        MODIFY C USING
        C.POPULATION = FC.POPULATION;
        END_MODIFY;
        END_FOR;
    END_ITEM;
END_FOR;

END_FORM;
}

/* Make form go away */
DELETE_WINDOW;

/* Check to see whether or not to commit updates */
printf ("Do you want to commit the updates (Y/N): ");
answer = getchar();
if ((answer == 'Y') || (answer == 'y'))
    COMMIT
else
    ROLLBACK;

/* Close down */
FINISH;
}
```

Sample Program 2

The following program illustrates the use of subforms and dynamic menus. It displays a menu of states from which the user picks one. The program then displays the cities in that state along with their populations. The user can update the populations if so desired, and go on to choose another state. If the user decides to exit, the program asks whether it should commit the updates, and then closes the window.

The code for the following program is included with your software in the *examples* directory. The filename is *forms_state_pops.e*.

To run the example program, you must first preprocess, then compile the program. Refer to the chapter on preprocessing with **gpre** in the *Programmer's Guide*.

```

DATABASE DB = 'atlas.gdb'

int * state_menu_handle;

main()
{
char answer;
short found;
char * valuep;

/* Open database and start transaction */
READY;
START_TRANSACTION;

/* Create the menu of state names. The value of each
 * entree is a pointer to the state code for that
 * entree */

FOR_MENU (MENU_HANDLE state_menu_handle) M

    PUT_ITEM E IN M
        strcpy (E.ENTREE_TEXT, "EXIT");
        E.ENTREE_LENGTH = strlen (E.ENTREE_TEXT);
        E.ENTREE_VALUE = 0;
    END_ITEM

    FOR S IN STATES SORTED BY ASCENDING S.STATE_NAME
        PUT_ITEM E IN M

```

Sample Program 2

```
        strcpy (E.ENTREE_TEXT, S.STATE_NAME);
        E.ENTREE_LENGTH = strlen (E.ENTREE_TEXT);
        valuep = (char *) malloc (strlen (S.STATE)
            + 1);
        strcpy (valuep, S.STATE);
        E.ENTREE_VALUE = (long) valuep;
    END_ITEM
END_FOR

END_MENU

/* Create forms window */
gds_$height = 20;
gds_$width = 80;
CREATE_WINDOW;

/* Loop until user selects EXIT from the States menu. */

while (1)
    {
    FOR FORM F IN CITY_POPULATIONS

        /* Set instructional message to be displayed
        * in form. */

        strcpy (F.TAG, "Choose a state (Choose EXIT
            when finished)");

        /* Display form and await selection of state */

        DISPLAY F DISPLAYING TAG NO_WAIT;

        FOR_MENU (MENU_HANDLE state_menu_handle) M
            strcpy (M.TITLE_TEXT, "States Menu");
            M.TITLE_LENGTH = strlen (M.TITLE_TEXT);

            DISPLAY M TRANSPARENT VERTICAL
            if (M.ENTREE_VALUE == 0)
                break;
            valuep = (char *) M.ENTREE_VALUE;
        END_MENU
    }
```

Sample Program 2

```
/* Look for state */

FOR S IN STATES WITH S.STATE = valuep

    /* Put city information into subform */
    FOR C IN CITIES WITH C.STATE = S.STATE
        SORTED BY C.CITY
            PUT_ITEM FC IN F.CITY_POP_LINE
                strcpy (FC.CITY, C.CITY);
                FC.POPULATION = C.POPULATION;
            END_ITEM;
        END_FOR;

    /* Put state information into form */
    strcpy (F.STATE, S.STATE);
    strcpy (F.STATE_NAME, S.STATE_NAME);
    strcpy (F.TAG, "Update populations if
        needed");

    /* Display current form and allow
        populations to be updated */
    DISPLAY F DISPLAYING STATE, STATE_NAME,
        CITY_POP_LINE.CITY,
            CITY_POP_LINE.POPULATION, TAG
        ACCEPTING CITY_POP_LINE.POPULATION;

    /* Perform modifications for any updated
        * populations */
    FOR_ITEM FC IN F.CITY_POP_LINE
        if (FC.POPULATION.STATE ==
            PYXIS_$OPT_USER_DATA)
            FOR C IN CITIES WITH C.CITY =
FC.CITY
                AND C.STATE = F.STATE
                MODIFY C USING
                    C.POPULATION =
FC.POPULATION;
            END_MODIFY;
        END_FOR;
    END_ITEM;
END_FOR;

END_FORM;
```

Sample Program 2

```
    }

    /* Deallocate the entree values for the menu, except
       the special EXIT entree. */

    FOR_MENU (MENU_HANDLE state_menu_handle) M

        FOR_ITEM E IN M
            if (E.ENTREE_VALUE != 0)
                free (E.ENTREE_VALUE);
            END_ITEM

    END_MENU

    /* Make form go away */
    DELETE_WINDOW;

    /* Check to see whether or not to commit updates */

    printf ("Do you want to commit the updates (Y/N): ");
    answer = getchar();
    if ((answer == 'Y') || (answer == 'y'))
        COMMIT
    else
        ROLLBACK;

    /* Close down */
    FINISH;
}
```


A

Apollo

fred implementation A-1

fred keyboard diagram A-10

atlas.gdb database B-1

B

Blob

using with forms 8-1

C

case_menu 7-10, 9-2

change 3-12

create_window 7-4

D

delete_window 7-5

display

forms 9-5

Dynamic menus 7-12

E

end_item 9-17

F

for_form 7-2, 9-9

for_item 9-12

for_menu 9-14

Forms

adding database fields 3-9

adding external fields 3-7

adding text 3-6

attributes 7-6

changing text 3-11

creating a form 2-1

creating menus 7-10

creating windows 7-4

datatypes valid 2-3

default elements 2-3

defining static menus 7-10

deleting a window 7-5

deleting elements 1-8, 3-13

displaying 7-2

editing 3-1

editing a blob 8-1

editing fields in 2-5

elements in 1-2

exiting 3-15

field values limiting 3-12

fields on 2-3

formatting in QLI 5-3

horizontal 9-2, 9-7

introduction 1-1

invoking 1-7

invoking in QLI 6-3

keyboard maps A-5

menus 1-7, 7-10

modifying data with 6-5

moving elements 3-4

navigating in 2-4

options for defining fields 3-8

overview 1-1

reformat 2-7

renaming 2-8

repeating groups 3-10

resizing 2-7

reversing 3-12

rolling back 1-8

sample application 7-13

sample programs C-1

saving 2-8, 3-14

saving to external file 2-9

selecting fields 3-4

setting attributes with GDML 7-6

size 2-7

starting 1-7

storing data using 6-5

tag line 2-3

terminating key 7-9

tutorial 5-2

using in programs 7-1

using subforms in GDML 7-13

using with GDML 7-1

using with QLI 6-1

- using with SQL 7-1
- valid field datatypes 2-3
- vertical 9-2, 9-7
- window 7-4

M

- Menu 1-7, 7-10
- Mouse support on Apollo A-1

O

- on_error** 7-17
- opaque** 9-7
- overriding** 9-6

P

- put_item**
 - described 9-17
- Pyxis
 - relation to forms 1-3

S

- Sample forms programs C-1
- set form** 6-3
- Static menus 7-10
- store**
 - using 6-5
- Subforms
 - changing 4-4
 - characteristics 4-4
 - editing 4-1
 - exiting 4-6
 - GDML 7-13
 - region 4-4
 - selecting 4-3
 - size 4-5
 - sub_item** 4-5
- Sun
 - fred** implementation A-4
 - fred** keyboard diagrams A-8

V

- VMS

- Index-2

- fred** keyboard diagrams A-6

W

- Window
 - delete** 7-5