This documentation and related computer software program (hereinafter referred to as the “Documentation”) is for
the end user's informational purposes only and is subject to change or withdrawal by Ingres Corporation ("Ingres")
at any time.

This Documentation may not be copied, transferred, reproduced, disclosed or duplicated, in whole or in part,
without the prior written consent of Ingres. This Documentation is proprietary information of Ingres and protected
by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of this Documentation for
their own internal use, provided that all Ingres copyright notices and legends are affixed to each reproduced copy.
Only authorized employees, consultants, or agents of the user who are bound by the confidentiality provisions of
the license for the software are permitted to have access to such copies.

This right to print copies is limited to the period during which the license for the product remains in full force and
effect. The user consents to Ingres obtaining injunctive relief precluding any unauthorized use of the
Documentation. Should the license terminate for any reason, it shall be the user’s responsibility to return to Ingres
the reproduced copies or to certify to Ingres that same have been destroyed.

To the extent permitted by applicable law, INGRES PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT
WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL INGRES BE LIABLE TO THE
END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USER OF THIS
DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR
LOST DATA, EVEN IF INGRES IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The use of any product referenced in this Documentation and this Documentation is governed by the end user's
applicable license agreement.

The manufacturer of this Documentation is Ingres Corporation.

For government users, the Documentation is delivered with "Restricted Rights" as set forth in 48 C.F.R. Section
12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013 or applicable successor
provisions.

Copyright © 2008 Ingres Corporation. All Rights Reserved.

Ingres, OpenROAD, and EDBC are registered trademarks of Ingres Corporation. All other trademarks, trade names,
service marks, and logos referenced herein belong to their respective companies.
## Chapter 3: Writing Scripts and Procedures

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scripts</td>
<td>67</td>
</tr>
<tr>
<td>Frame Scripts</td>
<td>67</td>
</tr>
<tr>
<td>Field and Menu Item Scripts</td>
<td>71</td>
</tr>
<tr>
<td>User Class Scripts</td>
<td>73</td>
</tr>
<tr>
<td>Statements in Scripts</td>
<td>74</td>
</tr>
<tr>
<td>Procedures</td>
<td>74</td>
</tr>
<tr>
<td>How You Can Create Procedures in OpenROAD</td>
<td>75</td>
</tr>
<tr>
<td>Global Procedures</td>
<td>76</td>
</tr>
<tr>
<td>Local Procedures</td>
<td>78</td>
</tr>
<tr>
<td>Database Procedures</td>
<td>79</td>
</tr>
<tr>
<td>3GL Procedures</td>
<td>80</td>
</tr>
<tr>
<td>Calling Procedures</td>
<td>83</td>
</tr>
<tr>
<td>How You Can Call 4GL Procedures</td>
<td>84</td>
</tr>
<tr>
<td>How You Can Call Database Procedures</td>
<td>85</td>
</tr>
<tr>
<td>How You Can Call Procedures in Expressions</td>
<td>86</td>
</tr>
<tr>
<td>How Procedure Handles Work (ProcHandle Objects)</td>
<td>87</td>
</tr>
<tr>
<td>How You Can Create a ProcHandle Object</td>
<td>87</td>
</tr>
<tr>
<td>How You Can Execute a ProcHandle Object</td>
<td>88</td>
</tr>
<tr>
<td>Restrictions to Using the Call Method</td>
<td>89</td>
</tr>
<tr>
<td>Global Procedures Available in the Core Library</td>
<td>89</td>
</tr>
<tr>
<td>Compiler Limitations</td>
<td>92</td>
</tr>
</tbody>
</table>

## Chapter 4: Programming Frames

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How You Can Invoke Frames</td>
<td>95</td>
</tr>
<tr>
<td>FrameExec Object</td>
<td>95</td>
</tr>
<tr>
<td>How You Can Transfer Control Between Frames</td>
<td>98</td>
</tr>
<tr>
<td>Differences Among the Frame-invoking Statements</td>
<td>100</td>
</tr>
<tr>
<td>How You Can Change a Frame's Definition</td>
<td>106</td>
</tr>
<tr>
<td>How You Can Block an Open Frame</td>
<td>106</td>
</tr>
<tr>
<td>Pop-up Frames</td>
<td>108</td>
</tr>
<tr>
<td>How You Can Use Ghost Frames</td>
<td>111</td>
</tr>
<tr>
<td>How You Can Run Non-interactive Applications</td>
<td>111</td>
</tr>
</tbody>
</table>

## Chapter 5: Working with Classes

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual Background</td>
<td>113</td>
</tr>
<tr>
<td>How You Can Reference Objects</td>
<td>114</td>
</tr>
<tr>
<td>How You Can Enhance Performance When Declaring Reference Variables</td>
<td>116</td>
</tr>
<tr>
<td>Field Function</td>
<td>117</td>
</tr>
<tr>
<td>How You Can Set and Get Attributes</td>
<td>117</td>
</tr>
<tr>
<td>How You Can Invoke Methods</td>
<td>119</td>
</tr>
</tbody>
</table>
Using an Object-oriented Approach to Programming

Inheritance

Class Relationships

User Class Hierarchies

How Polymorphism Works

How You Can Build an Inheritance Tree

How Invoking User Class Methods in 4GL Works

How You Can Create a User Class

How You Can Encapsulate Attributes and Methods

How You Can Write Methods

How You Can Use Polymorphism

Examples of Method: Hire Method

External Classes

ActiveX Support

How You Can Use External Classes

Dynamic Programming with External Objects

Non-Formfield External Objects

How You Can Use Collections with External Classes

Example—ActiveX Controls

How You Can Create Fields from User Classes

How You Can Manipulate Objects Generically: Casting

How You Can Work with Attributes

How You Can Work with Methods

Overview of the OpenROAD System Classes

Object

User Class Source Definition

User Class Runtime Definition

Source Definitions for Applications and Their Components

Running Applications and Their Components

Field and Menu Information

Event Information

Chapter 6: Working with a Database

Conceptual Background

How You Can Access a Database with Standard SQL Statements

Select Statement

Errors Originating in Called 3GL Procedures

How You Can Use the Insert Statement

How You Can Use the Update Statement

How You Can Use the Delete Statement

How You Can Use Other SQL Statements
Chapter 9: Working with Images and Text Strings  277
How You Can Work with Images ................................................................. 277
  Image Fields ............................................................................................ 278
  Image Trim ............................................................................................ 279
  Other Fields ......................................................................................... 279
  Palette Fields ....................................................................................... 280
  BitmapObject Class ............................................................................ 282
  ImageField Object ............................................................................. 286
How You Can Work with Text Strings ..................................................... 287
  How You Can Store Strings in the Database ......................................... 288
  How You Can Display a String from a Database .............................. 289
  How You Can Update Strings in a Database .................................... 290
  How You Can Delete Strings from a Database .................................. 291
  How You Can Create a String Storage Table ..................................... 292
  How You Can Determine If a Database is Unicode-enabled .............. 293

Chapter 10: Using 3GL in Your Application  295
How You Can Call 3GL Procedures .......................................................... 295
  Callproc Statement ........................................................................... 295
  How You Can Pass Parameters to 3GL Procedures ......................... 296
How You Can Use Exec 4GL Statements in 3GL Procedures .................. 298
  How You Can Check for Errors ........................................................... 298
  Summary of Exec 4GL Statements ....................................................... 299
  Example—Passing an Array to a 3GL Procedure ............................... 300
  Example—Sending a User Event from 3GL to 4GL ............................ 303
How You Can Use Exec SQL Statements in 3GL Procedures .................. 305
  How You Can Preprocess Exec 4GL and Exec SQL Statements ........ 305
  How You Can Link 3GL Procedures ................................................... 306
Technical Tips ......................................................................................... 306
  4GL/3GL Data Types ...................................................................... 306
Sample 3GL Procedures .......................................................................... 307
  3GL Sample .................................................................................... 307
  Exec 4GL and Exec SQL Sample ....................................................... 308

Chapter 11: Data Entry Error Handling  309
How You Can Use Data Entry Error Handlers ....................................... 309
  How You Can Change Values with a Data Entry Error Handler .......... 311
  How You Can Access a Hierarchy of Data Entry Error Handlers ......... 311
Chapter 12: Managing Event Queues 317

Conceptual Background ................................................................. 317
  How Events Are Executed .............................................................. 318
Event-based Programming.......................................................... 318
  Order of Events: Event Queues ..................................................... 319
  Event Types .................................................................................. 321
  How You Can Interrupt an Event Block ........................................ 328
  How You Can Obtain Information about the Current Event ......... 330

Chapter 13: Inter-Frame Communication Techniques 333

Communicating Between OpenROAD Frames ................................ 333
  User Events ............................................................................... 334
  Terminate Method ...................................................................... 334
  Global Variables ......................................................................... 335
  How You Can Communicate Between Frames Using User Events ... 336
  How You Can Communicate with an External Program Using External User Events .................. 347
    How You Can Register the External Event .................................. 348
    UserEvent Event Block ............................................................. 350
    How You Can Send the External User Event to OpenROAD ........ 351
    External User Event Demo ......................................................... 353
  How You Can Communicate Between OpenROAD Applications Using Database Events .......... 357
    How Communication Using a Database Event Works .................. 357
    DBEventObject ....................................................................... 358
    How Database Events Work with Multiple Sessions ................. 358
    How You Can Create, Register, and Raise Events ..................... 359
    How You Can Purge Database Events ........................................ 360
    How You Can Communicate Between Frames with Database Events ........................................ 361

Chapter 14: Creating Dynamic Frames 363

Conceptual Background ................................................................. 364
How You Can Change Field and Frame Appearance and Behavior ........................................ 365
  Biases for Fields ....................................................................... 366
  Biases for Menu Items ............................................................... 368
  How You Can Set Field and Menu Biases ..................................... 368
  How You Can Change Frame Modes ............................................ 370
  Frame Modes ........................................................................... 370
How You Can Create and Modify Fields Dynamically ......................................................... 373
  How You Can Create Dynamic Simple Fields ................................ 373
  How You Can Create Dynamic Composite Fields ....................... 375
How You Can Remove Fields Dynamically ............................................................................. 378
  How You Can Remove a Field from Display ................................ 378
How You Can Restore a Field to Display ................................................................. 379
How You Can Remove Columns from a Table Field .................................................. 379
How You Can Declare a Composite Field’s Array Variable Dynamically ...................... 380
How You Can Declare and Undeclare Data ............................................................... 380
How You Can Change a Dynamic Table Field after Declaring Data .......................... 381
How You Can Manipulate Data in Dynamic Fields ...................................................... 382
How You Can Set and Get Simple Field Values .......................................................... 382
How You Can Populate a Dynamic Table Field ....................................................... 384
How You Can Access Individual Cells of a Dynamic Table Field .............................. 385
How You Can Manage Operations with a Control Button ........................................... 389

Chapter 15: Creating a Frame at Runtime 391
Conceptual Background .............................................................................................. 391
How You Can Build a Frame Dynamically ............................................................... 392
  How You Can Create the Frame and Set Attributes .................................................. 393
  How You Can Attach Fields to the Form ............................................................... 394
  How You Can Create the Menu ............................................................................. 394
  How You Can Create the Script .......................................................................... 395
  How You Can Create the FrameSource Object ..................................................... 403
  Close Menu Item ................................................................................................. 417
Dynamic Statements Considerations ........................................................................... 418
  How You Can Populate a FlexibleForm, StackField, or MatrixField Dynamically .... 418

Chapter 16: Writing a Template Assistant 419
Conceptual Overview ................................................................................................. 419
  Assistants .............................................................................................................. 419
  How You Can Attach an Assistant to a Template .................................................. 419
  How You Can Ensure Compatibility Between a Frame Assistant and ApplyTemplate .... 419
How You Can Write an Assistant ............................................................................. 420
  Example—Frame Template Assistant .................................................................... 421
  Example—Field Template Assistant .................................................................... 423
How You Can Ensure Compatibility Between the Assistant and ApplyTemplate .......... 425
  How the Frame Creation Process Works .............................................................. 425
  How You Can Use ApplyTemplate on a Frame .................................................... 426
  How the Field Creation Process Works ............................................................... 428
  How You Can Use ApplyTemplate on a Field ....................................................... 428
  Example—ApplyTemplate Compatibility ............................................................ 428
Chapter 17: Preprocessing Your 4GL Code

Preprocessor .................................................................................................................................. 433
Include Scripts ............................................................................................................................ 434
User-defined Macro Variables ................................................................................................. 435
Built-in Macro Variables .......................................................................................................... 436
The #define Statement ............................................................................................................ 437
The #undef Statement ........................................................................................................... 438
The #if - #endif Statement .................................................................................................. 438
The #ifdef – #endif Statement ............................................................................................ 439
The ifndef - #endif Statement .......................................................................................... 439
Error Processing ....................................................................................................................... 439
Example—Error Processing .................................................................................................... 440

Chapter 18: Debugging Your Application

How You Can Use the Debugger ............................................................................................. 441
Start and Stop the Debugger .................................................................................................... 442
How You Can Test an Application .......................................................................................... 443
Debug a Single Application ...................................................................................................... 443
Debug Multiple Applications .................................................................................................. 443
How Running in the Debugger Environment Works ................................................................ 444
Frame States ............................................................................................................................ 444
Transactions ........................................................................................................................... 444
Errors ...................................................................................................................................... 444
Application Threads and Thread Identifiers ........................................................................... 444
Synchronization of Concurrent Threads .................................................................................. 445
How You Can Use Break Conditions ..................................................................................... 445
Debug Tab ............................................................................................................................... 446
Break Conditions .................................................................................................................... 446
Source Breakpoints ................................................................................................................ 448
Event Break Conditions .......................................................................................................... 450
Error Break Conditions .......................................................................................................... 452
Call Break Conditions ............................................................................................................. 454
Method Invocation Break Conditions ...................................................................................... 455
Thread Start Breakpoints ......................................................................................................... 455
How You Can Log Trace Information .................................................................................... 456
Trace Log Porlet ..................................................................................................................... 458
How You Can Monitor a Running Application ....................................................................... 459
Current Thread Display .......................................................................................................... 459
Thread Map ............................................................................................................................. 459
Call Stack Window .................................................................................................................. 460
Event Queue Window .............................................................................................................. 462
How You Can Debug Database Queries ................................................................. 463
  How You Can Specify a Database Session .......................................................... 464
  Run a Query ........................................................................................................ 465
How Debugging an Application Works ............................................................... 465
  Open the Debugger for an Application .............................................................. 466
  How You Can Use the Debugger Window ......................................................... 467
  Script Display Icons ......................................................................................... 468
  Display a Different Script .................................................................................. 469
  How You Can Control Execution of Your Application ........................................ 470
  How You Can Examine Values of Expressions .................................................. 473
  How You Can View Object Variables ............................................................... 476
  Select Array Display Settings ............................................................................ 477
  Assign Values to Expressions ............................................................................ 477
  How You Can Use the Watch Facility ............................................................... 478
  How You Can Customize Function Keys and Tooltip Text ............................... 480
  Edit a Script ....................................................................................................... 481
Debug an Imaged Application with the MonImage Utility .................................... 482

Appendix A: SQL Syntax Summary ................................................................. 483
  Dynamic Names in Syntax ................................................................................. 483
  SQL Syntax Summary ...................................................................................... 484
    Knowledge Management Extension Syntax ................................................ 486
  OpenSQL Syntax Summary .............................................................................. 487
  Distributed Ingres Syntax Summary ............................................................... 487

Appendix B: Operating System Differences .................................................. 489
  How You Can Use File Specifications .............................................................. 489
  Operating System Commands .......................................................................... 489

Glossary ........................................................................................................... 491

Index ............................................................................................................... 501
Chapter 1: Introduction

This section contains the following topics:

- **In This Guide** (see page 13)
- **Intended Audience** (see page 14)
- **Conventions** (see page 15)
- **Documentation** (see page 16)
- **Your Support Options** (see page 17)

The *Programming Guide* provides an overview of how to program in OpenROAD® by describing basic tasks, such as structuring an application and interacting with the database.

**In This Guide**

Specific topics covered in this guide include:

- Language elements
- Writing scripts and procedures
- Programming frames
- Working with classes
  - This topic discusses how system and user classes make an object-oriented approach to programming possible.
- Working with arrays, table fields, collections, list views, tree views, images, and text strings
- Using 3GL in your application
  - This topic includes sample 3GL procedures.
- Handling data entry errors and managing event queues
  - This topic includes examples of different types of data entry error handlers.
- Inter-frame communication techniques
- Creating dynamic frames and creating frames at runtime
- Writing a template assistant
- Preprocessing your 4GL code
- Debugging your application
The appendixes in this guide provide a summary of SQL syntax and operating system differences.

**Note:** The term *Windows* refers to the Microsoft Windows operating system, including Windows NT, Windows 95, Windows 98, and Windows 2000. Unless specifically indicated, *Windows* refers to any available Microsoft Windows operating system supported by OpenROAD. For more information on supported platforms, see the readme.

### Intended Audience

Because OpenROAD is intended for developers who want to create database applications, this guide assumes you know:

- Basic programming concepts, including the principles of object-oriented programming (OOP)
- Basic relational DBMS concepts, such as databases, tables, and cursors
- Structured Query Language (SQL)
- How to use basic OpenROAD features such as creating frames, forms, procedures, user classes, and global variables

In addition, you must be familiar with your window manager and toolkit, including terminology, navigational techniques, and how to work with standard items, such as menus and dialogs. If you are not, see your window manager documentation before using OpenROAD.

### Database-related Examples

When illustrating concepts that require interaction with a database, we assume you are connected to an Ingres database. If you are using a DBMS other than Ingres, sample code involving database interaction might not be 100% compatible.

**Note:** In this guide, "Ingres" is used as a convention to refer to all releases of Ingres.
Conventions

The following terminology distinctions are made when dealing with language items:

- A *command* is an operation that you execute from an OpenROAD menu or at the operating system level.
- A *statement* is an operation that you place within a program or called procedure. Statements can be written in OpenROAD's fourth-generation language (4GL), a database query language (such as SQL), or a 3GL (like C or COBOL).

When representing language elements in discussion text, the following conventions are used:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Indicates constants (for example, TRUE, FALSE, and FB_DIMMED)</td>
</tr>
<tr>
<td>lowercase</td>
<td>Indicates 4GL keywords and statements (for example, begin keyword and callproc statement)</td>
</tr>
<tr>
<td><em>italic, lowercase</em></td>
<td>Indicates a variable name (for example, cursor_variable)</td>
</tr>
<tr>
<td>Mixed Case/Initial Capitalization</td>
<td>Indicates class names, attributes, methods, or events (for example, DataStream class, ClientData attribute, FetchRow method, and WindowResized event)</td>
</tr>
</tbody>
</table>

When representing syntax, the following conventions are used:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boldface</strong></td>
<td>Indicates keywords, symbols, or punctuation that you must type as shown</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Indicates a variable name for which you must supply an actual value—this convention is used in explanatory text, as well as syntax</td>
</tr>
</tbody>
</table>
**Convention** | **Usage**
--- | ---
*Italic, underline* | Indicates a variable name which can be used in a statement either dynamically (when you run the application) or statically (when you create the application)

[ ] (square brackets) | Indicates an optional item

{ } (curly braces) | Indicates an optional item that you can repeat as many times as appropriate

| (vertical bar) | Indicates a list of mutually exclusive items (that is, you can select only one item from the list)

The following example illustrates the syntax conventions:

```
select [all | distinct]
resultexpression {, resultexpression}
from tablename [corrname] {, tablename [corrname]}
[where searchcondition]
[group by columnname {, columnname}]
[having searchcondition]
```

The documentation available with this release of OpenROAD is listed in the Getting Started.

This guide often refers to the online help, which indicates the Language Reference Guide help, available from the OpenROAD Workbench Help menu. This guide is also available in PDF.
Your Support Options

Enterprise customers with active maintenance and support contracts have full access to Ingres Support, including telephone support and online use of our call tracking system and knowledge base, Service Desk. For Customer Support contact details, see http://ingres.com/support/contact.php.

If you have an active support contract and want to register for access to Service Desk (https://servicedesk.ingres.com), use the enrollment form at http://www.ingres.com/users/register.php. (Your six-digit Account Number/Site ID is required.)

If you do not have a support agreement for Ingres Corporation products and are interested in purchasing support, contact us at sales@ingres.com.

For more information about support options, visit http://ingres.com/support.php.

Free support is available from the Ingres Open Source Community. Community members may obtain assistance for Ingres Corporation products by registering with the Ingres Community Site and using the available tools. To register, go to http://community.ingres.com/forums/home.php and click Register in the upper right corner of the page.

The Community Forums also provide Ingres Open Source Community members the opportunity to ask questions and interact with other community members and Ingres Corporation technical staff. For more information, visit http://community.ingres.com/forums/index.php.
Chapter 2: Language Elements

This section contains the following topics:

- OpenROAD Names (see page 19)
- Variables (see page 25)
- Data Types (see page 35)
- Nulls (see page 35)
- Expressions (see page 36)
- System-defined Functions (see page 50)
- Comments (see page 65)

When you create an OpenROAD application, you use the OpenROAD 4GL language to write scripts and procedures.

Note: For more information about how to write scripts and procedures, see Writing Scripts and Procedures (see page 67).

OpenROAD Names

In OpenROAD, the following items can be named:

- Applications
- Classes
- Databases
- Database tables and columns
- External class libraries
- Form fields
- Frames
- Global constants
- Objects
- Parameters
- Procedures
- Templates
- User events
- User classes
- Variables
The rules for these names, also known as alphanumeric identifiers, are as follows:

- Names can contain up to 32 alphanumeric characters.
- Names can begin only with an alphabetic character or an underscore (_).
- Names can contain an underscore as well as other alphanumeric characters (including #, @, $, and 0 through 9).
- Names are not case sensitive.

**Dynamic Names**

A dynamic name is a name that you can use in a statement either dynamically (when you run the application) or statically (when you create the application). All dynamic names are underlined in the syntax examples in this guide. Two examples of dynamic names are table names and column names, for example:

```
update tablename set columnname= expression
{. columnname=expression} where current of cursor_variable
```

Both statically-specified and dynamically-specified dynamic names must follow these rules:

- The variable that you use as a dynamic name can be a global variable or any variable in the current frame or procedure.

  To differentiate between a variable used as a dynamic name and an alphanumeric identifier, put a dereferencing colon (:) before the dynamic name. A dereferencing colon preceding a dynamic name indicates that you are using the contents of a variable to supply the value for an OpenROAD name.

- If the variable is part of an object, put the colon before the full name, for example:

  ```
callframe :framearray[3].name;
  ```

  Do not place the colon in the middle of the name, for example:

  ```
callframe framearray[3].:name;
  ```

  Also, in all SQL statements, the colon is required before variable names, for example:

  ```
select from tbl where col1 = :value;
  ```

  Placing the colon before "value" makes the select compare the col1 column with the value of the variable value. If the colon were missing from the statement, the select instead would compare the col1 column with a column named "value."
Although you can use a nullable variable as a dynamic name, the variable cannot contain a null when it is used in an SQL statement or a 4GL statement because the null causes a runtime error.

The statement that contains the error may be executed, but the value used in place of the variable is zero if a number is expected or “$NULL$ERROR” if a character string is expected, which may cause a runtime error.

When using dynamic names that are specified statically, do not put quotation marks around the name unless the name is identical to an OpenROAD word.

For more information, see the Language Reference Guide online help for a list of these reserved 4GL keywords.

When using dynamic names that are specified dynamically, specify a varchar variable.

At runtime, OpenROAD substitutes the current value of the variable for the name.

The advantage of using a dynamically-specified dynamic name is that the values can be substituted at runtime. This behavior allows processing to be guided by user input or other runtime conditions.

**Name Resolution**

OpenROAD application components share the same name space. This means that OpenROAD identifies the component strictly by name and does not consider the context in which the name is used. These components are:

- Variables
- Named constants
- Frames
- Procedures
- Classes

When OpenROAD encounters a name, it uses the first component it finds that matches the name. The context in which the name is specified has no effect on the name resolution. For example, when you use a procedure name in a callproc statement, OpenROAD searches for any component with that name, not just a procedure.

OpenROAD searches for names in the following order:

1. Current event block, local procedure, or user class method
2. Current field script
3. Enclosing field scripts (which are field scripts for fields containing the current field script's field)
4. Current frame, global procedure, or user class script
5. Current application
6. Included applications (in the same order they were included)
7. Core library

The following flowchart illustrates this name search order and lists the individual components at each stage of the name search:
If the component belongs to a frame, procedure, method, or user class script, you cannot change the search order. However, if the component belongs to an application, you can direct OpenROAD to search only within a specific application by qualifying the component name with the application's name. The syntax is:

application_name!component_name

When you include the application name, OpenROAD searches the specified application for the component rather than using the usual search order. You can specify the current application, any included application, or the core library.

When you specify a procedure or frame name at runtime, the search order differs from the first four steps of the name order search.

**Dynamic Frame and Procedure Name Resolution**

The following four statements let you specify a frame or procedure name at runtime:

- Callframe
- Openframe
- Gotoframe
- Callproc

When you use one of these statements to specify a frame or procedure name, OpenROAD resolves the reference at runtime instead of at compile time. OpenROAD first searches the currently executing component (a frame, global procedure, or user class method script) for a local procedure with the dynamically specified name. If the search is not successful, OpenROAD then searches for a global component (a frame or procedure).

By default, the global search order for the resolution starts with the topmost running application, that is, the application with which the user began the session.

This feature means that a frame in an included application can call a frame that belongs to the including application. For example, if Application A includes Application B, then Application B can use a dynamic name to call a frame that is part of Application A.
The following code example uses three applications:

**Application A**

Contains frames A1, A2, and X. The application also includes Application B and C.

**Application B**

Contains frames B1, B2, and X. The application also includes Application C.

**Application C**

Contains frames C1, C2, and X.

Assume that each of the frames in each of these applications has the following variable assignments:

/* The variables are varchar variables */

```
  nm1 = 'A1';
  nm2 = 'B1';
  nm3 = 'C1';
  nm4 = 'X';
  nm5 = 'A!X';
  nm6 = 'C!X';
```

The following table displays dynamic and explicit name resolution. It also shows the behavior for a variety of callframe statements using the assumptions described in this section. It also illustrates the behavior differences between dynamic and explicit name resolution.

<table>
<thead>
<tr>
<th>Statement</th>
<th>From AppA</th>
<th>From AppB</th>
<th>From AppC</th>
</tr>
</thead>
<tbody>
<tr>
<td>callframe A1</td>
<td>calls A!A1</td>
<td>compile warning and runtime error</td>
<td>compile warning and runtime error</td>
</tr>
<tr>
<td>callframe :nm1</td>
<td>calls A!A1</td>
<td>calls A!A1</td>
<td>calls A!A1</td>
</tr>
<tr>
<td>callframe B1</td>
<td>calls B!B1</td>
<td>calls B!B1</td>
<td>compile error</td>
</tr>
<tr>
<td>callframe :nm2</td>
<td>calls B!B1</td>
<td>calls B!B1</td>
<td>calls B!B1</td>
</tr>
<tr>
<td>callframe C1</td>
<td>calls C!C1</td>
<td>calls C!C1</td>
<td>calls C!C1</td>
</tr>
<tr>
<td>callframe :nm3</td>
<td>calls C!C1</td>
<td>calls C!C1</td>
<td>calls C!C1</td>
</tr>
<tr>
<td>callframe X</td>
<td>calls A!X</td>
<td>calls B!X</td>
<td>calls C!X</td>
</tr>
<tr>
<td>callframe :nm4</td>
<td>calls A!X</td>
<td>calls A!X</td>
<td>calls A!X</td>
</tr>
<tr>
<td>callframe A!X</td>
<td>calls A!X</td>
<td>compile error</td>
<td>compile error</td>
</tr>
<tr>
<td>callframe :nm5</td>
<td>calls A!X</td>
<td>runtime error</td>
<td>runtime error</td>
</tr>
<tr>
<td>callframe C!X</td>
<td>calls C!X</td>
<td>calls C!X</td>
<td>calls C!X</td>
</tr>
<tr>
<td>callframe :nm6</td>
<td>calls C!X</td>
<td>calls C!X</td>
<td>calls C!X</td>
</tr>
</tbody>
</table>
Variables

OpenROAD variables contain or point to data that the application manipulates. This can be data that is displayed to the user or used solely in scripts and procedures.

In OpenROAD, a variable is associated either with the application (global variables) or with a specific frame, procedure, method, field script, or event block (local variables).

**Global variables** provide data that is pertinent to the entire application and are available for use in any script or procedure. You use the OpenROAD Workbench to declare a global variable for your application. For more information about declaring global variables, see the *User Guide*.

**Local variables** contain data that is associated with a specific frame, procedure, method, field script, or event block. Local variables include:

- Variables declared in the initialize statement for a frame or a field script
  These frame variables are not associated with fields and the information in them is not displayed to the user directly.

- Variables associated with field and menu items
  You declare these local variables implicitly when you create the fields on a form. The information in them is displayed directly on the form, or you can declare them at runtime as described in the *Language Reference Guide* online help. When a variable is associated with a field on the displayed form, setting its value updates the display to show the new value. Conversely, referring to the variable reflects the current setting of the displayed form, including updates performed by the user.

- Variables declared in a procedure, method, or event block definition
  These variables are not associated with fields, and the information in them is not displayed to the user directly. They are available for use only in the procedure, method, or event block that defines them.

There are three types of variables in OpenROAD:

- Simple variables
- Reference variables
- Dynamic array variables
The following illustrates these variable types.

**Simple Variable**

DIRECTOR

**Reference Variable**

FILM

**More than 1 reference variable can point to the same object:**

FILM

**Dynamic Array Variable**

MOVIES[1]

MOVIES[2]

MOVIES[3]

MOVIES[4]

**Simple Variables**

A simple variable is a single data item. It contains only one value. A simple variable can be any of the basic data types, such as integer or varchar, with the exception of table_key and object_key as described in Data Types (see page 35).
How You Can Declare Simple Variables

The syntax for declaring a simple variable is:

```
name = datatype [with null|not null] [not default|with default][with] default
defaultvalue)
```

The following example shows a local variable declaration for an integer variable:

```i = integer;
```

By default, all simple variables are nullable. If you do not want the variable to be nullable, you must include the not null clause in the declaration, for example:

```i = integer not null;
```

By default, all numeric simple variables are assigned a default value of zero, and all character simple variables are assigned a default value of the empty string (''). If you want to specify a different default value, you must include the default clause. The default value must be null, a literal, or one of the system constants defined in the Language Reference Guide online help, for example:

```i = integer not null default false;
```

The with null, not default, and with default (without a value) clauses are provided for syntactic compatibility with SQL, but they have no effect in OpenROAD.

How You Can Reference Simple Variables

To reference a simple variable, use the variable name. For example, to assign the value of Smith to the person variable, use the following statement:

```person = 'Smith';
```

Individual elements in a reference variable or an array can be simple variables, as shown in the illustration in Variables (see page 25). In this illustration, Film is a reference variable with three attributes. Each of these attributes is a simple variable. Similarly, Movies is an array and each individual element in the array is a simple variable.

In a 4GL script, you can use simple variables as reference or array variable elements in any context that you can use other simple variables, for example:

```film.director = 'Hitchcock';

if movies[].title = 'ANNIE HALL' then ...
```
In the first example, film.director is a simple variable that is an element of a reference variable. In the second, movies[].title is a simple variable that is an element of an array.

For information about referencing the individual elements in a reference or array variable, see Reference Variables (see page 28) and Dynamic Array Variables (see page 31).

Reference Variables

A reference variable is a variable that points to an object of a given class, letting you access the value of an object. An object is a compound data structure that holds values that you can manipulate. A reference variable does not store its own values. Instead, when you reference the reference variable, OpenROAD uses the values in the corresponding object. The object is made up of attributes, which can be simple variables, reference variables, or array variables.

The class of an object defines its attributes. OpenROAD includes both system classes and user-defined classes. For more information about the system classes, see the Language Reference Guide online help.

In your 4GL code, when you reference a reference variable, you can work with the entire object as a single unit. For example, you can retrieve a row from a database table and assign all values from the row to a single object and then pass the object to a called procedure. You can also work with the individual attributes in the object. For examples of each case, see How You Can Reference Reference Variables (see page 29).

How You Can Declare Reference Variables

To declare a reference variable, use the following syntax:

\[
name = class \ [ \ [with] \ \text{default null}]\]

The value of class can be any named user class or system class (for a complete list of system classes, see the Language Reference Guide online help). The following example declares a reference variable of the StringObject class:

\[
strobj = StringObject;
\]

All reference variables are nullable.
When you create a field on a form, you specify the variable's name and data type on the field's property inspector in the OpenROAD Workbench. Unless you specify otherwise, OpenROAD automatically declares the variable that you use to access the data in that field. If the field is a named composite field, such as a stackfield or subform, the variable is a reference variable and you may or may not define its data type (class).

If you do define the data type for the reference variable, the data type must be a system or user class name. You can choose to exclude fields for some of the attributes (this provides a way to keep additional undisplayed control information).

If you do not name the class when you create a composite field, OpenROAD creates an unnamed class that you can reference only within the frame that contains the composite field.

How You Can Reference Reference Variables

In OpenROAD, you can work with an object as a whole or with the attributes of the object. To manipulate the object as a single unit, use the name of the object's reference variable in your 4GL code.

For example, you can create a subform and specify its variable's class as a named user class. In your program you could then pass the values in the subform fields to a procedure simply by naming the variable as a procedure parameter.

In the following statement, customer is the name of the variable associated with a subform:

callproc update_cust (new = customer)

To access an individual attribute that belongs to an object, use the dot notation:

referencevariable_name.attribute_name

For example, assume that the subform previously described contained a field called address. To reference that field, you would use:

customer.address

In the following example, city and street are attributes of class addr, and address is a declared reference variable:

address.city = 'Dallas';
x = address.street;
callframe delete_old (address = address.city);
OpenROAD checks references to attributes by name and produces compiler errors if the variable's class does not contain those attributes, for example:

/* Declarations */
a = addr;
b = addr;
tmp_char = varchar(30);

/* Some valid references */
a.street = tmp_char;
tmp_char = a.city;

/* Some invalid references */
a.province;  /* ERROR at compile time
** because attribute 'province'
** is not defined for class 'addr' */

da = null;
d.a.street;  /* ERROR at runtime because 'a' is
** null, and does not reference
** an object */

** Null Reference Variables **

When you declare a reference variable without specifying default null, the variable points to an object that is automatically initialized with the default values given in the definition of the object. However, you can set a reference variable to null. A reference variable that is set to null does not point to any object. Whenever all reference variables for an object are redirected or set to null, OpenROAD frees the associated object, for example:

/* Declarations */
a = addr;
b = addr;
a = b;     /* The original object for a is
** freed; both a and b point to the
** same object */
a = null;  /* The object still has 1
** reference (b) */
b = null;  /* The object is freed */

As an alternative, you can specify default null on the declaration and a default object is not created.

You can check for a null reference variable with the is null operator, for example:

if addr is null then
   /* processing statements */
endif;

For more information about null reference variables, see Nulls in Expressions (see page 49).
How You Can Create Objects for Reference Variables

You may need to create an object for a reference variable. For example, dynamic programs can create fields or forms at runtime by using the Create method, defined for the Class class. The Create method returns an object of a specified class.

In the following example, the reference variable a is declared with a default value of null, and a new object is created for it:

```/* Declarations */
   a = addr default null;
   a = addr.Create(); /* Creates an ADDR object */
   a.street = 'main street';
   a.city = 'dallas';
```

For more information about the Create method, see the Language Reference Guide online help. Examples of using the Create method are provided in Creating a Frame at Runtime (see page 391).

Dynamic Array Variables

A dynamic array variable is a named set of rows. All the rows in the array are reference variables that point to objects of a given class (or any of its subclasses).

Rows in an array are numbered. Because you can add to or subtract rows from an array and the array automatically adjusts as you do so, the array is said to be dynamic.

Some arrays have rows with non-positive numbers. These are rows that are marked "deleted." Deleted rows are not visible when the array is displayed in a table field, but they are not yet actually removed from the array.

Each array is associated with an array object, that is, an object of the class ArrayObject. The methods that you can use to manipulate arrays are defined for this class as described in the Language Reference Guide online help.

The following subsections describe how to declare a dynamic array and briefly discuss referencing arrays in your 4GL code. For more information about referencing array components, manipulating arrays, and the relationship between arrays and table fields, see Working with Arrays, Table Fields, and Collections (see page 205).
How You Can Declare Dynamic Array Variables

To declare an array variable, specify any system class or user class for the objects associated with the variable. The syntax is:

\[ \text{name} = \text{array of class} \quad \text{[with]} \quad \text{default null} \]

The following code is an example:

\[ \text{emparray} = \text{array of Emp;} \]

When you declare an array, it is empty. OpenROAD does not populate the array automatically. There are a variety of methods that you can use to add rows to an array. For more information, see Working with Arrays, Table Fields, and Collections (see page 205).

How You Can Reference Dynamic Array Variables

You can reference an array as a whole unit or you can reference the individual elements of the array—a single row, a single column, or a single cell.

By referencing the array itself, you can work with the entire set of objects as a unit. For example, you can pass the entire array as a parameter to a procedure or you can retrieve a group of rows from a database table into a single array. You can also get information about the array and manipulate the array.

To reference an array as a unit, use the array name. For example, assume that movies is the name of an array and that you want to pass this array to the movie_list frame. You could use the following statement:

\[ \text{callframe movie_list (mv_fld = movies);} \]

By referencing an individual row, you can work with an individual object in the array. To reference an individual row, use the following syntax:

\[ \text{arrayname}[n] \]

where arrayname is the name of the array in which the row resides and n is the number of the row. For example, to pass one of the rows from the movies array to a procedure:

\[ \text{callproc update_row (mv_row = movies[1]);} \]
By referencing a column, you can work with one attribute across all the objects. The syntax is:

\[
\text{arrayname}[*].\text{columnname}
\]

where \text{arrayname} is the name of the array and \text{columnname} is the name of the column. References to array columns are most commonly used to detect events, such as Entry or Exit events, on the column when the array is displayed in a table field, for example:

\[
\text{on entry movies}[*].\text{title}...
\]

To reference an individual element of an array, use the following syntax:

\[
\text{arrayname}[n].\text{columnname}
\]

where \text{arrayname} is the name of the array, \text{n} is the number of the row, and \text{columnname} is the name of the column.

An individual element of an array can be a reference variable or an array. This means that arrays can contain nested objects or arrays. You can also reference the individual elements of these nested objects or arrays in your 4GL code. For a nested object, the syntax is:

\[
\text{arrayname}[n].\text{nestedobject}.\text{attribute}
\]

where \text{arrayname} is the name of the object, \text{n} is the number of the row, \text{nestedobject} is the name of the nested object, and \text{attribute} is any attribute of that nested object.

For a nested array, the syntax is:

\[
\text{arrayname}[n].\text{nestedarrayname}[x].\text{attribute}
\]

where \text{arrayname} is the name of the outer array, \text{n} is the row number in that array, \text{nestedarrayname} is the name of the nested array, \text{x} is the number of the row within the nested array, and \text{attribute} is any attribute of the nested array.

**Field and Menu Item Variables**

Field and menu item variables are variables that are associated with the value in a field or menu item. These variables are not associated with the object represented by the field or menu item. When you create a field or menu item in the OpenROAD Workbench, you generally specify a variable name on the Property Inspector. This variable is associated with the data displayed in the field. You can reference this variable in your 4GL code whenever you need to access the data in the field.
For example, assume that you create an entry field and enter the variable name as *title*. You can set the value in that field with the following statement:

```plaintext
title = 'Destry Rides Again';
```

It is not necessary to declare field and menu item variables in your scripts; OpenROAD declares them automatically when you create the field or menu item in the OpenROAD Workbench.

To reference the object represented by a field or menu item instead of its value, use the field function, as described in Field Function (see page 65), with the field or menu item variable. For example, the following statement changes the background color of the previously described entry field:

```plaintext
field(title).bgcolor = CC_ORANGE;
```

For more information about the field function, see Field Function (see page 65).

**How You Can Initialize Variables**

When OpenROAD starts an application, any global variables associated with the application are initialized. Similarly, when you call a frame or procedure, OpenROAD initializes the variables associated with the frame or procedure.

OpenROAD uses the default value of the variable as the initial setting. For field and menu items or global variables, you can define this value in the OpenROAD Workbench. If you do not, OpenROAD uses the system defaults. OpenROAD also uses the system default for variables declared in the initialize statement of a frame script.

For simple variables, the system defaults are zero for numeric data types, or the empty string ("") for non-nullable character data types. However, you can overwrite these defaults using the default clause, for example:

```plaintext
i = integer not null default 3;
```

For reference variables, OpenROAD creates an object of the declared class and sets the reference variable to reference that object. The attributes of this object are initialized to their default values (which are specified in the class definition). For attributes that are reference variables, you can set the initial value of the variable to null. In this case, no object is created for the reference variable.
Data Types

When you explicitly declare a variable, you must assign a data type to it. A simple variable can be any of the following data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Length (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar(n)</td>
<td>1 &lt;= n &lt;= 2000</td>
<td>A variable-length character string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Gateway-accessed databases only:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 &lt;= n &lt;= 4096</td>
</tr>
<tr>
<td>char(n)</td>
<td>1 &lt;= n &lt;= 2000</td>
<td>A fixed-length character string</td>
</tr>
<tr>
<td>float</td>
<td>8</td>
<td>A floating-point number</td>
</tr>
<tr>
<td>smallint</td>
<td>2</td>
<td>An integer number</td>
</tr>
<tr>
<td>integer</td>
<td>4</td>
<td>An integer number</td>
</tr>
<tr>
<td>date</td>
<td>12</td>
<td>A time-date stamp or time interval</td>
</tr>
<tr>
<td>money</td>
<td>8</td>
<td>Currency</td>
</tr>
<tr>
<td>decimal(p,s)</td>
<td>1 &lt; p &lt; 31</td>
<td>Decimal with precision and scale</td>
</tr>
<tr>
<td></td>
<td>0 &lt; s &lt; p</td>
<td></td>
</tr>
</tbody>
</table>

A reference or array variable can be any named user class or system class. For information about creating a named user class, see the User Guide. For information about the system classes, see the Language Reference Guide online help.

By default, all data types are nullable. You can use the not null clause to declare a data type non-nullable. However, a class data type is always nullable.

Nulls

A null is an undefined or unknown value. A null is not the same as a zero, a blank, or an empty string.

You can assign a null to any nullable variable by using the null constant. For example, assume that your program contains a nullable variable named age. To set this variable to null, use the following statement:

age = null;
However, you cannot use the following statement:

```plaintext
if x = null ...
```

Instead, you must use the following statement:

```plaintext
if x is null ...
```

If you set the value of a reference variable to null, the reference variable is no longer associated with an object. (OpenROAD frees any object that does not have at least one reference variable pointing at it.)

The `ifnull` function and the `is null` operator provide you with ways to handle nulls in expressions. For more information about the `is null` operator, see Is [Not] Null Operator (see page 46).

**Expressions**

*Expressions* are language constructs that resolve to a value, a set of values, or TRUE or FALSE. Expressions can contain a wide variety of language elements. For example, you can use:

- Literals
- Variables
- Named constants
- Values returned by procedures
- Arithmetic and string expressions
- Logical (boolean) expressions

In addition, OpenROAD lets you use all the operators and functions of the Ingres DBMS.

Expressions are used widely in 4GL. For example, the value of an expression can:

- Be assigned to an OpenROAD variable
- Be used as part of a search condition in a query statement
- Serve as a condition in an OpenROAD if or while statement

The following subsections describe the language elements that you can use in expressions.
Literals

There are two basic types of literals: string and numeric. OpenROAD also supports hexadecimal constants and the null constant.

String Literals

String literals are represented by a sequence of characters enclosed in single quotation marks, for example:

'J. J. Jones'
'Hendersonville'
'17-Aug-1988 10:00'

You can include a literal single quote in a string literal by typing two single quotes (for example, 'J. J. Jones''s').

Numeric Literals

Numeric literals include integers and floating-point numbers, for example:

1209
7.77

Do not enclose numeric literals in quotes.

You can specify literals of type money either as strings ('$10.50') or numbers (10.5).

Hexadecimal Constants

Hexadecimal constants are a special version of string literals. They represent characters expressed in hexadecimal notation and are most often used for nonprintable characters such as a newline character. Hexadecimal constants are single-quoted strings of hexadecimal digits (0–9 and A–F) preceded by the letter X, for example:

X'10A665B'
X'00FF'
The general syntax for hexadecimal constants is:

\[ X' n(n)' \]

where \( n \) represents two hexadecimal digits. For example, using the ASCII character set, the following command inserts the string “XYZ” followed by the carriage return character into col1 and a numeric into col2 of table1:

```sql
insert into table1 (col1, col2)
values (X'58595A0D', 500);
```

OpenROAD interprets the hexadecimal constants as follows:

- 58 = X
- 59 = Y
- 5A = Z
- 0D = carriage return

If you select the inserted row from table1 with the following code fragment, “XYZ” is displayed in col1:

```sql
select :col1 = col1 from table1
where col2 = 500;
```

**Null Constant**

A null constant is represented by the special keyword `null`. You use this keyword to assign a null to a nullable variable or database column. For example, the following statement assigns a null to the variable amount:

```sql
amount = null;
```

For more information about nulls, see Nulls (see page 35).

**Named Constants**

A named constant is a literal value to which you give a name. You can then use the name in place of the constant in any 4GL expression. You specify the value for the named constant when you create it with the OpenROAD Workbench. This constant is global to the application.

You must use the Constant Editor to change the value of a constant. Although you can reference the named constant in your 4GL code, you cannot change its value at runtime.
Constants let you substitute a brief name for a long phrase or value that is used in several places, for example, if you have encoded a set of values as integers for storage. Constants also provide a consistent reference for message strings, for example, in internationalizing products.

To reference a named constant from your 4GL code, use the constant name, for example:

```java
if salary > HISALARYVALUE then ...
```

For more information on using the Constant Editor, see the User Guide.

### System Variables

*System variables* are built-in variables that are available in all frames and scripts. OpenROAD provides the following system variables:

**CurEventScope**

Specifies the context of the currently executing event block. This is a local variable of type Scope. Referencing CurEventScope outside of an event block causes a compile-time error.

**CurScriptScope**

Specifies the context of the currently executing field script. This is a local variable of type Scope. Referencing CurScriptScope outside of a field script causes a compile-time error.

**CurFrame**

Specifies the current frame. This is a local variable of type FrameExec. For more information about using the CurFrame variable, see the Language Reference Guide online help.

**CurMethod**

Specifies the currently executing user class method. This is a local variable of type MethodExec.

**CurObject**

Specifies the user class object on which the currently executing user class method was invoked. This is a local variable of type UserObject.

**CurProcedure**

The currently executing procedure. This is a local variable of type ProcExec.
CurSession

 Specifies, for each application, a global variable of type SessionObject that contains information about the current runtime environment, such as whether the user's display supports color or what operating system is being used.

IIDBMSerror

 Specifies the error code returned as a DBMS-specific error number. This error code reflects the last SQL statement encountered for the current frame or procedure's current DBMS connection. If no error occurred, IIDBMSerror is set to 0.

For the associated generic error number, see the IIerrornumber system variable.

IIerrornumber

 Specifies the error code returned as a DBMS-independent error number. This error number reflects the last SQL statement encountered for the current frame or procedure's current DBMS connection. If no error occurred, IIerrornumber is set to 0.

For the associated DBMS-specific error code, see the IIDBMSerror system variable.

IIrowcount

 Specifies the number of rows affected by the last SQL statement encountered for the current frame or procedure's current DBMS connection. After select statements, IIrowcount is the number of rows selected. After insert statements, it is the number of rows inserted into the database. After delete or update statements, it is the number of rows affected.

Operators

OpenROAD supports these operators:

- Arithmetic operators
- The String operator
- Logical operators
- Comparison (boolean) operators
- The Like operator
- The Is Null operator

The following sections describe these operators.
Arithmetic Operators

The arithmetic operators in OpenROAD combine numeric expressions into new expressions. These operators are listed in the following table:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
</tbody>
</table>

The following are examples:

```
num_days = num_days + 30;
area = pi * r**2;
```

In addition to standard arithmetic, OpenROAD supports date arithmetic. For example, if start_date is a date field, OpenROAD can compute the result value for the following statement:

```
start_date = start_date + '2 days';
```

The precedence of operators in expressions is as follows (highest to lowest):

```
**
*, / 
+, -
```

Operators are processed from left to right to control equal precedence.

For example, the following expression has a value of 14 because the multiplication operator (*) has precedence over the addition operator (+):  

```
2 + 3 * 4
```

You can use parentheses to force alternate precedence. For example, placing parentheses around the expression from the previous example alters its value to 20:

```
(2 + 3) * 4
```

Take care when dividing either a float variable into an integer literal constant or an integer literal constant into a float variable, as the result is of the float type but rounded down to the nearest integer.
**String Operator**

The concatenation operator (+) joins string expressions together into new expressions, for example:

```javascript
answer = prompt
    'Please enter department for ' + ' name + ' : ';
```

**Logical Operators**

The logical operators AND, OR, and NOT join logical expressions into new expressions with a boolean value. The following truth tables show the result of comparisons made with these operators.

**Truth table for AND comparisons:**

<table>
<thead>
<tr>
<th></th>
<th>True</th>
<th>False</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>null</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Null</td>
<td>null</td>
<td>False</td>
<td>null</td>
</tr>
</tbody>
</table>

**Truth table for OR comparisons:**

<table>
<thead>
<tr>
<th></th>
<th>True</th>
<th>False</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>null</td>
</tr>
<tr>
<td>Null</td>
<td>True</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

**Truth table for NOT comparisons:**

<table>
<thead>
<tr>
<th></th>
<th>True</th>
<th>False</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>null</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>
The order of precedence for these operators, from highest to lowest, is as follows:

**NOT**

**AND**

**OR**

You can use parentheses to change this order. In the following example:

```sql
if (empnum > 0 or deptno > 0) and status != 3
```

OpenROAD first evaluates empnum and deptno and then evaluates status.

The following example illustrates the use of comparison and logical operators:

```sql
if empnum > 0 and status != 3 then
  callframe newemployee;
endif;
```

The newemployee frame is called only if both conditions are TRUE, that is, if the current value of empnum is greater than 0, and status has any value other than 3.

### Comparison (Boolean) Operators

Boolean expressions yield the boolean values TRUE, FALSE, or (with nullable expressions) null. 4GL includes the logical operators AND, OR, and NOT, and the following comparison operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>is null</td>
<td>Value is null</td>
</tr>
<tr>
<td>is not null</td>
<td>Value is other than null</td>
</tr>
</tbody>
</table>
Expressions

**Operator** | **Operator**
---|---
like | Value is an instance of pattern-matching string
not like | Value is not an instance of pattern-matching string

With null, the result of a comparison will be true, false, or null. The result of a comparison (except for a comparison of reference variables) is null when one or both operands of the expression are null.

In 4GL, the if and while statements and the where clause of query statements test the results of boolean expressions. For example, assume that status is a character field on a form. Whenever the boolean expression "status = 'n' " is TRUE, the following statement calls the NewProject frame:

```plaintext
if status = 'n' then
   callframe NewProject;
endif;
```

In if and while statements, if the result of an expression is null, the flow of control occurs exactly as if the boolean expression evaluates to FALSE. For examples, see Nulls in Expressions (see page 49).

Reference variables may be compared, but only for equality, inequality, is null, or is not null. When comparing two reference variables that both contain null for equality (or inequality), the result is TRUE or FALSE, not null. Similarly, when comparing two reference variables only one of which contains null for equality, the result is FALSE.

**Like Operator and Pattern Matching**

The like operator lets you compare two strings to see whether they resemble each other in specific ways. To use this operator, specify a pattern (a string of characters with special formatting characters) that specifies what the compared string must look like. If the string being compared is an instance of the pattern described by the pattern string, the expression evaluates to TRUE.

The syntax for the like operation is:

```plaintext
charvar [not] like pattern [escape escapechar]
```
charvar

Specifies a character string variable

pattern

Specifies a character string literal or character string variable that can include the following special characters:

- An underscore (_) matches any single character. For example, the pattern string, "_a", would produce the matching text strings "Xa," "aa" and "/a".

- A percent sign (%) matches any string of characters, regardless of length. For example, the pattern string, "Fred%", would produce the matching text strings of "Fred," "Frederick," and "Fred S. Smith, Ph.D."

The following statement uses "%" to test the value of "emp_name" to see if it starts with "Fred" and ends with a last name of "Smith":

    if emp_name like 'Fred%Smith' then
        msg = 'Found a guy matching the pattern';
    endif;

- Square brackets [], where each bracket is preceded by an escape character, which you define, match the corresponding character position with any of the characters in the bracketed string. The typical definition of an escape character is usually done with quotes.

The escape character can be used to escape itself, so no matter what character you select, you can use it as a character in the pattern by giving it twice, for example:

    name like '..%.%' escape '.'

This phrase matches any string beginning with a period (.) and ending with a percent sign.

escapechar

Specifies a character string literal or character string variable of length 1. It is part of the escape clause, [escape escapechar], which has two functions:

- You can use it to "escape" the special interpretation of the pattern matching characters "_" and "%." For example, the following fragment matches a string of any first character and an underline as the second character:

    name like '\_\_' escape '\'
You can use the escape clause with square brackets to match any of the characters you specify. Typically, square brackets are treated the same as any other regular characters. However, when preceded by the escape character, the brackets define a “match-any-of-these-characters” string.

In the following example, you could use the escape clause with square brackets to match all strings ending with X, Y, or Z:

```
name like '%[XYZ]' escape '\'
```

As another example:
```
name like '_[BC]C%' escape '\'
```

This pattern matches “ABC,” “ACC,” “FCC Fairness Doctrine” and does not match “FDC Yellow #42” or “Access” (because “cc” is not uppercase).

Finally, the following example matches any string beginning with a left bracket and whose second character is “1,” “2,” “3” or a right bracket:
```
name like '[[123]]' escape '\'
```

The escape character cannot be followed in the pattern by any character other than underscore, percent, left or right bracket, or another escape character.

Pattern matching characters have no effect in comparisons using the arithmetic operators. For example, consider a statement that begins as follows:
```
if name = 'Fred%' then ...
```

The comparison tests the value of “name” to see whether it equals the constant literal “Fred%.” In the event that it does, the object of the condition is executed.

**Is [Not] Null Operator**

The is [not] null operator tests whether an expression is null. The syntax of this operator is:
```
expression is [not] null
```

The following code is an example:
```
if salary is null then
    sal_msg = 'Salary amount is unknown.'
endif;
```
Variables in Expressions

You can use simple variables alone or in expressions. In the following example, the age variable appears alone on the left side and is an element in the expression "age + 1" on the right of the assignment statement:

\[ \text{age} = \text{age} + 1; \]

OpenROAD uses the current value of age to compute a new value that replaces the current value.

You can use the simple variables that are elements of reference or array variables in the same way, for example:

\[
\begin{align*}
\text{total} &= 0; \\
\text{i} &= 1; \\
\text{while } &\text{i} \leq \text{emptable.LastRow()} \text{ do} \\
&\quad \text{total} = \text{total} + \text{emptable[i].salary}; \\
&\quad \text{i} = \text{i} + 1; \\
\text{endwhile};
\end{align*}
\]

In this example, the value in the salary column from the emptable table field is added to the total field as part of a while loop.

Procedures in Expressions

When a procedure returns a value, such as a return code, it functions as an expression. In the following syntax example, procname is the name of a procedure that returns a value:

\[ \text{returnval} = \text{procname()} + 1; \]

When you call a procedure as part of an expression, the following rules apply:

- Do not use the callproc keyword.
- Include the parentheses even if you are not passing any parameters to the procedure.
- Explicitly name the procedure. You can use a variable for the name of a procedure if you assign it to a variable. For example, the following assignment is legal:
  \[ \text{a} = :\text{varproc}(); \]

However you cannot use operators when you are assigning a variable as the procedure name. For example, the following expression is illegal and cannot be specified at runtime:

\[ \text{a} = :\text{varproc()} + 7 \]
Because procedures can return a value of any type, such as an object or an array, you can operate on the return value with any operation appropriate to the return value type. In the syntax example just described, procname is a procedure that returns a numeric type like integer or float, allowing you to use the return value in the addition.

If a procedure returns an object, the return value can be manipulated like any other object. That is, you can use the dot operator (.) to access individual attributes, or you can apply a method to that object.

Using the dot operator produces a variable of some kind (depending on the type of the attribute), so you can use the resulting variable wherever other variables can be used. For example, if the addr_proc procedure returns a variable of class ADDR that has an attribute “city,” the following expression is legal:

addr_proc().city = 'New York';

Because you can apply methods to the return value, and because methods can return values of any type, referencing can be nested on a procedure's return value, for example:

ret_framexec().objectsource.duplicate().name

### Methods in Expressions

The 4GL statement, method invocation, lets you invoke a method as part of an expression. For more information about this statement, see the Language Reference Guide online help.

When a method returns a value, it can function as an expression. In the following syntax example, methodname is the name of a method that returns a value:

```plaintext
returnval = objectref.methodname() + 1;
```

When you call a method as part of an expression, you must explicitly name the method. You can use a variable for the name of a method if you assign it to a variable. For example, the following assignment is legal:

```plaintext
a = objectref.:varmeth();
```

However you cannot use operators when you are assigning a variable as the method name. For example, the following expression is illegal and cannot be specified at runtime:

```plaintext
a = objectref.:varmeth() + 7
```
Because methods can return a value of any type, such as an object or an array, you can operate on the return value with any operation appropriate to the return value type. In this syntax example, methodname is a method that returns a numeric type like integer or float, allowing you to use the return value in the addition.

If a method returns an object, the return value can be manipulated like any other object. That is, you can use the dot operator (.) to access individual attributes, or you can apply a method to that object.

Using the dot operator gets a variable of some kind (depending on the type of the attribute), so you can use the resulting variable wherever other variables can be used. For example, if the addr_meth method returns a variable of class ADDR that has an attribute of city, the following expression is legal:

```
objectref.addr_meth().city = 'New York';
```

### Nulls in Expressions

Because a null cannot be compared to another value, the only test that you can perform is to see whether it is null or not. To perform this test, use the is null or is not null operator in a conditional expression. For more information, see Is [Not] Null Operator (see page 46).

If any item (other than a reference variable) in an expression has a null value, the value of the entire expression is null, for example:

```
msg = varchar (empno) +
     ' is not a valid employee number';
```

In this expression, if the variable empno has the value null, msg is null after the statement executes.

If any of the simple variables is null, the result of any comparison involving them is null, for example:

```
count = null;
if count + 1 > 0 then
    callframe newproject;
endif;
```

Because count is null, the result of the comparison that includes count is null. Therefore, the callframe statement is never executed.
This rule holds true for more complicated expressions, as in the following statements:

```
man_days = varchar(days) + 'days';
if (start_date + man_days) > 'today' then
    /* processing statements */
endif;
```

If start_date or man_days is null, the entire boolean expression evaluates to null and the processing statements are never performed.

**Expressions in SQL Statements**

When using expressions in SQL statements in OpenROAD, the operands of the functions and operators (or the entire expression) may be literals or database column names. In contexts where correlation names are defined (for example, with the select statement), a column name may be preceded by a correlation name (separated from the column name by a period).

You can replace any literal with a named constant or simple variable name (or a reference to a simple variable that is an attribute of a reference variable or an attribute of a row of an array variable) preceded by a colon. However, the field function is not allowed in database statements, nor are procedure or method invocations. SQL function and operators are allowed (they are evaluated by the database).

**System-defined Functions**

OpenROAD supports the following functions:

- All SQL functions except for the table_key and object_key scalar functions
- A non-SQL function, the field function

There are four types of SQL functions:

- Scalar functions, which take single-valued expressions as their arguments, can be used in all OpenROAD statements and in all SQL statements that are used within OpenROAD.
- Aggregate functions, which take a set of values (for example, the contents of a column in a table) as their arguments, can only be used in OpenROAD within SQL statements.
- The ifnull function
- The dbmsinfo function

These functions are described in the following subsections.
Scalar Functions

There are four types of scalar functions:

- data type conversion
- numeric
- string
- date

The scalar functions require either one or two single-value arguments. Scalar functions can be nested to any level. Scalar functions can be used in the select, insert, update, delete, while, and if statements.

The following subsections describe the scalar functions.

Data Type Conversion Functions

OpenROAD provides the following data type conversion functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Operand Type</th>
<th>Result Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c(expr)</code></td>
<td>any</td>
<td>c</td>
<td>Converts any value to a string</td>
</tr>
<tr>
<td><code>char(expr)</code></td>
<td>any</td>
<td>char</td>
<td>Converts any value to a char string</td>
</tr>
<tr>
<td><code>date(expr)</code></td>
<td>c, char, text, varchar</td>
<td>date</td>
<td>Converts a c, char, varchar, or text string to internal date representation</td>
</tr>
<tr>
<td><code>decimal(expr, 1 &lt; p &lt; 31, 0 &lt; s &lt; p)</code></td>
<td>c, char, varchar, text, float, money, integer(1), smallint, integer</td>
<td>decimal</td>
<td>Returns the decimal representation of the argument string</td>
</tr>
<tr>
<td><code>dow(expr)</code></td>
<td>date</td>
<td>c</td>
<td>Converts an absolute date into its day of week (for example, 'Mon,' 'Tue'); the result length is 3</td>
</tr>
<tr>
<td>Name</td>
<td>Operand Type</td>
<td>Result Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>float4(expr)</code></td>
<td><code>c, char, varchar, text, float, money, integer(1), smallint, integer</code></td>
<td><code>float4</code></td>
<td>Converts the specified expression to float4</td>
</tr>
<tr>
<td><code>float8(expr)</code></td>
<td><code>c, char, varchar, text, float, money, integer(1), smallint, integer</code></td>
<td><code>float</code></td>
<td>Converts the specified expression to float</td>
</tr>
<tr>
<td><code>hex(expr)</code></td>
<td><code>varchar, c, char, text</code></td>
<td><code>varchar</code></td>
<td>Returns the hexadecimal representation of the argument string; the length of the result is twice the length of the argument, because the hexadecimal equivalent of each character requires two bytes. For example, <code>hex('A')</code> returns '61' (ASCII) or 'C1' (EBCDIC).</td>
</tr>
<tr>
<td><code>int1(expr)</code></td>
<td><code>c, char, varchar, text, float, money, integer(1), smallint, integer</code></td>
<td><code>integer1</code></td>
<td>Converts the specified expression to integer1; floating point values are truncated</td>
</tr>
<tr>
<td><code>int2(expr)</code></td>
<td><code>c, char, varchar, text, float, money, integer(1), smallint, integer</code></td>
<td><code>smallint</code></td>
<td>Converts the specified expression to smallint; floating point values are truncated</td>
</tr>
<tr>
<td><code>int4(expr)</code></td>
<td><code>c, char, varchar, text, float, money, integer(1),</code></td>
<td><code>integer</code></td>
<td>Converts the specified expression to integer; floating point values are truncated</td>
</tr>
</tbody>
</table>
System-defined Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Operand Type</th>
<th>Result Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallint, integer</td>
<td>smallint, integer</td>
<td>money</td>
<td>Converts the specified expression to internal money representation; rounds floating point values, if necessary</td>
</tr>
<tr>
<td>money(expr)</td>
<td>c, char, varchar, text, float, integer(1), smallint, integer,</td>
<td>money</td>
<td></td>
</tr>
<tr>
<td>text(expr)</td>
<td>any</td>
<td>text</td>
<td>Converts any value to a text string; this function removes any trailing blanks from c or char string expressions</td>
</tr>
<tr>
<td>varchar(expr)</td>
<td>any</td>
<td>varchar</td>
<td>Converts any value to a varchar string; this function removes any trailing blanks from c or char string expressions</td>
</tr>
</tbody>
</table>

Numeric Functions

OpenROAD provides the following numeric functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Operand Type</th>
<th>Result Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(n)</td>
<td>all numeric types and money</td>
<td>same as n</td>
<td>absolute value of n</td>
</tr>
<tr>
<td>atan(n)</td>
<td>all numeric types and money</td>
<td>float</td>
<td>arctangent of n; returns a value from (-pi/2) to pi/2</td>
</tr>
<tr>
<td>cos(n)</td>
<td>all numeric types and money</td>
<td>float</td>
<td>cosine of n; returns a value from -1 to 1</td>
</tr>
<tr>
<td>exp(n)</td>
<td>all numeric types and money</td>
<td>float</td>
<td>exponential of n</td>
</tr>
<tr>
<td>log(n)</td>
<td>all numeric types and money</td>
<td>float</td>
<td>natural logarithm of n</td>
</tr>
<tr>
<td>mod(n,b)</td>
<td>integer, smallint, integer1</td>
<td>same as b</td>
<td>n modulo b; the result is the same data type as b</td>
</tr>
</tbody>
</table>
System-defined Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Operand Type</th>
<th>Result Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(n)</td>
<td>all numeric types and money</td>
<td>float</td>
<td>sine of n; returns a value from -1 to 1</td>
</tr>
<tr>
<td>sqrt(n)</td>
<td>all numeric types and money</td>
<td>float</td>
<td>square root of n</td>
</tr>
</tbody>
</table>

For trigonometric functions atan(), cos(), and sin(), you must specify arguments in radians. To convert degrees to radians, use the following formula:

\[ \text{radians} = \frac{\text{degrees}}{360} \times 2 \times \pi \]

To obtain a tangent, divide sin() by cos().

String Functions

The string functions operate on c, char, text, or varchar data.

The string functions can be nested to achieve other string functions, for example:

```sql
left(right(x.name, size(x.name) - 1), 3)
```

returns the substring of "x.name" from character positions 2 through 4.

The following string function:

```sql
concat(concat(x.lastname, ', '), x.firstname)
```

concatenates "x.lastname" with a comma and a blank and then concatenates "x.firstname" with the first concatenation result. You can also use the + operator to concatenate strings as follows:

```sql
x.lastname + ', ' + x.firstname
```

The following table lists the string functions supported by OpenROAD. The expressions c1 and c2 represent the arguments and can be any of the string types, except where noted. The expressions len and nshift represent integer arguments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Result Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concat(c1,c2)</td>
<td>any character</td>
<td>Concatenates one string to another. The result size is the sum of the sizes of the</td>
</tr>
<tr>
<td>Name</td>
<td>Result Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>data type</strong></td>
<td>two arguments. If the result is a c or char string, it is padded with blanks to achieve the proper length. To determine the data type results of concatenating strings, see String Concatenation Results (see page 57).</td>
<td></td>
</tr>
<tr>
<td>left((c1,len))</td>
<td>any character data type</td>
<td>Returns the leftmost len characters of c1. If the result is a fixed-length c or char string, it is the same length as c1, padded with blanks. The result format is the same as c1.</td>
</tr>
<tr>
<td>length((c1))</td>
<td>smallint</td>
<td>If c1 is a fixed-length c or char string, returns the length of c1 without trailing blanks. If c1 is a variable-length string, returns the number of characters actually in c1.</td>
</tr>
<tr>
<td>locate((c1,c2))</td>
<td>smallint</td>
<td>Returns the location of the first occurrence of c2 within c1, including trailing blanks from c2. The location is in the range 1 to size(c1). If c2 is not found, the function returns size(c1) + 1. (The function size() is described in this table.) If c1 and c2 are different string data types, c2 is coerced into c1’s data type.</td>
</tr>
<tr>
<td>lowercase((c1))</td>
<td>any character data type</td>
<td>Converts all uppercase characters in c1 to lowercase.</td>
</tr>
<tr>
<td>pad((c1))</td>
<td>text or varchar</td>
<td>Returns c1 with trailing blanks appended to c1. For example, if c1 is a varchar string that could hold 50 characters but only has two characters, then pad((c1)) appends 48 trailing blanks to c1 to form the result.</td>
</tr>
<tr>
<td>right((c1,len))</td>
<td>any character data type</td>
<td>Returns the rightmost len characters of c1. Trailing blanks are not removed first. If c1 is a fixed-length character string, the result is padded to the same length as c1. If c1 is a variable-length character string, no padding occurs. The result format is the same as c1.</td>
</tr>
<tr>
<td>Name</td>
<td>Result Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>shift</strong> (<em>c</em>&lt;sub&gt;1&lt;/sub&gt;, <em>nshift</em>)</td>
<td>any character data type</td>
<td>Shifts the string <em>nshift</em> places to the right if <em>nshift</em> &gt; 0 and to the left if <em>nshift</em> &lt; 0. If <em>c</em>&lt;sub&gt;1&lt;/sub&gt; is a fixed-length character string, the result is padded with blanks to the length of <em>c</em>&lt;sub&gt;1&lt;/sub&gt;. If <em>c</em>&lt;sub&gt;1&lt;/sub&gt; is a variable-length character string, no padding occurs. The result format is the same as <em>c</em>&lt;sub&gt;1&lt;/sub&gt;.</td>
</tr>
<tr>
<td><strong>size</strong> (<em>c</em>&lt;sub&gt;1&lt;/sub&gt;)</td>
<td>smallint</td>
<td>Returns the declared size of <em>c</em>&lt;sub&gt;1&lt;/sub&gt; without removing trailing blanks.</td>
</tr>
<tr>
<td><strong>squeeze</strong> (<em>c</em>&lt;sub&gt;1&lt;/sub&gt;)</td>
<td>text or varchar</td>
<td>Compresses white space (any sequence of blanks, null characters, newlines (line feeds), carriage returns, horizontal tabs, and form feeds (vertical tabs)). This function trims white space from the beginning and end of the string, and replaces all other white space with single blanks. This function is useful for comparisons. The value for <em>c</em>&lt;sub&gt;1&lt;/sub&gt; must be a string of variable-length character string data type (not fixed-length character data type). The result is the same length as the argument.</td>
</tr>
<tr>
<td><strong>trim</strong> (<em>c</em>&lt;sub&gt;1&lt;/sub&gt;)</td>
<td>text or varchar</td>
<td>Returns <em>c</em>&lt;sub&gt;1&lt;/sub&gt; without trailing blanks. The result has the same length as <em>c</em>&lt;sub&gt;1&lt;/sub&gt;.</td>
</tr>
<tr>
<td><strong>uppercase</strong> (<em>c</em>&lt;sub&gt;1&lt;/sub&gt;)</td>
<td>any character data type</td>
<td>Converts all lowercase characters in <em>c</em>&lt;sub&gt;1&lt;/sub&gt; to uppercase.</td>
</tr>
<tr>
<td><strong>charextract</strong> (<em>c</em>&lt;sub&gt;1&lt;/sub&gt;, <em>n</em>)</td>
<td>char</td>
<td>Returns the <em>n</em>th byte of <em>c</em>&lt;sub&gt;1&lt;/sub&gt;. If <em>n</em> is larger than the length of the string, then the result is a blank character.</td>
</tr>
</tbody>
</table>
String Concatenation Results

The following table shows the results of concatenating string expressions of various character data types:

<table>
<thead>
<tr>
<th>1st String</th>
<th>2nd String</th>
<th>Trim Blanks From 1st</th>
<th>Trim Blanks From 2nd</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>c</td>
<td>Yes</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>text</td>
<td>Yes</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>Yes</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>varchar</td>
<td>Yes</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>text</td>
<td>c</td>
<td>No</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>char</td>
<td>c</td>
<td>Yes</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>varchar</td>
<td>c</td>
<td>No</td>
<td>Yes</td>
<td>c</td>
</tr>
<tr>
<td>text</td>
<td>text</td>
<td>No</td>
<td>No</td>
<td>text</td>
</tr>
<tr>
<td>text</td>
<td>char</td>
<td>No</td>
<td>Yes</td>
<td>text</td>
</tr>
<tr>
<td>text</td>
<td>varchar</td>
<td>No</td>
<td>No</td>
<td>text</td>
</tr>
<tr>
<td>char</td>
<td>text</td>
<td>Yes</td>
<td>No</td>
<td>text</td>
</tr>
<tr>
<td>varchar</td>
<td>text</td>
<td>No</td>
<td>No</td>
<td>text</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>No</td>
<td>No</td>
<td>char</td>
</tr>
<tr>
<td>char</td>
<td>varchar</td>
<td>No</td>
<td>No</td>
<td>char</td>
</tr>
<tr>
<td>varchar</td>
<td>char</td>
<td>No</td>
<td>No</td>
<td>char</td>
</tr>
<tr>
<td>varchar</td>
<td>varchar</td>
<td>No</td>
<td>No</td>
<td>varchar</td>
</tr>
</tbody>
</table>

When concatenating more than two operands, OpenROAD evaluates expressions from left to right, for example: varchar + char + varchar is evaluated as (varchar+char)+varchar. To control concatenation results for strings with trailing blanks, use the trim, notrim, and pad functions.

Date Functions

OpenROAD supports functions that derive values from absolute dates and from interval dates. These functions operate on rows or variables that contain date values. An additional function, dow(), returns the day of the week (for example, mon, tue) for a specified date. The dow() function is described in Data Type Conversion Functions (see page 51).
Some date functions require you to specify a unit parameter; unit parameters must be specified using a quoted string. The following table lists valid unit parameters:

<table>
<thead>
<tr>
<th>Date Portion</th>
<th>How Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second</td>
<td>second, seconds, sec, secs</td>
</tr>
<tr>
<td>Minute</td>
<td>minute, minutes, min, mins</td>
</tr>
<tr>
<td>Hour</td>
<td>hour, hours, hr, hrs</td>
</tr>
<tr>
<td>Day</td>
<td>day, days</td>
</tr>
<tr>
<td>Week</td>
<td>week, weeks, wk, wks</td>
</tr>
<tr>
<td>Month</td>
<td>month, months, mo, mos</td>
</tr>
<tr>
<td>Quarter</td>
<td>quarter, quarters, qtr, qtrs</td>
</tr>
<tr>
<td>Year</td>
<td>year, years, yr, yrs</td>
</tr>
</tbody>
</table>

The following table lists the date functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Format (Result)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date_trunc(unit,date)</code></td>
<td>date</td>
<td>Returns a date value truncated to the specified <code>unit</code></td>
</tr>
<tr>
<td><code>date_part(unit,date)</code></td>
<td>integer</td>
<td>Returns an integer containing the specified (unit) component of the input date</td>
</tr>
<tr>
<td><code>date_gmt(date)</code></td>
<td>any character data type</td>
<td>Converts an absolute date into the Greenwich Mean Time character equivalent with the format <code>yyyy_mm_dd hh:mm:ss GMT</code>. If the absolute date does not include a time, Ingres returns blanks for the time portion of the result. For example, the query: <code>select date_gmt('1-1-93 10:13 PM PST')</code> returns the following value: <code>'1993_01_01 06:13:00 GMT'</code>.</td>
</tr>
</tbody>
</table>
### System-defined Functions

#### Language Elements

<table>
<thead>
<tr>
<th>Name</th>
<th>Format (Result)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval (unit, date_interval)</td>
<td>float</td>
<td>Converts a date interval into a floating-point constant expressed in the unit of measurement specified by unit. The interval function assumes that there are 30.436875 days per month and 365.2425 days per year when using the mos, qtrs and yrs specifications. For example, the query: select(interval('days', '5 years')) returns the following value: '1826.213'</td>
</tr>
<tr>
<td>_date(s)</td>
<td>any character data type</td>
<td>Returns a 9-character string giving the date s seconds after January 1, 1970 GMT. The output format is 'dd-mmm-yy'. For example, the query: select _date(123456) returns the following value: '2-jan-1970'</td>
</tr>
<tr>
<td>_time(s)</td>
<td>any character data type</td>
<td>Returns a 5-character string giving the time s seconds after January 1, 1970 GMT. The output format is 'hh:mm' (seconds are truncated). For example, the query: select _time(123456) returns the following value: '02:17'</td>
</tr>
</tbody>
</table>

### How You Can Use the Date_trunc Function

You can use the date_trunc function to group dates. The following expression is an example of grouping dates within the same month or year:

date_trunc('month', date('23-oct-1993 12:33'))

This expression returns "1-oct-1993."
The following expression:

\[
\text{date\_trunc('year',date('23-oct-1993'))}
\]

returns “1-jan-1993”.

Truncation takes place in terms of calendar years and quarters ("1-jan," "1-apr," "1-jun," and "1-oct").

To truncate in terms of a fiscal year, you must offset the calendar date by the number of months between the beginning of your fiscal year and the beginning of the next calendar year ("6 mos" for a fiscal year beginning July 1, or "4 mos" for a fiscal year beginning September 1):

\[
\text{date\_trunc('year',date+'4 mos') - '4 mos'}
\]

Weeks start on Monday. The beginning of a week for an early January date may fall into the previous year.

**How You Can Use the Date_part Function**

The date_part function is useful in set functions and in assuring correct ordering in complex date manipulation. For example, if date_field contains the value “23-oct-1993” then:

\[
\text{date\_part('month',date(date_field))}
\]

returns a value of 10 (representing October), and:

\[
\text{date\_part('day',date(date_field))}
\]

returns a value of 23.

The date_part function uses the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>months</td>
<td>Use numbers 1 to 12, starting with January</td>
</tr>
<tr>
<td>hours</td>
<td>Returned according to the 24-hour clock</td>
</tr>
<tr>
<td>quarters</td>
<td>Numbered 1 through 4</td>
</tr>
<tr>
<td>weeks</td>
<td>Week 1 begins on the first Monday of the year</td>
</tr>
<tr>
<td>dates</td>
<td>Dates before the first Monday of the year are considered to be in week 0</td>
</tr>
</tbody>
</table>
Aggregate Functions

An aggregate function returns a single value based on the contents of a column. Aggregate functions are also called set functions. These functions can be used in the select, insert, update, and delete SQL statements. You cannot use aggregate functions in if or while statements. Aggregate functions can only be used in OpenROAD within SQL statements.

The following example uses the sum aggregate function to calculate the total of salaries for employees in department 23:

```sql
select sum (employee.salary) 
    from employee 
    where employee.dept = 23;
```

The following table lists the OpenROAD aggregate functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Result Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>integer</td>
<td>Count of occurrences</td>
</tr>
<tr>
<td>sum</td>
<td>integer, float, money, date (interval only)</td>
<td>Column total</td>
</tr>
<tr>
<td>avg</td>
<td>float, money, date (interval only)</td>
<td>Average (sum/count)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The sum of the values must be within the range of the result data type.</td>
</tr>
<tr>
<td>max</td>
<td>same as argument</td>
<td>Maximum value</td>
</tr>
<tr>
<td>min</td>
<td>same as argument</td>
<td>Minimum value</td>
</tr>
</tbody>
</table>

The general syntax of an aggregate function is:

```
function_name ([distinct | all] expr)
```

where `function_name` denotes an aggregate function and `expr` denotes any expression that does not include an aggregate function reference (at any level of nesting).

To eliminate duplicate values, specify `distinct`; to retain duplicate values, specify `all`. (The default is `all`.) `Distinct` is not meaningful in conjunction with the functions min and max, because these functions return single values.

Nulls are ignored by the aggregate functions, with the exception of count, as described in Count Function and Nulls (see page 62).
Count Function and Nulls

The count function can take as an argument the wildcard character "*". This character is used to count the number of rows in a result table, including rows that contain nulls. For example, the following statement counts the number of employees in department 23:

```sql
select count(*)
  from employee
  where dept = 23;
```

You cannot qualify the asterisk (*) argument with all or distinct.

Because count(*) counts rows rather than columns, count(*) does not ignore nulls. Consider the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Exemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allen</td>
<td>0</td>
</tr>
<tr>
<td>Curtiz</td>
<td>2</td>
</tr>
<tr>
<td>Fleming</td>
<td>4</td>
</tr>
<tr>
<td>Wellman</td>
<td>null</td>
</tr>
<tr>
<td>Hitchcock</td>
<td>null</td>
</tr>
</tbody>
</table>

The statement:

```sql
count(exemptions)
```

returns the value “3”, whereas the statement

```sql
count(*)
```

returns “5”.

With the exception of count, if the argument to an aggregate function evaluates to an empty set, then the function returns a null. The count function returns a zero.

How You Can Use the Group By Clause with Aggregate Functions

The group by clause allows aggregate functions to be performed on subsets of the rows in the table. The subsets are defined by the group by clause.
For example, the following statement selects rows from a table of political candidates, groups the rows by party, and returns each party’s name and the average funding for the candidates in that party.

```
select party, avg(funding)
  from candidates
  group by party;
```

**Restrictions on the Use of Aggregate Functions**

The following restrictions apply to the use of aggregate functions:

- You cannot nest aggregate functions.
- You can use aggregate functions only in select or having clauses.
- If a select or having clause contains an aggregate function, columns not specified in the aggregate must be specified in the group by clause, for example:

```
select dept, avg(emp_age)
  from employee
  group by dept;
```

This select statement specifies two columns, dept and emp_age, but only emp_age is referenced by the avg aggregate function. The dept column is specified in the group by clause.

**The Ifnull Function**

The ifnull function enables you to specify a value, other than a null, that is returned to your application when a null is encountered. The ifnull function is specified as follows:

```
ifnull(v1, v2)
```

If the value of the first argument is not null, then ifnull returns the value of the first argument. If the first argument evaluates to a null, then ifnull returns the second argument.

For example, the sum, avg, max, and min aggregate functions return a null if the argument to the function evaluates to an empty set. To receive a value instead of a null when the function evaluates to an empty set, you can use the ifnull function, as in this example:

```
ifnull(sum(employee.salary)/25, -1)
```

Ifnull returns the value of the expression "sum(employee.salary)/25" unless that expression is null. If that expression is null, the ifnull function returns -1.
Result Data Type

If the arguments are of the same data type, then the result is of that data type. If the two arguments are of different data types, then they must be of comparable data types. When the arguments are of different but comparable data types, then OpenROAD uses the following rules to determine the data type of the result:

- The result type is always the higher of the two data types; the order of precedence of the data types is as follows:
  
  \[\text{date} > \text{money} > \text{float4} > \text{float} > \text{integer} > \text{smallint} > \text{integer1}\]
  
  and
  
  \[\text{c} > \text{text} > \text{char} > \text{varchar}\]

- The result length is taken from the longest value, for example:
  
  \[\text{ifnull (varchar (5), c10)}\]
  
  results in c10.

The result is nullable if either argument is nullable. The first argument is not required to be nullable, though in most applications it would be nullable.

Dbmsinfo Function

In OpenROAD, if you use the dbmsinfo function as a statement by itself, the only valid request_name is \_bintim:

\[\text{variable = dbmsinfo('\_bintim')}\]

All other values of request_name return blanks unless the function is part of a larger SQL statement. For example, the following statement returns a user name:

\[\text{select logname = dbmsinfo('username')}\]

However, the following statement returns blanks:

\[\text{logname = dbmsinfo('username')}\]
Field Function

The field function gives you access to the attributes of the object associated with a displayed field or menu item, rather than to the actual value of the field or menu item. For example, assume that you create a frame with a slider field. In the OpenROAD Workbench's Property Inspector for the slider field, you specify the field's variable as temperature. This variable is a simple integer variable. When you reference the variable in a program, the reference provides the current value in the slider field:

```plaintext
if temperature >= 85 then ...
```

In contrast, if you use the field function with the variable name, then the reference is to the object that the field represents, the SliderField object. Using the field function lets you read or change attributes of that object:

```plaintext
field(temperature).TypeFace = TF_LUCIDA;
```

This statement changes the type face of the value settings on the slider field.

For more examples and further discussion of the field function, see Creating Dynamic Frames (see page 363).

Comments

You can include comments anywhere in your scripts, 4GL procedures, and in 3GL procedures written in C. A comment is bounded by "/*" on the left and by "*/" on the right, for example:

```plaintext
/* This is a comment */
```

Comments can extend over more than one line, for example:

```plaintext
/*
** This is a multiline comment.
*/
```

You cannot nest comments.
Chapter 3: Writing Scripts and Procedures

This section contains the following topics:

- **Scripts** (see page 67)
- **Procedures** (see page 74)
- **Calling Procedures** (see page 83)
- **How Procedure Handles Work (ProcHandle Objects)** (see page 87)
- **Compiler Limitations** (see page 92)

This section explains how to write scripts and procedures, as well as the details of how to call procedures.

### Scripts

A *script* is a set of statements that define the actions performed by a procedure or frame. In OpenROAD, you can write scripts for the following:

- Frames
- Frame events
- Events for a field or menu item
- Procedures
- User class methods

Generally, every frame has at least one event block associated with it, in either a frame script or in an individual field or menu item script associated with the frame. If an interactive frame has no event blocks, then the frame is displayed but the user cannot interact with it. For more information about events, see the *Language Reference Guide* online help.

The following subsections describe these types of scripts. The last subsection, Statements in Scripts (see page 74), gives a general overview of the types of statements that you can include in your scripts.

### Frame Scripts

Each frame can have one frame script. In the frame script, you can do the following:

- Initialize local variables for the frame
- Perform any start-up operations for the frame
- Include event blocks for events that apply to the frame as a whole
- Include event blocks for events on individual fields or menu items

The general syntax of a frame script is:

```
[initialize ([parameterlist]) =
[declare
  [localvariablelist]
  [localprocedureforwardreferences]
[enddeclare]]
[begin
  statementlist
end [:]]
{eventblock [:]}
{localprocedure [:]}
{parameterlist [:]}
{localvariablelist [:]}
```

**parameterlist**

Specifies the following:

```
{variable = datatype {, variable = datatype}}
```

**localvariablelist**

Specifies the following:

```
{variable = datatype:}
```

The initialize statement contains the local variable definitions and start-up operations for the frame. The event blocks contain the code for the events for the frame, form fields, and menu items. You can include comments at any point in your script. For more information, see Comments (see page 65).

Most applications have a frame script for each frame. However, if the frame needs no initialize statement, you can create the field and menu scripts separately and omit the frame script.

**Initialize Statement**

When the optional initialize statement is used, it must be the first statement in the frame script.

The initialize statement can have a parameter list that declares the parameters of a frame. Variables declared as frame parameters may be specified as parameters of the callframe, openframe, and gotoframe statements. For more information about these statements, see the *Language Reference Guide* online help.
Declare local variables that are not parameters and forward references to local procedures in the declare section of the initialize statement. If a declare block is not followed by a begin block of initialization statements, enddeclare is required at the end of the local variable declarations in the declare block.

You can use local variables declared in the initialize statement anywhere in the frame script or in any of the field or menu item scripts associated with the frame.

A local variable can be a simple variable of any acceptable base data type or a reference or array variable of any named user or system class. For more information about declaring variables, see Language Elements (see page 19).

The local variables you declare with the initialize statement are in addition to the automatically declared variables that are associated with named fields and menu items. (You do not need to explicitly declare the variables associated with named fields and menu items.)

When you use the statement block, which is delimited by the keywords begin and end, OpenROAD executes the statements in this block when the frame is started, before displaying the frame on the window. You can use this statement block to perform any start-up operations, such as loading a table field or setting variables.

You may use the curly brackets { and } in place of the keywords begin and end.

**Frame Script Event Blocks**

A frame script can include one or more event blocks. These blocks can contain frame events or events for any field or menu item defined for the frame.

An event block is a sequence of statements associated with one or more specific events. The syntax for an event block is:

```plaintext
on event [variablename]
    {, on event [variablename]} =
    [declare
        localvariablelist
    [enddeclare]]
    begin
        statementlist
    end[:]
```

When the event occurs, OpenROAD executes the statements specified by the statementlist.
The following rules apply when coding event blocks:

- If the event is a field or menu item event, you must also specify the variable name associated with that field or menu item.

  For example, the following event block provides the code for the Click event for a field called close_button:

  ```plaintext
  on click close_button =begin
    parent_frame.SendUserEvent
    (eventname = 'INDICATOR_OFF',
     messageinteger = row_number);
    return;
  end;
  ```

- If the event is a frame event, do not include a variable name.

  For example, the following event block in a frame script is a frame event that is executed whenever the user exits any field on the frame form:

  ```plaintext
  on childexit =
    begin
      statementlist
    end;
  ```

- You can include more than one event type in an event block.

  OpenROAD activates the event block whenever any of the specified events occurs.

- Declare local variables in the declare section of an event block.

  The variables defined in this section are visible only in the event block.

- Separate the statements in the `statementlist` with semicolons.

  You can include as many statements as you want, and each can continue over as many lines as necessary. Also, you can include comments in the event block. For more information, see Comments (see page 65).

The following frame events are typically included in frame scripts:

**Terminate**

Exits the frame

**ChildEntry**

Enters a subform, composite field, or any field on the form

**ChildExit**

Exits a subform, composite field, or any field on the form

**ChildSetValue**

Changes a subform, composite field, or any field on the form

**FrameResized**

Changes the size of the window
UserEvent

Specifies a user event

For more information about these events, see the Language Reference Guide online help.

Examples—Frame event blocks:

The following frame event block is an example of a standard quit sequence:

```plaintext
on click menu.file_menu.quit_menu,
on click quit_button =
begin
    return;
end;
```

The following frame event block is an example of a standard frame termination sequence:

```plaintext
on windowclose,
on terminate =
begin
    if CurFrame.Topform.HasDataChanged = TRUE then
        /* save data */
    endif;
end;
```

The following frame event block is an example of a typical field on a form:

```plaintext
on childsetvalue =
begin
    data_is_changed = TRUE;
end;
```

Field and Menu Item Scripts

A field or menu item script contains one or more event blocks for the particular field or menu item. The syntax is:

```plaintext
[initialize =
[declare
    [localvariablelist]
    [localprocedureforwardreferences]
[enddeclare]]
[begin
    statementlist
end [:]]
```
where \textit{eventblock} is:

\begin{verbatim}
on event {,on event } =
   [declare
      localvariablelist
   [enddeclare]]
begin
   statementlist
end[:]
\end{verbatim}

\textit{event}

Specifies any event described in the the \textit{Language Reference Guide} online help. In a field or menu item script, the event must be one that can be used for that particular field or menu item.

For a list of the event types that you can use with each system class, see online help.

\textit{statementlist}

Specifies a list of statements that OpenROAD executes when the specified event occurs.

Like event blocks in frame scripts, the statement list can include any number of statements of any length. Separate the statements with semicolons.

Because the script for an individual field or menu item contains event blocks only for that particular field or menu item, you need not specify the name for the field or menu item in the event block (although it is legal to do so).

If you use an initialize statement in both a frame script and a field script, the frame's initialize statement will run before the field's initialize statement. If you use an initialize statement in both a field script for a composite field and in a field script for a child of that composite field, the child field's initialize statement will run after the composite field's initialize statement.
Example—Field and Menu Item Event Blocks

The following event block for the salary field checks that the value in the field is below a specified maximum when the user leaves the field:

```
on setvalue =
begin
  if salary > 100000 then
    CurFrame.InfoPopup
      (messagetext = 'Salary too high.',
       messagetype = MT_ERROR);
    resume;
  endif;
end;
```

The following sample event block executes when the user selects Close from the File menu:

```
on click =
begin
  return;
end;
```

User Class Scripts

A user class script contains methods that you can invoke to use with variables defined for a user class. The syntax for this type of script is:

```
[initialize =
 declare
  localprocedureforwardreferences
 enddeclare]
 [method methodname [([parameterlist])] =
 [declare
  localvariablelist
 enddeclare
 begin
  statementlist
 end [:]]
 {localprocedure[::]}
```

**methodname**

Identifies a method defined for the user class that the script uses. You cannot define a method twice in a user class script. For more information about methods, see the Language Reference Guide online help.

For more information about using methods in a user class script, see Working with Classes (see page 113).
**Statements in Scripts**

Scripts can contain any mixture of 4GL statements and SQL statements acceptable to OpenROAD. These statements enable OpenROAD to perform a wide variety of tasks:

- **Declaration and assignment**
  These statements let you define variables and assign values to them.

- **Data management**
  These statements, such as select, insert, delete, and update, let you query and update the database.

- **Loop control**
  These conditional statements, including for, if, while, continue, and endloop, let you control the processing flow within a script, event block, or procedure.

- **Program flow control statements**
  These statements let you control the basic flow of the application by calling frames, procedures, operating system, or even another Ingres tool.

For detailed descriptions of all the OpenROAD statements, see the *Language Reference Guide* online help. For more information about SQL statements, see SQL Syntax Summary (see page 483).

**Procedures**

A *procedure* is a named piece of code that performs a single task. Unlike the event-driven code that comprises frame and field script code, procedures process in sequence. Procedures also differ from frames in that they are not associated with a form. They are used primarily to eliminate duplication among your scripts and to make code more readable. Procedures also provide the means of accessing a 3GL language from an OpenROAD script.

You can define procedures in two ways:

**Globally**

Procedures that are defined globally to an application can be called from any OpenROAD script in that application or, if included in another application, from a second application as well.

**Locally**

Procedures that are defined locally within a frame, field, or procedure script can be called only from the script in which they are defined or from descendent fields (fields within the scope of the calling script).
OpenROAD lets you create three types of procedures:

**4GL procedures**

Global 4GL procedures are created in the OpenROAD Workbench. Local 4GL procedures are declared and defined in the frame, field, or procedure script in which they are called. For the procedure statement syntax to write a 4GL procedure, see the *Language Reference Guide* online help.

**Database procedures**

Database procedures are created and maintained independently of OpenROAD, but must be registered with specific applications. They are available outside of OpenROAD applications.

**3GL procedures**

3GL procedures are created and maintained independently of OpenROAD, but must be registered with specific applications.

**How You Can Create Procedures in OpenROAD**

Creating a *global procedure* in OpenROAD is a two-step process:

1. Declare the procedure in OpenROAD with the Procedure Editor. At this stage, you give the procedure a name and declare the return value type. For instructions about this process, see the *User Guide*.
2. Write the body of the procedure directly in the Script Editor using the procedure statement.

Creating a *local 4GL procedure* in OpenROAD is also a two-step process:

1. Declare the procedure directly in the frame, field, or procedure script.
2. Write the body of the procedure using the procedure statement.

You register and create database procedures or 3GL procedures using the appropriate Procedure Editor in OpenROAD Workbench. You write 3GL procedures as you would any 3GL program. After you create and compile the source code for a 3GL procedure, you create a dynamic-link library to link the procedure into the application.

To call a procedure, you can use the callproc statement. There are slightly different versions of the callproc statement for each type of procedure. Calling 4GL and database procedures is described in subsections of this chapter.

For more information about creating, calling, and linking 3GL procedures, see Using 3GL in Your Application (see page 295).
Global Procedures

A global 4GL procedure is a routine written in 4GL that you can call by name from a script or another procedure. The procedure is a component of an application and is stored in the application's database.

To create a global 4GL procedure, use the OpenROAD Workbench to name the procedure and declare its return value. You then write the actual procedure statement in 4GL. The following section describes this statement. For more information about creating 4GL procedures, see the User Guide.

Procedure Statement

The procedure statement defines a 4GL procedure. This statement declares the parameters for the procedure and provides the actual processing statements. The syntax for a global 4GL procedure is:

```
procedure procname [([parameterlist])] =
[declare
 [localvariablelist]
 [localprocedureforwardreferences]
enddeclare]
begin
_statementlist
end[:]
{localprocedure[:]}
```

**procname**

Specifies the procedure name. You use this name when you call the procedure in the application.

**parameterlist**

Specifies parameters that are passed to the procedure. You can also use a parameter as a local variable within the procedure. A 4GL procedure can have any number of parameters.

A parameter can be any simple data type acceptable to OpenROAD or any named user class or system class. If a procedure has no parameters, the parentheses are optional.

For more information about the declare block, see Initialize Statement (see page 68).

**statementlist**

Includes any OpenROAD statement
For a complete description of the procedure statement, see the *Language Reference Guide* online help.

For more information about local procedures, see Local Procedures (see page 78).

**Returning from a Procedure**

Use the return statement in your procedure to return control to the script or procedure that called it. In returning control, you can also pass a value back to the calling script or procedure. The single value that you pass back must match the return value data type that you specified when you created the procedure. The value is passed back to the caller and assigned as the value of the return variable in the calling script or procedure.

You can use a procedure that returns a value as an expression by itself or as part of a larger expression, as described in Procedures in Expressions (see page 47). The data type of the returned value must be compatible with the requirements of the expression. In the following example, square_root is a procedure that returns a value.

```
returnfield = square_root(number = 36) * 5;
```

A procedure that returns a value can be used as a parameter value in another procedure. The following is an example of nested OpenROAD procedures:

```
returnfield = p (arg1 = square_root(number = 17));
```

The following example calls the procedure with the name specified in the procfield field of the current frame and returns a value to the result variable:

```
result = :procfield (x = 5, y = price);
```

If you use the value found in a variable for the procedure name, you must use the colon. At runtime, the user enters the correct procedure name into procfield on the current form, and OpenROAD then calls that procedure.

For more information about callproc, see the *Language Reference Guide* online help.
Local Procedures

You can use a local procedure in 4GL to code a callable procedure for a single frame, field, procedure, or user class script.

Although you define local procedures in the frame, field, procedure, or user class script in which they are called, you must declare a forward reference before you can define a local procedure. Put the forward reference in the declare block for the script in which the local procedure is defined.

You define local 4GL procedures directly in the frame, field, procedure, or user class script using the procedure statement as described in the following section.

Use the callproc statement to call a local procedure defined in the current frame or procedure. For more information see the Language Reference Guide online help.

Declaring Forward References for Local Procedures

Use one of the following formats when declaring the local procedure forward reference:

localprocname = procedure returning typeofreturnvalue

localprocname = procedure returning none

localprocname = procedure

The procedure returning none and the procedure formats are equivalent. They specify that the procedure does not return a value.

How You Can Define Local Procedures

Use the following syntax to define a local procedure:

procedure procname [([parameterlist])] =
[declare]
localvariablelist
[enddeclare]
begin
statementlist
end[]
For more information about the parameterlist, see Procedure Statement (see page 76).

For more information about the declare block, see Initialize Statement (see page 68).

The runtime system searches local scopes of the currently executing frame or procedure and executes the local procedure if it is in the callproc string variable.

Example—Local Procedure

This section provides an example of a procedure definition and a sample call to the procedure. Here is the procedure definition:

```sql
procedure addtax (tax=float8, price=float8) =
{
    return (price * tax);
}
```

The name of the sample procedure is addtax. It has two parameters, tax and price. The data types of each are float8. The caller passes information to the procedure using these parameters. When you call the procedure, it uses the parameter values it receives from the caller in its calculations and returns the result to the caller.

The following is a sample statement calling the addtax procedure:

```sql
tax = callproc addtax (tax = taxpercent, price = costfield);
```

The order in which the parameters are specified in the callproc statement need not match the order in which they appear in the procedure's heading. However, they must be identical in name to the parameters in the procedure definition. Here is a second call to the same procedure:

```sql
cost = currprice + addtax (tax = .1, price = currprice);
```

Database Procedures

A database procedure is a data-oriented procedure stored in the database and executed within the database server that you can call by name in a script or 4GL procedure. Database procedures are often used to increase performance and help ensure data integrity and consistency.
To use a database procedure in an OpenROAD application, create the procedure using the Database Procedure Editor and then register it with the OpenROAD Workbench. For instructions about how to register a database procedure, see the User Guide.

For more information about calling database procedures in OpenROAD, see Calling Procedures (see page 83).

3GL Procedures

A 3GL procedure is a procedure written in a third-generation language, such as C, that you can call by name from a script or 4GL procedure. 3GL procedures are used to perform operations that are outside the scope of OpenROAD but are available from a 3GL.

You maintain and compile 3GL procedures outside of OpenROAD. For more information about how to register 3GL procedures in an application, see the User Guide.

The following subsections describe how to declare parameters for 3GL procedures. They also provide language-specific information about coding the procedures. For more information about calling 3GL procedures, see Using 3GL in Your Application (see page 295).

3GL Parameters

OpenROAD lets you pass simple variables to a 3GL procedure. You can also pass objects and arrays as described in the Language Reference Guide online help and in Using 3GL in Your Application (see page 295). When you pass values to a 3GL procedure, the data type of the values must match the data types of the parameters receiving them in the procedure.

The following table lists the 4GL simple data types and describes their corresponding 3GL data type declarations:

<table>
<thead>
<tr>
<th>4GL Data Type</th>
<th>Host Variable Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>Fixed-length, null-terminated character string. The size is determined by field size or declared variable length. Data entered into the field is padded with blank characters up to its full declared length before being passed to the 3GL procedure.</td>
</tr>
<tr>
<td>Varchar</td>
<td>Variable-length, null-terminated character string. Unlike char type, varchar variables are not extended with blank characters before being passed to the 3GL procedure.</td>
</tr>
<tr>
<td>4GL Data Type</td>
<td>Host Variable Declaration</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>date</td>
<td>Fixed-length, 25-byte, null-terminated character string</td>
</tr>
<tr>
<td>money</td>
<td>Double</td>
</tr>
<tr>
<td>float, f</td>
<td>Double</td>
</tr>
<tr>
<td>integer</td>
<td>Long, 4-byte integer</td>
</tr>
<tr>
<td>smallint</td>
<td>Integer, 4-byte integer</td>
</tr>
</tbody>
</table>

You can use the byref option with any variable that you pass to a 3GL procedure, regardless of the variable's data type.

You cannot pass a char or varchar variable that contains an embedded zero byte (hexadecimal '00') to a 3GL procedure. No runtime error occurs, but a truncated version of the 4GL variable may be passed.

When a 4GL date variable passes to a 3GL procedure by name, or when the return type of a 3GL procedure is date, 4GL receives it back from the 3GL procedure as a 25-byte string and must then convert it to the internal date format. The date must be valid before the conversion can succeed.

Passing a 4GL variable containing a null value to a 3GL procedure causes a runtime error. Use the ifnull function and is null comparison operator to pass nullable types between 3GL and 4GL, for example:

```sql
ifnull (v1, v2)
```

This reference returns the value of v1 if v1 is not null; otherwise, if v1 is null, it returns the value of v2. The variables v1 and v2 must be the same data type.

If an impossible value exists for the argument, use the impossible value to indicate a null:

```sql
callproc empcalc (ifnull (age, -1));
```

If no impossible value exists for the argument, pass a separate indicator variable to indicate a null argument:

```sql
null_indicator = 0;
if (age is null) then
    null_indicator = -1;
endif;
callproc empcalc (age, null_indicator);
```
Guidelines for Writing C Procedures

Use the following syntax when writing a C procedure:

\[
\text{procname}([\text{parameters}])
\{
\quad \text{processing statements}
\}
\]

**Note:** You cannot name your C procedure "main," and the procedure must not be static. You can call any C procedure from 4GL except the "main" function.

Follow these guidelines for passing parameters to C procedures:

- Pass an integer as four bytes by value (or by reference if byref is specified).
- Pass a smallint as four bytes by value (or by reference if byref is specified).
- Pass a float as a double-format float by value (or by reference, if byref is specified).

To ensure full portability, pass all floating point parameters to C procedures using the byref qualifier. For example, the following code fragment declares some variables and calls a C procedure, passing the procedure a floating-point parameter:

/* variable declarations */
test_float = float;
test_integer = integer;
test_return = integer
...
test_return = callproc myCproc (test_integer,
byref(test_float));

The corresponding C procedure is declared as follows:

```c
Long myCproc (ivalue, fvalue)
long ivalue;
double *fvalue;
{
\quad \text{processing statements}
}
```

- Pass a string as a pointer to a null-terminated string as follows:
  - Pass fixed-length string types (c, char) with trailing blanks up to their full length.
  - Pass variable-length string types (text, varchar) without trailing blanks.
In this call to a procedure “q”:

callproc q (1 + 2, 2.3, 'This is a string');

the following declarations are required:

q(x, y, z)
long x;
double y: char *z;
{
...

To receive x and y that are passed by reference, make the following changes to their formal argument declarations:

long *x;
double *y;

No changes are necessary to receive z that is passed by reference.

A C procedure must return an int, a long, a double, or a char * value, as shown in the following examples.

- To return an integer:

```c
int reti()
{
    return 10;
}
```

- To return a floating-point value:

```c
double retf()
{
    return 10.5;
}
```

- To return a character string:

```c
char * rets()
{
    return "Returned from rets";
}
```

Any C procedure that returns a char * value to 4GL must return a pointer to a valid address (a string constant or static or global character buffer). The procedure cannot return a pointer to a local character buffer.

**Calling Procedures**

In OpenROAD, the frame or procedure that includes the callproc statement is referred to as the *calling frame or procedure*. The procedure you specify in the callproc statement is referred to as the *called procedure*. 
You usually call a procedure with one or more parameters. A parameter is the name of a local variable in the called procedure that is visible to code outside the procedure. You can use parameters to pass values between the procedure and the code that called it. Each time you call the procedure, you can pass different parameters, enabling one procedure to operate on different data. Passing parameters differs for the three types of procedures.

If the procedure returns a value, you can use it as an expression or as part of an expression in an OpenROAD statement. For more information, see How You Can Call Procedures in Expressions (see page 86).

**How You Can Call 4GL Procedures**

The callproc statement lets you call a global 4GL procedure from any OpenROAD script or another 4GL procedure. You also use the callproc statement to call a local procedure defined in the current frame or procedure.

In the simplest version of the statement, you can call a procedure with no return value or parameters. The basic syntax is:

```
callproc procedurename
```

*procedurename* specifies the name that you gave to the 4GL procedure when you created it. You can enter the procedure name directly or you can use a variable to specify the procedure name dynamically. Using a variable lets you specify the procedure name at runtime.

The following example shows the use of this statement:

```
callproc error_handler;
```

When the procedure returns a value, you must specify a variable in the calling frame or procedure to receive the return value. The return variable must be the same data type as the return value. The basic syntax is:

```
[return_variable =] callproc procedurename
```

**How You Can Pass Parameters to 4GL Procedures**

To pass a value to a 4GL procedure in the callproc statement, you specify the name of the parameter in the 4GL procedure that is to receive the value. The basic syntax is:

```
[return_variable =] callproc procedurename [(parameter = expression {(, parameter = expression})]
```
**parameter**

Assigns the expression in the callproc statement to the parameter in the called procedure that you declared with the procedure statement. For more information about declaring procedure parameters, see the *Language Reference Guide* online help.

Because you specify the parameters by name rather than by position, you can pass them in any order, and you need not pass all the parameters.

**expression**

Specifies a constant or any legal OpenROAD expression, as long as the resulting data type is compatible with the data type of the local variable in the procedure. Variables in the expression can be any type, including reference variables and array variables.

The following example from the video_list frame script uses the callproc statement to call a procedure that removes a row from the video_detail frame:

```script
callproc remove_new_detail(new_details_frames =
    new_details_frames, details_frame =
    video.details_frame);
```

Any variable in the procedure that you do not specify in the parameter list is set to its default value.

Parameters to 4GL procedures can be passed by value and by reference. For a discussion of passing parameters by value and by reference, see How You Can Pass Parameters Between an Active Child and Inactive Parent (see page 101).

The value of the referenced variable is not updated until the called procedure returns. At this point, the variable in the calling frame is updated the same way it would be if you had used an assignment statement. If a field on the form is associated with the variable, the field does not display the new value until the current event block completes. To cause the value to update before the end of the event block, you can use the Flush method immediately after the callproc statement. For more information about the Flush method, see the *Language Reference Guide* online help.

### How You Can Call Database Procedures

A *database procedure* is a data-oriented procedure stored in the database and executed within the database server that you can call by name in a script or 4GL procedure. Database procedures are often used to increase performance and help ensure data integrity and consistency.

In OpenROAD, you call database procedures the same way you call 4GL procedures.
The syntax for calling a database procedure is:

```
[integer_variable = ] callproc procedurename
  [(parameter = expression
    {. parameter = expression})]
```

**procedurename**

Specifies the name you specified when you created the database procedure and that you later provided when you registered the database procedure in OpenROAD Workbench.

In the following example, the callproc statement calls the database procedure named mydbprocedure:

```
callproc mydbprocedure;
```

**How You Can Pass Parameters to and From Database Procedures**

You use parameter names to pass parameters to database procedures just as you do with 4GL procedures. You can pass the parameters in any order. For database procedures, the parameters must be simple variables only.

In the following example, intparam, floatparam, and charparam are integer, float, and varchar variables, respectively:

```
returnvalue = callproc mydbprocedure(intparam = 256,
  floatparam = 3.625,
  charparam = 'seen any good movies lately');
```

When the procedure returns a value, you must specify an integer variable in the calling frame or procedure to receive the return value, for example:

```
returnvalue = integer;
```

**How You Can Call Procedures in Expressions**

If a procedure returns a value, you can use the procedure as an expression or as part of an expression in an OpenROAD statement. The data type of the return value must be compatible with the requirements of the expression.

To call a procedure as part of an expression, do not use the callproc keyword. Simply give the procedure name followed by the parameter list. In the following example from the check_out frame script, the expression in the if statement calls the error_handler procedure:

```
if error_handler(frm = CurFrame, commit_evt =
  'Commit') != ER_OK then
  resume;
endif;
```
When you call a procedure in an expression, you must include the parentheses even if you are not passing parameters to the procedure. In the following example, the parentheses are empty because no parameters are being passed to the `dbms_error_message` procedure:

```informal
message 'Cannot insert graphic into the
database. ' +dbms_error_message();
```

In OpenROAD, 4GL procedures can return a value of any type. You can operate on the return value with any operation appropriate to the return value type. In the previous example, the `dbms_error_message` procedure returns a text value. Therefore you can use the return value within the text expression.

If the return value is a reference variable, you can manipulate it as any other reference variable. For example, you can use dot notation to access a specific attribute or to invoke a method of the object.

### How Procedure Handles Work (ProcHandle Objects)

A **procedure handle** (ProcHandle object) represents a global or local procedure and the scope in which to execute it. ProcHandle objects have several uses, for example:

- They let called frames call procedures declared locally in their calling frame.
- They let field scripts declare local procedures that can be called from anywhere in the frame.

ProcHandle objects are useful to pass procedures to contexts in which the procedure is not otherwise visible. They are used similarly to function pointers in the C language.

For example, you could write a field script that defines its own local cleanup procedures. In its initialize block it could insert its handles into an array of cleanup procedure handles defined at the frame level. You can create local cleanup procedures for other field scripts as well. Later, at cleanup time, the frame can loop through the cleanup array and use the handle to call each procedure.

### How You Can Create a ProcHandle Object

To obtain a ProcHandle object, invoke the `GetProcHandle` method on a Scope object. The syntax is:

```informal
prochandle = scope.GetProcHandle(name = string);
```
**string**

Specifies a literal string or a string variable that specifies the name of a procedure.

**scope**

Specifies where the procedure:

- Is visible
- Is to be executed

Generally, the scope of a ProcHandle object is CurScriptScope (the context of the currently executing field script) or CurFrame.Scope. To create a ProcHandle object for a global procedure named identically to a local procedure, use a global scope to represent the current application (CurSession.Scope).

If the currently executing frame or procedure is part of an included application, CurSession.Scope represents the global scope for that included application as well as the global scopes for all applications directly included by that application.

The names of included applications can be specified explicitly. For example, the following code fragment gets the procedure handle for a procedure named procname defined for an application named appname. The procedure handle is put into a variable named handle:

```
handle = CurSession.Scope.GetProcHandle (Name = 'appname!procname');
```

**Note:** appname must reference an application directly included by the currently active application.

---

**How You Can Execute a ProcHandle Object**

To execute the procedure specified by the ProcHandle object, invoke the Call method defined for the ProcHandle class. The syntax is:

```
[retval =] prochandle.Call(parameters);
```

The parameters are specified just as in a callproc statement. For an explanation about using the callproc statement, see How You Can Pass Parameters to 4GL Procedures (see page 84).

The Call method uses the scope specified by the ProcHandle object to execute it.
Restrictions to Using the Call Method

There are some restrictions to using the Call method. This method can be invoked only in the following places:

- In a statement by itself
- As the right side of an assignment

The Call method can never be invoked inside an expression.

If a procedure handle represents a local procedure that was defined in a frame or global procedure component, there are two additional restrictions on when the Call method can be invoked on the procedure handle:

- The Call method cannot be invoked after the frame or global procedure in which it was defined (and which contains the scope on which the GetProcHandle was issued) has terminated.
- The Call method cannot be invoked from a “thread” other than the one in which the GetProcHandle was issued.

For example, a frame can create a procedure handle on one of its local procedures and pass it as a callback routine to a called frame, but not to an opened frame (because the openframe statement creates a new thread).

These restrictions do not apply to a procedure handle that represents a global procedure or a local procedure defined in a user class script.

Global Procedures Available in the Core Library

The following global procedures are available in the core library:

_**StringParseKeyword**

Returns the values of the specified keyword

_**StringSub**

Substitutes values for %parameters in a string

_**StringTokenSub**

Substitutes a value for a single token in a string. Only the first occurrence of the token is replaced.

The following sections describe these procedures.
**The _StringParseKeyword Procedure**

The _StringParseKeyword procedure returns the value of the specified keyword in a specially formatted string. This string's format is:

```
"keyword1=value1[;keyword2=value2]"
```

The equal sign separates a keyword from its value. The semicolon must be used to separate the end of a value from the next keyword. There is no space allowed before or after the equal sign. Everything between the semicolon (or the beginning of the string) and the equal sign is treated as part of the keyword. The last value does not have to end with the semicolon. If a value has trailing white space, it will be trimmed. Leading and embedded white space is retained.

The search for the keyword is not case sensitive. Trailing white space on keyword is trimmed before the search begins. If the keyword is found, its value is returned. If either the string or the keyword argument is empty, an empty string is returned.

Example syntax for the _StringParseKeyword procedure is:

```sql
varchar(2000) = _StringParseKeyword(string = varchar(2000),
    keyword = varchar(32));
```

The following is an example of how to use the _StringParseKeyword procedure to find the value of a keyword:

```sql
str1 = 'Name=John Doe; 
Address=101 California, USA'; 
value = _StringParseKeyword(string = str1, 
    keyword = 'address');
```

This example returns the string "101 California, USA".

**The _StringSub Procedure**

This procedure substitutes values for parameters that are embedded in a specially formatted string. It also substitutes "\t" with HC_TAB, and "\n" with HC_NEWLINE.

The string must be a varchar string embedded with parameters in the format of "%1", "%2", and so forth, up to "%9", as well as "\t" and "\n". A parameter can occur multiple times in the string.
The syntax for the _StringSub procedure is:

```plaintext
varchar(2000) = _StringSub(string=varchar(2000),
arg1=varchar(100), arg2=varchar(100), arg3=varchar(100),
arg4=varchar(100), arg5=varchar(100), arg6=varchar(100),
arg7=varchar(100), arg8=varchar(100),
arg9=varchar(100));
```

The following code is an example of this syntax:

```plaintext
str1 = _StringSub(string = 'Employee Name:%1, Address:%2(c/o %1)',
arg1 = 'John Doe',
arg2 = '101 California, USA');
```

This example returns "Employee Name: John Doe, Address: 101 California, USA (c/o John Doe)".

**The _StringTokenSub Procedure**

This procedure substitutes a value for the specified token in a string. The syntax for this procedure is:

```plaintext
varchar(2000) = _StringTokenSub(string=varchar(2000),
token=varchar(256),
replacewith=varchar(256),
[remainingtokens=byref(integer)]);
```

The optional argument remainingtokens, if specified, is set to the number of occurrences of tokens that remain in the string after the substitution.

The following example shows how to use the _StringTokenSub procedure:

```plaintext
str1 = _StringTokenSub(string='Employee Name:John Doe, Address: 101 California, USA (c/o John Doe)',
token = 'John Doe', replacewith='Jane Doe',
remainingtoken=byref(icount));
```

This example returns "Employee Name: Jane Doe, Address: 101 California, USA (c/o John Doe)". The icount variable contains 1, indicating that there is one more occurrence of John Doe in the substituted string.
Compiler Limitations

A compilation unit (that is, an entire 4GL source file, including all its included files) should not contain more than:

- **2728 blocks**
  Each of these items constitutes a block: the compilation unit itself, each field script, event block or initialize block with a declare statement, local procedure, or method.

- **32767 integer literals**
  This limit includes integer literals generated internally by the 4GL compiler.

- **32767 floating-point literals**

- **32767 string literals inside expressions**

- **32767 string literals outside expressions**
  This limit includes string literals generated internally by the 4GL compiler.

A single execution unit, a local procedure, method, global procedure (excluding its local procedures) or frame (excluding its local procedures), should not contain more than 32767 variables. This limit includes temporary variables generated internally by 4GL.

The OpenROAD compiler imposes a limit on the complexity of a single expression. The limit may be violated in complex conditions of if, for, and while statements, especially if the condition contains several groups of comparisons that have both AND and OR clauses. For example, a statement like the following can cause a compilation error if there is insufficient memory:

```plaintext
if a = 't' and b = 'h' and c = 'e' and d = 'n'
or a = 'w' and b = 'a' and c = 'n' and d = 'q'
or a = 's' and b = 'g' and c = 'r' and d = 'k'
or a = 't' and b = 'h' and c = 'e' and d = 'n'
then
callproc found_it;
endif;
```
You can avoid the compilation error by splitting the complex if statement into multiple if and elseif statements. For example:

```c
f = false;
if a = 't' and b = 'h' and c = 'e' and d = 'n'
    then f = true;
elseif a = 'w' and b = 'a' and c = 'n' and d = 'q'
    then f = true;
elseif a = 's' and b = 'g' and c = 'r' and d = 'k'
    then f = true;
elseif a = 't' and b = 'h' and c = 'e' and d = 'n'
    then f = true;
endif;
if f = true then
    callproc found_it;
endif
```
Chapter 4: Programming Frames

This section contains the following topics:

- How You Can Invoke Frames (see page 95)
- How You Can Use Ghost Frames (see page 111)
- How You Can Run Non-interactive Applications (see page 111)

This chapter describes how to program frames. A frame is a window that consists of a form, with or without a menu, used to display and input data.

How You Can Invoke Frames

The callframe, openframe, and gotoframe statements invoke frames and provide options for executing additional frames. They differ in how they handle control and communication between frames and how they pass and return values.

However, there are also many similarities among these commands. One example is the process by which the calling frame can place the called frame into a particular state, presenting the user with continuity between frames.

This section presents the differences and similarities in the usage of these three statements.

FrameExec Object

Each time you open a frame with the openframe, callframe, or gotoframe statement, OpenROAD creates a FrameExec object for the frame. FrameExec is the OpenROAD system class that contains information about the running instance of a frame, such as the parent frame, the list of fields that are currently selected, and the starting menu for the frame. The FrameExec object also provides methods for manipulating the frame, such as the SendUserEvent method for communicating with other frames and the methods for opening predefined pop-up frames. For a complete list of the FrameExec attributes and methods, see the Language Reference Guide online help.

There is one FrameExec object for each running instance of the frame. If you open the same frame more than once, OpenROAD creates a separate FrameExec object for each instance of the frame.
How You Can Invoke Frames

How the CurFrame System Variable Works

To access the FrameExec object for the current frame from OpenROAD, use the CurFrame (for current frame) system variable. CurFrame can be used to change attributes such as:

- The size of the frame
- The background color of the frame
- The field that receives the input focus after the current event block has completed

For more examples of using CurFrame and information about how to reference the FrameExec object of another frame, see Inter-Frame Communication Techniques (see page 333).

How Frames Access Child and Parent Frames

CurFrame is also useful in letting a child frame access its parent (regardless of whether the child was opened or called by the parent frame). A child frame can always access its parent by using the value of the FrameExec's ParentFrame attribute, as in the following example:

```
parent_frame = FrameExec(CurFrame.ParentFrame);
```

It later uses the value of parent_frame to send a UserEvent to the parent.

If you want to change an opened frame's parent or enable two frames to communicate with each other, you can specify a FrameExec object as the ParentFrame in the with clause of the openframe statement. For example, a parent frame can pass a null FrameExec to a child frame, establishing the child as a detached frame. If the parent frame closes, the child remains open. For more information about using the with clause, see How You Can Change a Frame's Definition (see page 106).
The parent can also assign the opened child to a different parent by specifying a third frame's FrameExec as the child's ParentFrame. You might change a frame's parent to allow the child frame to remain open when the frame that called it closes. The following illustration demonstrates two frames, Frame_B and Frame_C, that are opened from Frame_A. Frame_D is opened by Frame_C. If Frame_C closes, Frame_D closes automatically. To allow Frame_D to remain open after Frame_C closes, you could parent Frame_D to Frame_B or Frame_A.

For Example, to change Frame_D's parent from its calling frame (Frame_C) to that frame's parent (Frame_A), you would type:

```plaintext
openframe Frame_D with parentframe =
    FrameExec(Curframe.Parentframe);
```

**Note:** Because the data type of the ParentFrame attribute is ProcExec, the previous statement converts it to FrameExec. The process used to convert the data type is called *casting*. For more information about casting, see How You Can Work with Attributes (see page 140).

You can also enable two child frames to communicate with each other by passing the appropriate FrameExec to a child or storing a FrameExec object in a global variable.

A child frame can always communicate with its parent by accessing the value of the ParentFrame attribute. For a parent frame to access its child, it must get the child's FrameExec object when it opens the child. The openframe statement provides access to the FrameExec object for the child frame. For an explanation of obtaining the FrameExec object of the called frame from the return value of the openframe statement, see How You Can Communicate Between Open Frames (see page 104).

Access to a frame's FrameExec object lets you use the SendUserEvent method to send events to the called frame. For more information about communicating between frames, see Inter-Frame Communication Techniques (see page 333).
How You Can Invoke Frames

The three OpenROAD statements that create a new FrameExec object by invoking a child frame differ greatly in how control is passed to the child frame. However, all three use similar syntax to run a second frame. A simplified syntax for all three statements is:

```
openframe | callframe | gotoframe framename [(parameterlist)]
```

You can enter the `framename` directly or you can use a variable to specify the frame name dynamically. Using a variable lets you specify the frame name at runtime. For an example of specifying a frame name dynamically, see How You Can Transfer Control to a New Frame: Callframe Statement (see page 100).

How You Can Transfer Control Between Frames

When you run a frame with the openframe, callframe, or gotoframe statement, you can pass values to it with a parameter list. The syntax for the parameter list is:

```
parameter_name=expression{, parameter_name=expression}
```

- **parameter_name**
  Specifies the name of a variable in the called frame. This variable can be either a field variable or a variable that you declare in the initialize block's parameter list.

- **expression**
  Can contain constants, variables, or expressions from the calling frame. OpenROAD transfers the value of this expression to the corresponding variable in the called frame.

The following code example demonstrates opening a frame using the openframe statement. It uses the vlist table field to store information about individual videos. Every time a detail frame is opened for a particular video, the FrameExec for the detail frame is stored in the vlist[].details_frame. Before opening a new frame, the code checks to see whether the details frame has already been opened to avoid opening a second instance of it.

How You Can Pass Parameters Between Frames
The example also illustrates how the calling frame can set an attribute (WindowTitle) of the called frame using the with clause of the openframe statement.

```plaintext
on click view_button =
{
  if (vlist[].details_frame is null) then
    vlist[].details_frame = openframe video_detail(video_info = vlist[])
    with WindowTitle = vlist[].title;
  else
    vlist[].details_frame.BringToFront();
  endif;
}
```

Any parameter in the called frame that you do not include in the parameter list is set to its default value.

When you pass a simple variable, such as an integer or text string, any changes made to its value in the called frame are not reflected in the calling frame. However, when you pass a 4GL object, such as a table field, you are passing a reference to the object, not the object itself. Therefore, both the called frame and the calling frame have references to the same object, and any changes made to the object by the called frame are reflected in the calling frame as well. For more information about changing the value of a simple variable in both the calling and the called frames, see How You Can Pass Parameters Between an Active Child and Inactive Parent (see page 101).

**How You Can Close a Frame**

To close a frame, use the return statement. This statement closes the frame and returns control to the calling frame. The following example shows how the return statement is used to take users back to the control frame when they click the Close button or close the window:

```plaintext
on click close_button, on windowclose =
begin
  cleanup processing statements
  return;
end
```

In addition, when you return from a frame invoked by the callframe statement, you can use the return statement to pass a return value back to the calling frame.
For more information about returning from the openframe, callframe, and gotoframe statements, see the following sections:

- How You Can Open Concurrent Frames: Openframe Statement (see page 103)
- How You Can Transfer Control to a New Frame: Callframe Statement (see page 100)
- How You Can Pass Control While Closing the Parent: Gotoframe Statement (see page 104)

A return statement for the starting frame of the application closes all frames in the application and returns control to the level where the user entered the application. To allow users to return to the operating system, the starting frame for your application should include a Quit operation (either as a menu operation or a button) that executes the return statement. For more information about the return statement, see the *Language Reference Guide* online help.

If there is some reason for the user to be able to leave the application from a frame other than the starting frame, use the exit statement. This statement closes all open frames and returns control to the level where the user entered the application. For more information about the exit statement, see the *Language Reference Guide* online help.

**Differences Among the Frame-invoking Statements**

Because the three frame-invoking statements behave differently with reference to the frame they call, they have many unique features. The following subsections describe the different aspects of the callframe, openframe, and gotoframe statements.

**How You Can Transfer Control to a New Frame: Callframe Statement**

The callframe statement opens a new active frame that blocks the calling frame (makes it inactive). The called frame has control until you close it and returns control to the calling frame with the return statement. After you return control to the calling frame, the calling frame becomes active again.
How You Can Return a Value from an Active Child to the Inactive Parent

One feature of the callframe statement is its ability to pass a return value from the called frame to the calling frame using the return statement.

When you use the return statement in a frame that was opened with the callframe statement, you can pass a single value back to the calling frame. This enables the calling frame to get information back from the frame that it called. For example, you can use the return statement to send back a status code, which the calling frame can then use in conditional processing.

When you issue the original callframe statement, you must specify a variable name to receive the return value. The syntax is:

```
variable = callframe framename ([parameterlist])
```

**variable**

Specifies any variable within the scope of the calling component, including a field variable or a local variable declared in the initialize or declare blocks. The data type of the variable must be compatible with the return value data type you specified when you created the frame with the OpenROAD Workbench.

Then, in the return statement, you must specify the return value. The syntax is:

```
return expression
```

**expression**

Specifies any variables, constants, or expressions from the called frame. The value of the expression must be appropriate for the return data type of the called frame.

For example, in the following callframe statement, id is an integer variable:

```
id = callframe fakeframe;
```

The following return statement passes the integer value back to the frame that issued the callframe statement:

```
return 33;
```

How You Can Pass Parameters Between an Active Child and Inactive Parent

Another feature of the callframe statement is its ability to pass parameters by reference as well as by value. To pass a parameter by value, simply specify the parameter name and assign a value to it in the parameter list of the callframe statement.
For example, in the following statement, int_var, ref_var, and array_var are all parameters; and 2, check_out_row, and videos are their assigned values, respectively:

```
callframe myframe (int_var = 2, ref_var =
    check_out_row, array_var = videos);
```

When you pass a simple variable, such as int_var, by value, any changes that you make to its value in the called frame are not reflected in the calling frame. For example, if myframe changes int_var to a value of 10, this change is not visible to the frame that called myframe. When myframe closes, int_var still has a value of 2 in the calling frame.

In contrast, when you pass a parameter that references a 4GL object or an array variable by value, any changes that you make to the object identified by the variable are reflected in the calling frame. This is because both array variables and variables referencing 4GL objects (reference variables) are pointers. When you pass a pointer to a called frame, both the called and the calling frame have a pointer to the same object. Consequently, any changes to that object in the called frame are visible in the calling frame when the called frame closes.

To pass a parameter by reference, you use the byref keyword. The syntax is:

```
callframe framename (parameter = byref(variable)
    {, parameter = byref(variable)})
```

The following code is an example of this syntax:

```
callframe myframe (number = byref(int_var),
    check_rec = ref_var,
    movies = byref(array_var));
```

This example passes two variables by reference, number and movies, ; and one variable by value, check_rec. Because the variable passed by value references a 4GL object, changes made to the object in the child frame are visible in the parent frame.

Because the simple variable, number, was passed by reference, it also displays changes to the parent that are made in the child. When you pass a simple variable, such as int_var, by reference, any changes made to the parameter in the called frame are reflected in the variable in the calling frame after the called frame closes. Passing simple variables by reference, therefore, gives you a way to pass multiple return values to a calling frame.
Using the *byref* keyword to pass a reference or array variable lets you reassign the variable (a pointer) to a different object in the called frame and have that reassignment visible in the calling frame. The previous example shows how the *array_var* parameter is passed to *myframe* by reference. Consequently, if *myframe* reassigns the *movies* array variable to point to a different object, when *myframe* closes, *array_var* in the calling frame will also point to the new object.

In all cases, no changes reflected in the calling frame are visible until the called frame closes. At this point, the variable in the calling frame is updated the same way it would be if you had used an assignment statement. That is, if there is a field associated with the variable, it does not display the new value until the current event block completes. To cause the value to be updated before the end of the event block, use the *Flush* method immediately after the callframe statement. For more information about the *Flush* method, see the Language Reference Guide online help.

**How You Can Open Concurrent Frames: Openframe Statement**

The *openframe* statement opens a second frame while keeping the calling frame active. Use the *openframe* statement when you want to create concurrent frames.

The frame that you open with an *openframe* statement is considered a child of the calling frame. OpenROAD automatically terminates the child frame when the parent frame terminates. For example, if the user closes the *video_list* frame, which calls the *video_detail* frame, OpenROAD also closes *video_detail*.

When you use the *openframe* statement, it is important to keep the active frames synchronized. When the frames are displaying shared data, you must ensure that any changes that the user makes in one frame are reflected in the other open frames. For example, if the user makes a change in a called frame, the called frame can send a user event to the calling frame to inform it of the change.

When you first open the frame with the *openframe* statement, you can send parameters as starting values for the new frame. After issuing the *openframe* statement, however, you must explicitly coordinate all subsequent communication between the frames. For more information about the process involved, see Inter-Frame Communication Techniques (see page 333).

**How You Can Pass Parameters Between Active Frames**

If you pass an object as a parameter to the opened frame, you are actually passing the pointer to the object rather than the object itself. Therefore, because both the parent and child frames have pointers to the same object, any changes made to the object by the child frame are reflected in the parent frame.
If the object is associated with a field in the child frame, OpenROAD does not display the new value in the field until the current event block in the parent frame completes. To cause the value to be updated before the end of the event block, use the Flush method immediately after the openframe statement. For more information about the Flush method, see the Language Reference Guide online help.

**Note:** The openframe statement does not include a byref option. This is available for the callframe statement only. For more information, see How You Can Transfer Control to a New Frame: Callframe Statement (see page 100).

**How You Can Communicate Between Open Frames**

Because the calling frame remains open when a called frame is executed by an openframe statement, it is important that the calling frame be able to communicate with the child frame. Consequently, the openframe statement returns a reference variable of type FrameExec to the calling frame. For the calling frame to use this value, you must capture the FrameExec object of the child frame in a variable. The syntax is:

```
[FrameExec_var =] openframe framename ([parameterlist])
```

The calling frame can also use the returned FrameExec object with the SendUserEvent method to communicate with the child frame. For more information, see Inter-Frame Communication Techniques (see page 333).

**How You Can Pass Control While Closing the Parent: Gotoframe Statement**

The gotoframe statement closes the calling frame and opens a new frame that replaces the calling frame. OpenROAD transfers control to the called frame, and the user cannot return to the calling frame.

Because the gotoframe statement closes the calling frame and therefore does not return to it, it has no return value. You should use this statement when you do not want users to return to the calling frame; for example, when you want an interim frame that the user fills out between using one frame and another.
How You Can Invoke Frames

Because the frame invoked by the gotoframe statement replaces the calling frame, the parent frame of the calling frame now has a new child frame. The following is an illustration of this concept:

1. Frame A opens Frame B with the openframe statement. Frame A is the parent of Frame B.

   ![Frame A open Frame B](image1)

2. Frame B opens Frame C with the gotoframe statement. Frame B closes. Frame A is now the parent of Frame C.

   ![Frame A, Frame B, Frame C](image2)

OpenROAD automatically terminates the child frame when the parent frame terminates. Therefore, if the user closes the calling frame's parent frame, OpenROAD automatically terminates the new child frame.

How Returning to a Closed Parent Works

You cannot return to the calling frame (the frame that issued the gotoframe statement). However, you can use the return statement to return to the parent of the calling frame. You can specify an expression in the return statement if the data type of the parent frame’s return type matches that of the child.

How You Can Pass Parameters Between an Active Child and a Closed Parent

Because passing a 4GL object as a parameter means passing the reference to the object rather than the object itself, even though gotoframe closes the calling frame, the reference variable on the called frame still points to the original object. It may not be useful, however, to pass a field object from the form on the calling frame because that field is no longer displayed.

Note: The gotoframe statement does not provide a byref option. Passing values by reference is available for the callframe statement only. For more information, see How You Can Open Concurrent Frames: Openframe Statement (see page 103).
How You Can Change a Frame's Definition

When you open a frame, you can reset most attributes of the original frame definition. The openframe, callframe, and gotoframe statements accept an optional with clause that lets you set any of the attributes of the FrameExec object associated with the frame (with the exception that only the openframe statement can specify a value for the ParentFrame attribute). The syntax is:

```
openframe|callframe|gotoframe framename ([parameterlist]) [with optionlist]
```

**optionlist**

Specifies the following:

```
option=value {, option=value}
```

**option**

Specifies any FrameExec attribute that you are allowed to set, such as WindowPlacement, ParentFrame, and DBSession.

For a complete list of these FrameExec attributes, see the Language Reference Guide online help.

**value**

Specifies any appropriate value for the particular attribute. Any attributes that you do not set continue to have the same value that you specified in the original frame definition.

How You Can Block an Open Frame

The BlocksFrames attribute defines whether an executing event block or called procedure in a frame blocks activity in other active frames until it completes.

By default, when OpenROAD executes an event block or a called procedure in a frame, activity in other active frames is blocked. The user cannot perform any actions until the event block or procedure completes. This behavior is controlled by the BlocksFrames attribute.

When you start an application, OpenROAD sets the BlocksFrames attribute to TRUE for the application's starting component. Each subsequent component inherits the value of this attribute from its calling component. Consequently, unless you explicitly set BlocksFrames to FALSE, it remains TRUE for all components, and the application exhibits the default behavior.
If you set the BlocksFrames attribute to FALSE, the frame in which you changed it and any children of that frame (opened after you made the change) appear active to the user. Although the frame is actually still blocked while OpenROAD is executing an event block or procedure, the hourglass cursor does not appear, and the user can type data in an entry field. Note that this ability does not affect the processing of the event queues. The work the user does while OpenROAD is executing code is buffered and then placed in the native event queue when OpenROAD completes the event block or procedure.

You can change the BlocksFrames attribute in either of the following ways:

- When you open a frame, using the with clause of the callframe, openframe, or gotoframe statement, for example:

  ```
  status = callframe newframe with
     BlocksFrames = FALSE;
  ```

- Inside an event block, by direct assignment, for example:

  ```
  on click updt_button =
  begin
     CurFrame.BlocksFrames = TRUE;
     ...
     CurFrame.BlocksFrames = FALSE;
  end;
  ```

Set BlocksFrames to FALSE only for those events and procedures that should return quickly. If a frame has some event blocks that take little time to complete and some that take much longer, start the frame with BlocksFrames set to FALSE and change it to TRUE for the long event blocks. If you allow a long event to run with BlocksFrame set to FALSE, an inactive frame will not display the hourglass cursor and yet the user cannot choose it to execute an operation such as clicking a button.

The BlocksFrames attribute does not affect the behavior of the calling frame when OpenROAD is executing a:

- Callframe statement
  
  A frame executing a callframe statement is always blocked.

- Gotoframe statement
  
  The frame that executes a gotoframe statement is closed.

- Message statement
  
  A frame executing a message statement, directly or indirectly, is always blocked.
How You Can Invoke Frames

- **Prompt statement**
  A frame executing a prompt statement, directly or indirectly, is always blocked.

- **WaitFor method call**
  A frame that executes a WaitFor method is blocked until the method completes.

### Pop-up Frames

The openframe, callframe, and gotoframe statements enable you to open a frame as a pop-up frame. Pop-up frames are closely connected to the calling frame, always displaying on a layer above it and never iconifying independently of it (the pop-up is always iconified whenever the calling frame is iconified).

Pop-up frames are useful when you want to force the user to provide information or acknowledge a warning prior to continuing the action of the calling frame. Because the pop-up can never be hidden by the calling frame, it serves the purpose of gathering information better than an ordinary called frame (which could get lost behind the calling frame).

In addition to enabling you to define your own frames as pop-ups, OpenROAD provides four types of predefined pop-up frames. This section discusses user-specified and predefined pop-up frames.

### How You Can Create Your Own Pop-up Frames

You specify a frame as pop-up by setting IsPopup to TRUE in the optionlist when you invoke the frame's Property Inspector.

User-specified pop-up frames can be invoked to display on top of active as well as inactive parent frames. To access an active parent frame that has been covered by a child pop-up, the user must move it adjacent to the pop-up. The parent can never overlay the pop-up, which prevents the pop-up from being hidden by the parent. You can specify the usual frame attributes for a pop-up frame, such as resizability.

Pop-up frames must be called by a parent frame or procedure. Starting frames cannot be pop-ups. If a pop-up is specified as the starting frame for an application, it is treated as a standard frame at runtime, as is a pop-up frame run from the Go menu of the Frame Editor.

If a pop-up frame is invoked by an object that is not itself viewable, such as a procedure or ghost frame, it displays in relation to the closest viewable ancestor (for example, the frame that called the procedure).
How You Can Use Predefined Pop-up Frames

The four predefined pop-up frames display in the center of the calling frame and always make it inactive. Only after the pop-up frame is closed does the calling frame become active again.

The following table describes the four types of predefined pop-up frames:

<table>
<thead>
<tr>
<th>Pop-up Frame Type</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information</td>
<td>A message and an OK button</td>
</tr>
<tr>
<td>Confirm</td>
<td>A message, a Cancel button, and an OK button</td>
</tr>
<tr>
<td>Reply</td>
<td>A message, a prompt line, a Cancel button, and an OK button</td>
</tr>
<tr>
<td>File</td>
<td>A message, a file selection area, a Cancel button, and an OK button. (Your window toolkit determines the appearance of the file selection area.)</td>
</tr>
</tbody>
</table>

The following illustrates a Confirm pop-up frame:

![Confirm pop-up frame](image)

How You Can Call Predefined Pop-up Frames

Predefined pop-up frames can be called only for the current frame or procedure. To call them, you invoke one of four methods of the ProcExec system class:

- InfoPopup
- ConfirmPopup
- ReplyPopup
- FilePopup
How You Can Invoke Frames

The syntax is:

CurFrame.method(parameters)

or

CurProcedure.method(parameters)

**method**

Refers to one of the four previously mentioned methods. For more information about the parameters available, see the ProcExec system class in the *Language Reference Guide* online help.

You also can open Information and Reply pop-ups with the OpenROAD message and prompt statements. For more information about using these statements, see A Simpler Process for Creating Information and Reply Popups (see page 111).

To specify the value of a parameter, you must enter the parameter name followed by the value. The following example from the video_detail frame script uses the ConfirmPopup method to verify that the user wants to delete the video. The messagetext parameter specifies the text displayed in the pop-up frame.

```plaintext
on click delete_button =
begin
    status = CurFrame.ConfirmPopup(messagetext = 'You are going to delete this video. ');
    if status != PU_OK then
        resume;
    endif;
    ...
end:
```

ConfirmPopup, ReplyPopup, and FilePopup have return values. You must specify a variable of the appropriate type to receive the return value. In the previous example, the status variable is used to hold the return value.

Both the ReplyPopup and FilePopup methods have reply parameters. For the ReplyPopup method, this parameter contains a response that the user enters. For the FilePopup method, this parameter contains a file that the user selects. This reply parameter is a reference variable of type StringObject (for more information about the StringObject system class, see Working with Images and Text Strings (see page 277)).

When you invoke the ReplyPopup or FilePopup method, you can specify this parameter to display an initial reply in the pop-up frame. For example, with the FilePopup method you can use the reply parameter to specify a default file selection. If the user types in a different reply or selects a different file name, the value of the reference variable is updated.
A Simpler Process for Creating Information and Reply Pop-ups

The OpenROAD message and prompt statements provide an alternative process for opening Information and Reply pop-up frames, respectively. These frames function identically to the pop-up frames you open with the InfoPopup and ReplyPopup methods.

The OpenROAD message statement opens an Information pop-up frame that displays a specified message. The syntax of the message statement is:

```
message varchar_expression
```

The OpenROAD prompt statement opens a Reply pop-up frame that displays an entry field with the specified message as a prompt. You specify the variable name in which to store the user's response. The syntax of the prompt statement is:

```
variable = prompt varchar_expression
```

**variable**
- Specifies any varchar variable in the frame

**varchar_expression**
- Contains any variables, constants, or expressions

How You Can Use Ghost Frames

*Ghost frames* differ from other OpenROAD frames in that they do not contain forms (making them invisible to the user). They differ from 4GL procedures in that they can receive and execute user and database events. Ghost frames are useful for handling operations that run continuously without requiring user intervention.

How You Can Run Non-interactive Applications

Because ghost frames and 4GL procedures do not require the window manager to be running, you can write OpenROAD applications that run entirely without user interaction. To execute these applications without running the window manager, use the -nowindows flag when invoking the runimage or rundbapp commands.
When you run an OpenROAD application without the window manager, information that would display as a pop-up frame to executing windows is redirected as follows:

<table>
<thead>
<tr>
<th>Window System</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows NT,</td>
<td>Displayed in the trace window and written to the OpenROAD error log file</td>
</tr>
<tr>
<td>Windows 95</td>
<td></td>
</tr>
<tr>
<td>UNIX or Linux</td>
<td>Displayed in the trace window only</td>
</tr>
</tbody>
</table>

This redirection of information includes:

- Informational pop-up messages provided by OpenROAD (for example, those containing error messages)
- 4GL statements that do not require user response (for example, the message statement and the InfoPopup method)
- 4GL statements that require user response (for example, the prompt statement and the ReplyPopup method)

**Note:** Because they block the application until answered, you should not include statements requiring user response in frames or procedures that are to be run non-interactively. Moreover, because users may not notice informational statements directed to the log file or trace window, you also may want to avoid informational statements that require user response.

Other restrictions to running applications without the window manager include the following:

- Attempting to call, open, or go to frames that contain forms causes an error message.
- Attempting to set those CurSession attributes that contain information about the display or the window manager (for example, WindowSystem or ScreenHeight) has no effect.

For more information about running OpenROAD applications, see the *User Guide*. 
Chapter 5: Working with Classes

This section contains the following topics:

- **Conceptual Background** (see page 113)
- **How You Can Reference Objects** (see page 114)
- **Using an Object-oriented Approach to Programming** (see page 120)
- **Inheritance** (see page 121)
- **How You Can Create a User Class** (see page 127)
- **External Classes** (see page 135)
- **How You Can Create Fields from User Classes** (see page 139)
- **How You Can Manipulate Objects Generically: Casting** (see page 139)
- **Overview of the OpenROAD System Classes** (see page 142)

This chapter explains the concept of classes in OpenROAD and describes how system, external, and user classes make an object-oriented approach to programming possible. The detailed information in this chapter about creating user classes supplements the description of creating a user class provided in the *User Guide*.

### Conceptual Background

All application components are objects that are created using the OpenROAD system classes. For example, a frame is an object of the FrameExec system class, and a button field is an object of the ButtonField system class.

The class of an object defines the attributes you can read and set for that object. For example, the ButtonField class includes a BitmapLabel attribute that lets you specify an image as the button's label. For more information about how to read and set these attributes, see **How You Can Set and Get Attributes** (see page 117).

The class of an object also defines the methods that you can invoke to manipulate the object. For example, the EntryField class includes a MarkAllText method that you can use to highlight the text in an entry field. For more information about how to invoke a method on an object, see **How You Can Invoke Methods** (see page 119).

Following an object-oriented model, the OpenROAD system classes use an inheritance hierarchy in which each subclass inherits the attributes and methods defined for all its superclasses. When you create your own user classes, you can structure them into an inheritance hierarchy. For more information about class hierarchies, see **Inheritance** (see page 121).
In OpenROAD, you do not reference objects directly. Instead, every object is associated with a reference variable. A reference variable is a variable that points to an object of a given class, letting you access the value of an object. For example, to manipulate a value stored in a field, you use the field's variable name directly:

```plaintext
acct_balance = 1000.00
```

However, to manipulate the field as an object, accessing such characteristics as its background color or executing one of its methods, you must access the field's reference variable. A reference variable lets you manipulate all of the attributes and methods of that field's class. Whereas the variable storing the field's value is scalar, reference variables are pointers to compound data structures that reference all the object's attributes and methods.

A reference variable does not store an object's data values. Instead, when you reference the reference variable, OpenROAD uses the values stored in the corresponding object's attributes. If there is no corresponding object, the value of the reference variable is null.

An object's properties are stored in the attributes that define the object. Attributes can be:

- Simple data items, such as background color
- Other reference variables, such as an image used for a button label
- Dynamic array variables, such as the ChildFields array associated with a composite field

In 4GL code, you can work with individual attributes of the object (such as the background color) or with the object as a whole. Working with the whole object allows you, for example, to retrieve a row from a database table and assign all values from the row to a single object. You can then easily map the object to a set of fields in a subform or a row in a table field, or pass it as a unit to a procedure.
The following is an illustration of a single object of the VIDEO_ROW user class. It illustrates referencing an object as a whole. It shows a reference variable called video, which stores a single object from the user class VIDEO_ROW. Each object of the user class represents a row of data stored in the video database table.

When you declare a reference variable, you specify its class, and optionally specify its default value. The class defines the attributes in the associated object. OpenROAD automatically creates an object of the specified class when you create or declare the reference variable unless you also specify that it defaults to null.

The following example demonstrates creating the video reference variable for the VIDEO_ROW user class:

```plaintext
initialize ()=
    declare
        video = VIDEO_ROW default null;
    enddeclare
begin
...
```

To reference an individual attribute of the VIDEO_ROW object, such as a video's title, use the following dot notation:

```plaintext
video.title
```
How You Can Enhance Performance When Declaring Reference Variables

Because OpenROAD automatically creates an object of the specified class when you declare the reference variable without specifying that it defaults to null, you can enhance performance by specifying the null default in the following circumstances:

- Declaring reference variables as parameters
  
  If you declare a reference variable as a parameter and do not specify that it default to null, an instance of the object is allocated in the called frame or procedure even if no argument of that type is passed to the child.

  If an object is passed as an argument, the implicitly created object is overridden by the explicitly passed object. The time spent creating and removing the implicit object can be avoided by declaring each reference variable with default null.

- Declaring local variables
  
  Similarly, you can avoid an implicit allocation and variable removal in local variable declarations by specifying default null.

  Specify this clause when creating a variable that is used to reference an object, as opposed to creating an instance of the object. In the 4GL code you can do one of the following:

  - Point this reference variable to an existing object of the same type.
    
    Assume that subfrm.customer.address is an entry field that is referenced throughout a long script. To enhance performance when working with such a deeply nested attribute, you can point a second reference variable to it. You can then use the second variable as an abbreviation for the first, for example:
    
    ```
    declare
      ef1 = EntryField default null; /* instance not required */
    enddeclare
    ... ef1 = field(subfrm.customer.address);
    ```

  - Create an object of the same type. If you create a reference variable for an object that does not exist and you set its default to null, you must explicitly create the object before you can reference it, for example:
    
    ```
    Declare
      ef2 = EntryField default null; /* instance not required */
    enddeclare
    ... ef2 = EntryField.Create( );
    ```

**Note:** If you do not declare a variable with default null, you do not need to create its object explicitly. For examples of explicit object creation, see Creating Dynamic Frames (see page 363).
Field Function

Every field or menu item has two variables associated with it:

The data variable
Stores the data that is displayed in a field. OpenROAD declares this variable automatically when you create the field or menu item using the OpenROAD Workbench.

The reference variable
Points to the field object. The field object contains information about the field itself, such as its size or color, as well as containing the data displayed in the field. To access this reference variable, you must use the field function. The field function returns the reference variable that is associated with the field object. The basic format is:

```
field(display_variable)
```

When you want to reference an attribute of a field object, use the field function to specify the reference variable as follows:

```
field(display_variable).attribute
```

In the following example, the if statement uses the field function to reference the UpdateBias attribute of the customer field:

```
if field(customer).UpdateBias = FB_VISIBLE then
```

How You Can Set and Get Attributes

Setting an attribute in 4GL may change the appearance or behavior of the object. For example, setting the value of the FileHandle attribute for the BitmapObject system class actually causes OpenROAD to load a bitmap image into the object.

Sometimes setting an attribute for one object affects other objects. For example, if you set the InputFocusField attribute of the FrameExec object, OpenROAD moves the input focus from the current field to the field you specify; doing so triggers appropriate Exit and Entry events for the fields.
Use an assignment statement to assign a new value to the attribute. The basic format is:

\[ \text{reference\_variable\_attribute} = \text{expression} \]

**reference\_variable**

Specifies the name of the reference variable that points to the object

**attribute**

Specifies any one of the attributes defined for the object's system class

**expression**

Specifies any 4GL expression that is the appropriate type for the particular attribute

The following example changes the mode of the current frame by setting the value of the CurMode attribute to FM_READ:

```plaintext```
CurFrame.CurMode = FM_READ;
```

To set the value of an attribute for a field or menu item, you must use the field function in the assignment statement. The syntax is:

\[ \text{field(display\_variable).attribute} = \text{expression} \]

The following example uses the field function to set the value of the CurBias attribute for the Create button:

```plaintext```
field(create_button).CurBias = FB_INVISIBLE;
```

**Note:** When you set the value of an attribute for a child field inside a named composite field, you must specify the full name of the field. To change the bias of the cname field in the customer subform, for example, you would use the following statement:

```plaintext```
field(customer.cname).CurBias = FB_INVISIBLE;
```

There are also occasions when you want to get an attribute's value. For example, you may want to learn which field has the current input focus or which field triggered an event. The process for getting the value of an attribute is very similar to that for setting one. Use an assignment statement to obtain the value of the attribute. The basic format is:

\[ \text{variable} = \text{reference\_variable\_attribute} \]

**variable**

Can be a global variable or any variable. This variable must have an appropriate data type for the particular attribute.
The following example assigns the field that currently has input focus to the reference variable `charfield`. Note that `InputFocusField` is an object.

```plaintext
charfield = CurFrame.InputFocusField;
```

To get the value of an attribute for a field or menu item, you use the field function in the assignment statement. The syntax is:

```plaintext
variable = field(display_variable).attribute
```

The *Language Reference Guide* online help specifies which attributes you can set (or write) and which attributes are read-only for each system class.

### How You Can Invoke Methods

To invoke a method for an object, use the same dot notation that you use for referencing attributes. If the method has parameters, you must specify the parameters by name, just as you do with the parameters for a frame or procedure. If the method has no parameters or you do not want to specify parameters, you must include a pair of empty parentheses. The basic syntax is:

```plaintext
reference_variable.method({parameter_list})
```

- **reference_variable**
  - Specifies the name of the reference variable that points to the object

- **method**
  - Specifies any one of the methods defined for the object's system class

- **parameter_list**
  - Specifies an optional list of parameters using the following syntax:

```plaintext
parameter_name = expression {, parameter_name = expression}
```

  - **parameter_name**
    - Specifies any of the parameters defined for the method

  - **expression**
    - Specifies any 4GL expression that is the appropriate type for the particular parameter

You can specify the parameters in any order.

To invoke a method for a field or menu item object, use the field function to return the reference variable:

```plaintext
field(display_variable).method({parameter_list})
```
The following example illustrates this:

`field(acctno_form.acctno).MarkAllText();`

A method may return a value. You must specify a variable name to receive the return value. The syntax is:

`variable = reference_variable.method([parameter_list])`

**variable**

Specifies any variable in the current scope and must have an appropriate data type.

The following example specifies the `video` variable to hold the return value for the `Create` method (that is, to be the object created):

`video = VIDEO_ROW.Create();`

When the method returns a value, you can use it as an expression or as part of an expression in an OpenROAD statement. The following example uses the value set by the user's response in an expression:

`if CurFrame.ConfirmPopup(messageText = 'some message') = PU_OK then ...;`

For a complete list of the methods that are defined for each system class and for the data type of each method's return value, see the *Language Reference Guide* online help.

---

**Using an Object-oriented Approach to Programming**

User classes let you take an object-oriented approach to your data. Rather than writing table-oriented applications with database access specified in frame and procedure scripts, you can create objects that reflect your work environment.

For example, suppose that you have a database containing information about a library. There may be a table, Books, containing information about books in the library. You can easily create a user class, books, with attributes that correspond to items in the Books table. The Class Editor lets you create attributes from a database table. After you create the class books, you can then define useful methods. For example, a books class might have the methods CheckOut and CheckIn.

When you create a frame, you can create fields that correspond to attributes of a user class by using the Insert Fields from User Class menu option.
To build meaningful objects representing your organization's data, you must understand basic concepts concerning classes and specific information about user classes. The following sections lay the groundwork for you to build object-oriented applications.

**Inheritance**

OpenROAD classes are organized into an inheritance hierarchy. With *inheritance hierarchy*, a class that is a child of another class is considered its *subclass* and a class that is the parent of another class is considered its *superclass*. Each subclass inherits the attributes and methods for all its superclasses.

The higher a class's position in the hierarchy, the more generalized are the characteristics and functions that it defines. Classes lower in the hierarchy describe objects with specialized characteristics that function in a specific manner. For example, the characteristics and functions defined for the FieldObject system class apply to all fields and menu items, whereas the ButtonField system class defines a very specific object.

The inheritance hierarchy is useful in avoiding redundant definition of functions and characteristics. Attributes and methods are defined at the most general level possible and inherited by all subclasses.

For example, the ButtonField system class is a subclass of ScalarField. The complete list of attributes and methods available for ButtonField includes the attributes defined for ScalarField and its superclasses (ActiveField, FormField, FieldObject, and Object) as well as those defined specifically for ButtonField.

Because both ButtonField objects and SliderField objects have background color, both of these system classes inherit their background color (BgColor) attribute from the same superclass, FieldObject. The BgColor attribute functions identically for both field objects, for example:

```cpp
field(my_button_field).BgColor = CC_YELLOW;
field(my_slider).BgColor = CC_YELLOW;
```

Some classes are created for the sole purpose of high-level attribute and method definition. General classes that are used for classification only are known as *abstract classes* in object-oriented programming. Examples of abstract system classes in OpenROAD include ActiveField and FieldObject.
Class Relationships

You can create user classes to structure your organization's data into objects. These objects, with all of their characteristics and behaviors built in directly, can serve as the basis for your applications.

Before creating classes to represent your data, you must understand the two basic kinds of relationships between classes. Only after you have clearly identified a class and its relationships to other classes can you best define attributes and methods at the most general level. These kinds of relationships are:

- Inheritance Hierarchy (or Generalization/Specialization)
- Aggregation (or Containing)

Inheritance Hierarchy (Generalization/Specialization)

The inheritance hierarchy inter-class relationship assumes that each class defines its own attributes and methods and inherits those defined for its superclass.

In developing an application for managing a company, you might define a superclass called Employee with two subclasses, PermanentEmployee and TempEmployee. Because every employee must be either permanent or temporary, Employee is an abstract (general) class that defines attributes common to all employees, such as LastName, FirstName, EmpNum, and HireDate. PermanentEmployee and TempEmployee are specialized classes that represent actual employees.

The PermanentEmployee and TempEmployee subclasses inherit the attributes of Employee. In addition to the Employee attributes, PermanentEmployee defines attributes specific to its class, such as InsurancePlan and NextReviewDate.

Because managers are a specific kind of permanent employee, you might define a Manager user class that is a subclass of PermanentEmployee. In addition to the attributes inherited from both of its superclasses, this class also has a BonusRate attribute.

Methods common to all staff are also defined at the top level. The Employee class, for example, defines the Hire and Terminate methods. Methods specific to a subclass are defined at the subclass level. For example, the Manager class might define a CalculateBonus method.
Examples of OpenROAD system classes that follow this relationship model are ActiveField and ButtonField. ActiveField is an abstract class that defines the characteristics and behavior of all fields with which users can interact. The ButtonField class is a subclass of ActiveField that provides an area on the form or which the user can initiate an action. Its characteristics and behavior are more specifically defined than its parents.

**Aggregation (Containing)**

The *aggregation* inter-class relationship describes an object that is composed of separate, smaller objects that function together. The Employee and Job classes provide an example:

- The Job object contains attributes (such as Title, Class, and Pay) that describe a specific job held by a specific employee during a specified time.
- An Employee object consists not only of scalar attributes (such as LastName and HireDate) but also of a complete job history for each position the employee has filled at the company. (For a full description of this class, see How Polymorphism Works (see page 125).

The JobHistory attribute of the Employee class, which is an array of Job objects, exemplifies an aggregation relationship.

Although the data pertaining to employees is stored in several database tables and structured in different user classes, the application user who manipulates the Employee object does not think in terms of the separate objects composing the whole, but works with the aggregate as a single object. Aggregate objects allow far more complicated structures than simple objects.

An example of an OpenROAD system class that follows this relationship model is TableFields. TableFields contain such other classes as an optional scrollbar (ScrollBarField), an optional control menu (ControlButton), and an array of column fields (StackField).

Most objects participate in both types of relationships simultaneously. For example, Employee and Job are both members of an inheritance hierarchy and share an aggregation relationship.

**User Class Hierarchies**

You can organize your user classes into an inheritance hierarchy similar to the hierarchy of the OpenROAD system classes. The Employee superclass with PermanentEmployee and TempEmployee subclasses is one such example.
You can specify a user class as the superclass of another user class; otherwise the superclass is automatically defined as UserObject. If you specify another user class as the superclass, it must be defined in the current application or any of its included applications.

Although you cannot create subclasses of a system class by specifying the system class as the superclass of a user class, you can use field templates and frame style sheets to simulate a child of a system class. For example, if you want different types of ButtonField objects to have certain behaviors or visual characteristics, you can add new button fields to a frame template, setting the attributes to your specifications and attaching a script that ensures certain behaviors.

For example, if you always want a Clear button to look and act in a certain way, you can create a field template that contains this button. Fields that are built from the field template would all use this version of the Clear button.
How Polymorphism Works

Although classes inherit the attributes and methods of their superclasses, an inherited method may be redefined for a class to allow it to manipulate its own defined attributes. For example, assume you have created the user class hierarchy shown in the following illustration:

![Class Hierarchy Diagram]

Every user class in the illustration has the attributes defined at its own level, as shown in the box for each user class, as well as all attributes defined by its superclasses. TempEmployee's attributes, therefore, are identical to Employee's, whereas Manager and PermanentEmployee define additional attributes.

Methods defined for each class are illustrated by a brace to the right of the class. For example, the Terminate method is defined only for the Employee class and inherited by all subclasses. The Hire method is defined for the Employee class and redefined at the level of each subclass.
The Terminate method is defined at the top of the hierarchy because terminating employees involves the same operation for all classes.

The Hire method inserts appropriate entries in database tables. In addition, the Hire method for PermanentEmployee inserts such information as NextReviewDate and InsurancePlan. The Hire method of the Manager class inserts the same information as the PermanentEmployee Hire method, but also enters information into the BonusRate column.

To cause a method to supersede the functions defined at the superclass level, you use the same name for the method (in this case, Hire), but you redefine the procedure in the user class script for the appropriate subclasses. The process of a subclass's method superseding the behavior of an identically-named method defined for a superclass is called polymorphism.

**How You Can Build an Inheritance Tree**

The illustrated example in How Polymorphism Works (see page 125) suggests that two of the subclasses (PermanentEmployee and Manager) define additional attributes at their own level, requiring the Hire method to behave differently for the superclass and its subclasses. The scenario is different if two subclasses define identical additional attributes.

Suppose that you create a class to represent contract employees. If the Hire method is the same for both temporary employees and contract employees, you could use the same source code for the Hire method in both the TempEmployee and ContractEmployee classes. However, this is a case where defining an intermediate abstract class may be more efficient. The following diagram shows how you can create an HourlyEmployee class that contains the Hire method for both TempEmployee and ContractEmployee:
You thus define the Hire, GetCurrent, and GetHistory methods, and the attributes they use, only once: for abstract class HourlyEmployee. The TempEmployee and ContractEmployee classes inherit these methods from HourlyEmployee.

**How Invoking User Class Methods in 4GL Works**

User classes are functionally equivalent to system classes, that is, any operation you can perform with a system class in the 4GL code of an executable component can be done with a user class. However, although the 4GL syntax for invoking a method on a user class object is identical to that of a system class, you often handle user classes differently from system classes.

For example, the following code invokes the Hire method on a temporary employee:

```plaintext
tempemp.Hire(LastName = 'Goodman', FirstName = 'Robert', HireDate = 'today', EmpNum = 1292, TerminationDate = '2/28/94', CurrentClass = 14, CurrentPay = 5.00);
```

In all likelihood, however, most values for the Hire method are input at runtime by the user typing into form fields. The form fields are children of a composite field that is mapped to the TempEmployee user class. Because these values are automatically available to any method invoked on that composite field, you may not need to pass method parameters programmatically as shown in the code example.

**How You Can Create a User Class**

You create a user class, define its attributes, and optionally declare and define its methods and superclass in the Class Editor of the OpenROAD Workbench. For instructions to create a user class, see the User Guide.

When you define attributes, you specify:

- The name by which they are referenced
- A data type
- A default value (optional)
- Nullability
- A remark describing the attribute
- Whether the attribute is private to the class
When you define methods, you specify:

- The name by which they are invoked
- A data type for the return value (can be nullable)
- A remark describing the attribute
- Whether the method is private to the class

The data type of an attribute or method can be either a simple data type or a class, including another user class. If a user class is defined as the data type, the user class must exist at the time the attribute or method is created.

You define all of a user class's methods in a single 4GL script that is part of the user class source object. Any operations that can be performed syntactically in the 4GL code of a frame or procedure can be done in the 4GL code of a user class method, including invoking another of the class's methods.

Like many other OpenROAD application components, user classes can be created dynamically.

### How You Can Encapsulate Attributes and Methods

In object-oriented programming, you can limit access to an object's attributes and methods. The process of allowing only the object itself to have access to its attributes and methods is called *encapsulation*. In OpenROAD, you accomplish this restriction by flagging an attribute or method as private. You can only reference a private attribute or method in the 4GL code of the methods of the defining user class or its subclasses.

Attribute encapsulation is useful when you have an attribute that is affected by several different methods, each of which needs that attribute to be in a known state. To prevent programmers from changing the attribute in the 4GL code, you can make the attribute private so that programmers can only access it from the object's methods.

Method encapsulation is useful if you want a method to be invoked only by another method of the user class or its subclasses. For example, if you have a function that you would usually define as a local procedure, but you want child objects to have access to the function as well, you would code the function as a private method rather than a local procedure, thus overcoming the scoping limitations of local procedures. If you defined the function as a local procedure, you would have to redefine the procedure in every relevant method.

When you create a user class as an abstract class, you can keep others from using this class by making all of its attributes and methods private. Caution is required, however, when flagging everything in an abstract class as private. Private classes deny access to developers who are using objects that inherit features of the abstract class.
**How You Can Write Methods**

You access the script for the methods of a user class from the Class Editor. For procedures to edit a method script, see the *User Guide*.

The methods created in the script must correspond to methods defined for the user class. For the syntax for writing a method script, see the *Language Reference Guide* online help.

The 4GL code for a method can make use of two predefined system variables:

**CurObject**

References the current instance of the user class. It provides access to every attribute of the current instance of a user class to every method for that class. When defining a method's functions, you access attributes of the current user class by using the CurObject system variable.

For example, the first function of the Employee class's Hire method is to check for the employee's hire date. If no date has been assigned, the method supplies the Ingres special constant (today) for the current date. The Hire method refers to the HireDate attribute of the object on which the method is invoked by using the CurObject system variable.

The following code performs this function:

```4gl
if CurObject.HireDate = '' then
  CurObject.HireDate = 'today';
endif;
```

When you invoke the Hire method on an object called tempemp, CurObject refers to tempemp.

**CurMethod**

This variable refers to the running instance of the current method. It is analogous to CurProcedure for a procedure. For example, the TempEmployee script that redefines the Hire method includes the following statement to display a predefined popup:

```4gl
if CurObject.TerminationDate = '' then
  CurMethod.InfoPopup(messageText =
    'Temporary employees must have a termination' + ' date specified.',
  messageType = MT_ERROR);
```

For more information about defining methods in user class scripts, see the *Language Reference Guide* online help.
How You Can Use Polymorphism

Because permanent and temporary employees do not share identical attributes, and because different business rules apply, their hiring processes differ. For this reason, the Hire method, defined in a general way for the Employee superclass, is modified at the subclass level to handle additional attributes defined for each subclass.

Moreover, because OpenROAD allows polymorphism, you can write a method for a subclass that builds on the functions of a method of the same name defined for the superclass. For example, the Employee user class defines a Hire method that contains functions common to all of the subclasses of Employee:

- Provides a hire date
- Provides a unique employee number
- Inserts information about the new employee into the database
- Checks for errors

Each of the subclasses of Employee modifies the Hire method to its specific needs:

- TempEmployee ensures that a termination date is provided at the time of hire and supplies a termination reason
- PermanentEmployee inserts the following information into the database:
  - NextReviewDate
  - VacationDays
  - InsurancePlan
  - Bonus_temp (null if the new employee is not a manager)
- Manager provides a BonusRate

The MethodExec system class defines the SendSuperclass method to facilitate the process of supersedence. The SendSuperclass method invokes the superclass's version of the current method, avoiding the need to repeat the code in the subclass's method.

The SendSuperclass can be invoked only on the CurMethod system variable. The same arguments used to invoke the method on the subclass are passed implicitly to the parent. Arguments passed explicitly to SendSuperclass are ignored and a warning message is generated at runtime.

The SendSuperclass method returns current parameters without requiring a byref keyword. If invoking SendSuperclass causes the parent class to change the value of an attribute, that change is returned automatically to the subclass when control returns to the subclass.
The code position of the statement invoking the SendSuperclass method is significant. If it is important for the parent class's method to manipulate attributes or data before running the subclass-specific operations, the SendSuperclass method must be invoked before the subclass's code. If the operations performed by the parent's method are dependent on the operations specific to the subclass, the SendSuperclass method must be invoked after the operations specified for the subclass's code.

The following sections provide methods that exemplify both of these situations.

**Examples of Method: Hire Method**

The following sections describe the Hire method as it is defined for the Employee class and the TempEmployee, PermanentEmployee, and Manager subclasses.

Each of these methods handles error checking, which performs a rollback if the database access fails. A failure at the bottom of the hierarchy causes rollbacks in every method previously invoked by the SendSuperclass method.

**Note:** No commit statements are issued in these methods to allow for future additions to the user class hierarchy. Adding a new subclass whose method includes additional SQL statements as part of the same transaction would force a revision of all superseded methods containing a commit statement.

These methods assume that the programmer handles all commits. For consistency, the rollback statements could alternatively be handled by individual programmers and removed from the method scripts.

**Employee Version**

In this top-level version of the Hire method, the method script defines the functions described in the following steps. The attributes referenced in this script are set interactively by the user in a frame whose fields are mapped directly to the user class. The method script does the following:

1. Declares a variable to store error status:
   ```
   method Hire() =
   declare
   errstat = integer not null;
   enddeclare
   ```

2. Checks the employee's hire date. If no date has been assigned, supplies the current date:
   ```
   if CurObject.HireDate = '' then
   CurObject.HireDate = 'today';
   endif;
   ```
3. Generates an employee number by increasing the sequence value in v_highest_number:
   
   ```
   update v_highest_number set h_number = h_number + 1 where tbl_name = 'emp'
   ```

4. Selects the highest employee number into the EmpNum attribute for the current employee:
   
   ```
   select h_number as :CurObject.EmpNum from v_highest_number
   where tbl_name = 'emp';
   ```

5. Inserts a new row into both the emp and jobs tables and checks for errors:
   
   ```
   insert into emp (emp_num, emp_type, last_name, first_name, hire_date, term_date, term_reason)
   values (:CurObject.EmpNum, :CurObject.Type, :CurObject.LastName, :CurObject.FirstName,
   :CurObject.HireDate, :CurObject.TerminationDate, :CurObject.TerminationReason);
   ```

   ```
   if CurMethod.DBSession.ErrorNumber = 0 then
   insert into jobs (emp_num, job_class, pay, start_date)
   ```

   ```
   endif;
   ```

   ```
   if CurMethod.DBSession.ErrorNumber != 0 then
   rollback;
   message 'Database error; unable to insert employee record.';
   ```

   ```
   endif;
   ```

   ```
   return CurMethod.DBSession.ErrorNumber;
   ```

   ```
   end
   ```

**TempEmployee Version**

Because all temporary employees have a known termination date and a standard termination reason at the time of hire, the purpose of redefining the method for this user class is to guarantee that these values are inserted into the emp table. After gathering the first of these values and providing the second, this method invokes its parent’s method by using the SendSuperclass method.

The TempEmployee invokes this method after setting values for the two critical attributes (TerminationDate and TerminationReason) so that the parent’s version of the Hire method runs with real values for these attributes.

The following 4GL code is the entire Hire method script for TempEmployee:

```
method Hire() =
declare
   errstat = integer not null;
enddeclare
```
begin
  if CurObject.TerminationDate = '' then
    CurMethod.InfoPopup(messageText = 'Temporary employees must have a termination ' +
     date specified.', messageType = MT_ERROR);
    errstat = -1;
  else
    CurObject.TerminationReason = 'Contract period end';
    errstat = CurMethod.SendSuperClass();
  endif;
  return errstat;
end

Because the TempEmployee subclass requires termination information for a new employee to be added to the database, it sets this information prior to calling its superclass's version of the method.

**PermanentEmployee Version**

Unlike the TempEmployee class, which invokes the SendSuperclass method only after providing values for attributes critical only to that subclass, PermanentEmployee invokes the SendSuperclass method first and then does additional work.

The PermanentEmployee subclass requires a successful insertion of new employee information into the emp and jobs tables (performed by the parent's version of the method) before it adds a new row to the permemp table. Therefore, it forces successful completion of the first insertions before performing its own insertion.

The PermanentEmployee version of the Hire method handles the following values for the defined attributes of the PermanentEmployee subclass:

- NextReviewDate
- VacationDays
- InsurancePlan
- BonusTemp

The Hire method defined for this subclass inserts values from all of these attributes into the permemp table. When this method is invoked on a permanent employee, rather than on its manager subclass, the value of the bonus_rate column of the permemp table defaults to null (only managers are entitled to a bonus).

If the BonusTemp attribute had not been defined for the PermanentEmployee class, hiring a manager would require two separate database operations, an insertion and an update (both into the permemp table). The insertion would be the same one performed by the PermanentEmployee's Hire method, and the update would be required to store the manager's bonus rate.
To enhance performance by executing a single database operation when hiring managers, the PermanentEmployee class has a private BonusTemp attribute, which is null by default. Because this attribute is private, no value can be supplied by invoking this method on a permanent employee object.

The following code is the entire Hire method for PermanentEmployee:

```plaintext
method Hire() =
declare
    errstat = integer not null;
enddeclare
begin
    errstat = CurMethod.SendSuperclass();
    if errstat = 0 then
        CurObject.NextReviewDate = CurObject.HireDate + '3 months';
        insert into permemp (emp_num, next_review_date, insurance_plan,
            bonus_rate) values (:CurObject.EmpNum,
            :CurObject.NextReviewDate, :CurObject.InsurancePlan,
            :CurObject.BonusTemp);
        errstat = CurMethod.DBSession.ErrorNumber;
        if errstat != 0 then
            rollback;
            CurMethod.InfoPopup(messagetext = 'Database insert failed with error ' +
                varchar(CurMethod.DBSession.ErrorNumber),
                message_type = MT_ERROR);
        endif;
    endif;
    return errstat;
end
```

Manager Version

The only difference between a manager and its PermanentEmployee superclass is that managers are given a bonus rate when hired. Therefore, the script for the Hire method that is redefined for the Manager subclass is the simplest of all the classes described.

The following code is the entire Hire method script for Manager:

```plaintext
method Hire() =
declare
    errstat = integer not null;
enddeclare
begin
    CurObject.BonusTemp = CurObject.BonusRate;
    errstat = CurMethod.SendSuperclass();
    return errstat;
end
```
This script first sets a value for the BonusTemp attribute and then invokes its superclass's version of the method. The Manager class's superclass, PermanentEmployee, also invokes the Hire method defined by its superclass, Employee. The SendSuperclass method in this script, therefore, has the effect of superseding the Hire method all the way up the hierarchy.

The SendSuperclass method is effective even if the immediate superclass does not define the method being superseded. For example, assume that the PermanentEmployee class did not define its own version of the Hire method, but its subclass, Manager, needs to supersede the version defined at the top of the hierarchy for the Employee class.

In this case, invoking the SendSuperclass method in the Hire method for Manager automatically invokes the current method as it was defined by Employee. The Manager class's script does not need to specify that it is superseding the method defined for Employee.

For the complete method scripts of the user classes discussed, see the user class components in the online Videos application.

**External Classes**

An *external class* defines the class properties, methods, and events for one or more external objects. An external class provides a wrapper for a single external object so that OpenROAD can use it seamlessly in your application. This wrapper provides display and storage space for the external object in your OpenROAD application. For instance, it allows a spreadsheet, word processor, or other application to be used within an OpenROAD application. OpenROAD can use a Microsoft Excel spreadsheet as an external object, read data from it, and use the data in OpenROAD code.

**ActiveX Support**

External classes let you use ActiveX controls and ActiveX automation in your OpenROAD applications.

ActiveX objects have properties, methods, and events just like other class objects. Therefore, they are treated as external classes in OpenROAD. Each unique ActiveX object used in OpenROAD must have an external class to represent it. The external class is viewed by OpenROAD as an actual system class. As such, external classes can be displayed and manipulated like any other class in the system.

For more information about ActiveX, see the documentation supplied with the ActiveX object.
How You Can Use External Classes

To use an external object in OpenROAD, you must first register the library that contains the external objects by using the External Class Library Editor.

Next, you have the option of doing either or both of the following:
- Adding external objects to an OpenROAD frame using the Frame Editor
- Declaring variables that reference external classes

For more information about these procedures and facilities, see the User Guide.

When you register an external class library, be aware that one library can have more than one external class. For example, registering a single external class library might provide three external classes that can be used in an OpenROAD application.

Dynamic Programming with External Objects

The ExtObjField class is a formfield that provides the wrapper for a single external object. By using the InsertObject method of the ExtObjField class, you can dynamically populate a form with external objects. For an example of how to do this, see the Language Reference Guide online help.

Non-Formfield External Objects

Not all external objects can be displayed in a form. One example would be an Excel automation object. Because a Microsoft Excel spreadsheet is an automation object, you cannot display it in an OpenROAD frame; however, you can use OpenROAD to print or manipulate it.

How You Can Use Collections with External Classes

ActiveX objects can be collections, which are like sets, and can be used as arrays if they have been declared as an OpenROAD collection. If you want to index into a collection using the array syntax, you first must declare it as an OpenROAD collection.
The following code is an OpenROAD collection example, which is discussed in the paragraphs that follow:

```
x = sheets;
x1 = sheets a collection of cells;
y = cell;
x.item(1)= y;
x1[1] = y;
```

In the preceding example, x and x1 are declared as reference variables of the sheets class type. Sheets is an external class that is actually an ActiveX collection. This collection contains objects of the cells class type. The x1 variable is also qualified as an OpenROAD collection. The y variable is declared as a reference variable of the cell class type. The syntax for indexing into the ActiveX collection depends on how the variables are declared. OpenROAD array syntax can be used if the variable is qualified as an OpenROAD collection. Otherwise, the collection is indexed using the standard item method.

The following code shows the indexing of a collection using array syntax:

```
y = cell;
x = sheets a collection of cells;
x[1]=y;
```

If the reference variables are declared as follows:

```
y = cell;
x = sheets
x1 = sheets a collection of cells;
```

Then the following statements are wrong and will fail:

```
x[1] = cells;
x1.item(1) = cells;
```

However, the following statements are correct:

```
x.item(1) = cell;
x1[1] = cell;
```

For more information about collections, see Collections (see page 254).
Example—ActiveX Controls

The following code is an example of how an ActiveX control can be coded in OpenROAD. It displays a web page in an OpenROAD frame. It assumes that the frame contains the following fields:

**browser**

Specifies an OpenROAD external object (an ActiveX control inserted into the frame)

**urlfield**

Specifies an OpenROAD entry field

**forward**

Specifies an OpenROAD button field

**back**

Specifies an OpenROAD button field

```pascal
initialize()=
declare
enddeclare
begin

curframe.flush();
urlfld = 'http://www.cai.com';
browser.navigate(urlfld);

end
on click forward =
begin
  browser.goforward();
end
on click back =
begin
  browser.goback();
end

on setvalue urlfld =
begin
  browser.navigate(urlfld);
end;

on extclassevent 'statustextchange'
  (text = varchar(256) not null)=
begin
  curframe.statustext = text;
end;

on extclassevent 'beforenavigate'
  (url = varchar(256) not null) =
begin
  urlfld = url;
end;
```
How You Can Create Fields from User Classes

You can create fields that correspond to the simple attributes of a user class. In the Frame Editor, click Insert, Fields from User Class. A pop-up window lets you select a user class from those defined in your application. You can also specify whether you want to create simple fields or a table field. Once you select a user class, the Frame Editor lets you place the new field or fields on the frame. If the user class contains both simple attributes and attributes that are objects, fields are created only for the simple attributes.

If the data type of a composite field (except tablefield) is set to be a user class, the displayed values of the fields in the composite field can be modified by changing the value of the underlying user class object—if the UpdField method is called on the field. However, setting the underlying user class object to NULL does not change the display.

How You Can Manipulate Objects Generically: Casting

In developing OpenROAD applications, you generally work with an object of a specific class. For example, when you set the background color for a button, you use the BgColor attribute of the ButtonField class. When your code is compiled, OpenROAD ensures that the attribute or method you specify is defined for the system class or one of its superclasses.

There are occasions, however, when you do not know at development time what object you will be manipulating at runtime. For example, if you want to change the characteristics of a field based on user interaction, such as changing the color of a field as the user enters and leaves it, you must be able to set attributes for an object whose type you do not know as you write the code. To facilitate your writing such generic code, OpenROAD provides special attributes and methods for certain system classes.

FrameExec's TriggerField attribute, for example, lets you detect which field triggered the executing event block. To color-highlight a field when the user moves into and out of it, you can use the TriggerField attribute to locate the field triggering the Entry and Exit events.

There are also occasions when you code generically with methods. For example, when you send a MessageObject from one frame to another by passing it as a parameter to FrameExec's SendUserEvent method, the receiving frame must know what kind of object to expect.
How You Can Manipulate Objects Generically: Casting

Because the TriggerField attribute can be used on any field or menu item, you must make explicit to OpenROAD the class of the object being manipulated in order for your code to manipulate specific attributes and methods. Casting involves making the system class of a generic object explicit.

To cast a generic object, enclose its reference variable in parentheses and precede the parentheses with the name of the subclass to which you want to cast it. If, for example, you want to change the background pattern of a field as the user enters it, you use the BgPattern attribute defined for FormField. The following example uses casting to specify that the generic TriggerField is an entry field and changes the background pattern of the field to shaded.

EntryField(CurFrame.TriggerField).BgPattern = FP_SHADE;

You can cast a generic object to a more specific object only if the specific object is a subclass of the generic one. This casting example is possible because the data type of FrameExec's TriggerField attribute is FieldObject, and FormField and EntryField are both subclasses of FieldObject.

Although the class to which you cast a generic object must be in the hierarchy below the generic object, you can specify any valid class as the type for the generic object. For example, to change the background pattern of a field, you can cast the generic object to FormField or any field below FormField on the hierarchy. The following example works similarly to the previous example, but allows all form fields to trigger the change in background pattern:

FormField(CurFrame.TriggerField).BgPattern = FP_SHADE;

When you cast an object, OpenROAD produces an error at runtime if the actual object is not of the same class as, or a subclass of, the class to which you have cast it.

Other Attributes Commonly Used Generically

The FrameExec system class defines several other attributes that facilitate generic code development:

**InputFocusField**

Contains the field on the form that currently has the input focus. The following example demonstrates changing the background pattern of the current field without knowing its name:

CurFrame.InputFocusField.BgPattern = FP_SHADE;

This example does not require casting because the data type of the InputFocusField attribute is ActiveField, and ActiveField is a subclass of the system class (FormField) for which the BgPattern attribute is defined.
OriginatorField

Contains the field defined in the on event statement that originated the specified event. For example, assume that you code a ChildExit event block for a composite field (On ChildExit CompField). Whenever the user exits any of the child fields composing CompField, the OriginatorField attribute contains the composite field name, CompField. In contrast, the TriggerField attribute contains the name of the specific child field that the user exited.

This attribute is particularly useful when you have multiple events defining the same event block. For example, if you have a radio field and menu list that you use to enable multiple interfaces to the same action and you want to keep them synchronized, you could use the following code:

```plaintext
on click TestRadioField,
on click TestMenuList =
begin
if CurFrame.OriginatorField =
  field(TestRadioField)then
  TestMenuList = TestRadioField;
else
  TestRadioField = TestMenuList;
endif;
/* common code goes here */
end;
```

TargetField

Contains the name of the field that will next have the input focus (assuming the focus is not moved)

How You Can Work with Methods

Assume you send a StringObject as a MessageObject from FrameA to FrameB, and you want to use the WriteToFile method on the MessageObject in FrameB. Because WriteToFile is defined for the StringObject class, you must cast the message object to avoid a compile error when you use the WriteToFile method.

FrameA contains the following statement:

```plaintext
FrameB_exec.SendUserEvent(EventName = 'Send_String',
  MessageObject = string_file);
```
The following code fragment from FrameB declares a string variable to receive the MessageObject, casts the MessageObject to a StringObject, and uses the WriteToFile method:

```plaintext
initialize (...) =
declare
  stringobj = StringObject default null;
enddeclare
begin
  ...
  on userevent 'Send_String' =
  begin
    stringobj = StringObject(CurFrame.MessageObject);
    stringobj.WriteToFile(filename = msg_string);
  ...
```

### Overview of the OpenROAD System Classes

This section provides a brief description of the most frequently used top-level system classes and provides a general idea of when to use them. The classes are grouped by the function they serve in an application, beginning with the most general classes.

Many of these system classes are most useful when you want to create a user class dynamically.

For more information on system classes, see the *Language Reference Guide*.

### Object

This abstract class is the superclass of all objects in the system:

**Object**

Provides methods that you can use on any object in the system. The SetAttribute and GetAttribute methods are useful in generic applications. The Duplicate method makes an exact duplicate of an object. The IsA method is useful to determine whether an object is of the same class as, or one of the subclasses of, a specified class name.
Class

This abstract class contains information about all the system classes:

Class

Contains information about all OpenROAD system classes. It provides a method (Create) that you use to create a new object of a specified class.

The Class system class has a single subclass, UserClassObject, which is a runtime object used to manage user class information.

User Class Source Definition

The following system classes contain definitions for a user class:

AttributeObject

Stores the information for defining a user class attribute

ClassSource

Stores the definitions for a user class, including a list of all attributes and methods for a given class, as well as the name of its superclass.

It also provides a CreateClass method for creating a UserClassObject object from a fully defined ClassSource object.

MethodObject

Stores the information for defining a user class method

These system classes are most useful when you want to create a user class dynamically.

User Class Runtime Definition

The following system classes represent runtime versions of a user class:

UserClassObject

Specifies an instance of a ClassSource that represents a specific user class. The UserClassObject object for a user class is instantiated at application start-up time from the definition stored in the ClassSource object for the user class.

As a subclass of the Class system class, it inherits the Create method that lets you create objects of this user class.
**UserObject**

Is the default superclass for user classes. The UserObject class inherits all of its methods from the Object class.

These system classes are most useful when you want to create a user class dynamically.

**Source Definitions for Applications and Their Components**

The following system classes contain the source definitions for an application and its components:

**AppSource**

Contains the application definition, which you can use to import an application component from or export an application component to an external file.

For more information about importing and exporting applications, see the *User Guide*.

**ClassSource**

Stores the definitions for user classes, including a list of all attributes and methods for a given class.

**CompSource**

Represents the components of an application; used for classification only

**Proc4GLSource**

Contains the source definition of a procedure. Proc4GLSource has one subclass, GhostSource, which contains the source definition of a ghost frame. GhostSource also has a single subclass, FrameSource, which contains the source definition of a frame.

These system classes are most useful when you are coding dynamically.

**Running Applications and Their Components**

The following system classes contain information about a running application and its components:

**SessionObject**

Contains information about the running version of the application, such as the current operating system

**DBSessionObject**

Contains information about the current DBMS session
**FrameExec**
Contains information about the running version of a frame

**GhostExec**
Contains information about the running version of a ghost frame.
For more information about ghost frames, see How You Can Use Ghost Frames (see page 111).

**MethodExec**
Contains information about the running version of a user class method.
For more information about user class methods, see How You Can Write Methods (see page 129).

**ProcExec**
Contains information about the running version of a procedure

**Field and Menu Information**

The FieldObject class and its subclasses describe the fields and menu items on the form. Every field and menu item on your frame is associated with a reference variable that points to a FieldObject (for more information about referencing field and menu objects, see Field Function (see page 117)).

An object of the FieldObject class contains information about the appearance of the field and has methods for manipulating the field. For example, every button is associated with a ButtonField object that contains information about the button, such as its size, color, and title.

The following are the system classes for the fields and menu items:

<table>
<thead>
<tr>
<th>Class</th>
<th>MenuStack (composite field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarField</td>
<td>MenuStack (composite field)</td>
</tr>
<tr>
<td>BoxTrim</td>
<td>MenuToggle</td>
</tr>
<tr>
<td>ButtonField</td>
<td>OptionField</td>
</tr>
<tr>
<td>ColumnField</td>
<td>OptionMenu</td>
</tr>
<tr>
<td>ControlButton</td>
<td>PaletteField</td>
</tr>
<tr>
<td>DisplayForm</td>
<td>PopupButton</td>
</tr>
<tr>
<td>EllipseShape</td>
<td>RadioField</td>
</tr>
<tr>
<td>EntryField</td>
<td>RectangleShape</td>
</tr>
<tr>
<td>ExtObjField</td>
<td>ScrollBarField</td>
</tr>
<tr>
<td>FlexibleForm</td>
<td>SegmentShape</td>
</tr>
</tbody>
</table>
### Event Information

The following system classes contain information about events received by the frame:

**DBEventObject**
- Contains information about a database event

**Event**
- Contains message parameter values that are passed to the frame by a SendUserEvent call to complete a WaitFor call

For more information about using events and database events, see Inter-Frame Communication Techniques (see page 333).
Chapter 6: Working with a Database

This section contains the following topics:

- **Conceptual Background** (see page 147)
- **How You Can Access a Database with Standard SQL Statements** (see page 148)
- **How You Can Access a Database with DataStream Objects** (see page 165)
- **How You Can Manage Transactions** (see page 182)
- **How Programming for Multiple Sessions Works** (see page 188)
- **Handling Database Errors** (see page 197)

This chapter describes how an OpenROAD application interacts with a relational database. This chapter explains the implementation of these features in the OpenROAD environment.

**Conceptual Background**

Most OpenROAD applications provide users with an interface to a relational DBMS that uses the four primary data manipulation statements—select, update, insert, and delete—to enable the following operations:

- Retrieve data for viewing or updating
- Add new data
- Delete existing data
OpenROAD also provides access to the database through the DataStream system class without directly executing an SQL statement. DataStream's subclass, SQLSelect, provides greater power and flexibility than the primary data manipulation statements. In particular, SQLSelect objects are useful for:

- Letting users step backwards through the retrieved data or to move directly to a specific row
- Creating dynamic queries when the target fields or variables are unknown until runtime
- Changing queries at runtime (for example, modifying the where and order by clauses)

SQLSelect objects interact dynamically with the database, verifying that the names and data types of the columns in their target list are the same as those coded when the application was developed.

QueryObject objects, a subclass of the SQLSelect class, also support partitioning a query and enable update, insert, and delete operations on dynamically retrieved data.

For a detailed discussion and an example of using an SQLSelect to retrieve data, see How You Can Access a Database with DataStream Objects (see page 165). For a detailed discussion and an example of using a query object to change data, see How You Can Use Query Objects (see page 177).

How You Can Access a Database with Standard SQL Statements

In addition to the four primary data manipulation statements (select, update, insert, and delete), OpenROAD lets most SQL statements to be used directly in your 4GL scripts. For a list of these statements, see the Language Reference Guide online help.

To use SQL statements that you cannot code directly in a 4GL script, such as the statement for creating a database procedure, OpenROAD provides the execute immediate statement. For more information about this statement, see How You Can Use the Execute Immediate Statement (see page 155).

The following subsections describe how to use the select, update, insert, and delete statements in your OpenROAD applications.
Select Statement

The select statement lets you retrieve data from the database into:

- Simple variables
- Reference variables
- Array variables

If the variable receiving data is associated with a field on a form that is currently displayed, the data is displayed in the field as it is assigned to the variable.

A select statement can retrieve:

**A single row**

A select statement that retrieves only one row is called a *singleton select*. The select statement deposits the retrieved data into variables that the program can process as needed.

**Multiple rows**

When a select statement returns more than one row, only the first row returned is visible to the program. To access the additional rows, use one of the following constructs:

**Select loop**

A *select loop* is a block of 4GL statements that are performed on each row returned by the select statement. The loop ends when there are no more rows or an endloop statement is encountered. Select loops are the simplest and often most efficient way to handle multiple rows.

**Cursor**

A *cursor* is a row marker that designates an individual row in the select statement's result table (the set of rows returned by a non-singleton select). As the program moves the cursor forward through the result table, you can update or delete the row to which the cursor points as well as use the values in the row.

For a detailed discussion of cursors in OpenROAD and examples of their use, see How You Can Use Cursors to Access the Database (see page 157).
Example—Singleton Select

The following example represents a singleton select. Given an account number, it selects information about a customer from a table that contains customer information:

```
declare
cname    = varchar(100) not null;
cphone   = varchar(10) not null;
enddeclare
{
    /*
    **/Given an account number, get customer information
    */
    select    cname as  :cname,
            cphone as  :cphone
    from      customer
    where     acctnum = :account_number;
...}
```

Example—Select Loop

Suppose that cust is a class with attributes that correspond to columns in the customer table. The following select loop extracts information from the customer table and stores it in an array of cust objects:

```
declare
cust_arr        = array of cust;
i               = integer not null;
enddeclare
{
...}
i = 1;
select         acctnum    as cust_arr[I].accnt,
            cname      as cust_arr[I].name,
            cphone     as cust_arr[I].phone
from        customer
order by   cname
{
    i = i + 1;
};
...}
```

How You Can Select Data into Reference or Array Variables

Regardless of whether you retrieve one row or multiple rows, when you retrieve data into a reference or array variable, the select statement must see the individual attributes of the variable rather than the variable as a whole.
How You Can Use the Select Statement Generically

You can use a standard select statement when you do not know the database table until runtime, provided you know the column names. Assume, for example, that you have several identically structured tables—po91, po92, and po93—that store purchase order information. You can write a single select statement to access all the purchase order tables as follows:

1. Use a variable to reference the table name.
   You must notify the DBMS to interpret the variable representing the table name by preceding the variable with a leading colon (\:). 

2. Supply the table name at runtime.

The following statement selects data from table tblname into known fields on the form. The actual table name is substituted at runtime.

```sql
select ponum as :field1, podate as :field2,
     vendorname as :field3
from :tblname;
```

If you also do not know the names of the columns until runtime, you can build a valid select statement using the execute immediate statement. However, you must know the name and data type of the variables that are to receive the data. If you do not have the necessary information about the target variables, create an SQLSelect object or a query object to build the retrieval statement dynamically.

For more information about dynamic query creation, see How You Can Access a Database with DataStream Objects (see page 165) and How You Can Use Query Objects (see page 177). For more information about the execute immediate statement, see the Language Reference Guide online help.

Errors Originating in Called 3GL Procedures

Many OpenROAD applications call 3GL procedures. If the 4GL calls an Embedded SQL routine that does not switch sessions, the status from the last query issued by the Embedded 3GL routine is available to 4GL. After the 3GL routine returns control to the 4GL, the 4GL can send a user event to a standard error-handling procedure to process the error.
For example, assume that your application uses the SendUserEvent method and a common error-handling procedure to deal with DBMS errors. Assume also that your application calls a 3GL procedure that includes the following Embedded SQL statements:

```
select ...
exec sql inquire_sql (:dbmserror = dbmserror);
if (dbmserror != 0)
   return;
exec sql insert ...
exec sql inquire_sql (:dbmserror = dbmserror);
if (dbmserror != 0)
   return;
...
```

When control is returned to the 4GL, the following code fragment obtains the error number returned by the 3GL procedure and uses the SendUserEvent method to send the error number to its 4GL error-handling procedure:

```
if curframe.dbsession.dbmserror != 0 then
   errorproc.SendUserEvent
      (eventname = '3gleror',
       messagevarchar = dbmserror);
endif;
```

If the 4GL calls a 3GL procedure that produces an error in an exec 4GL statement (such as using an invalid attribute name or trying to fetch a character-valued global variable into an integer-valued host variable), error handling must be done in the 3GL procedure. 4GL has no constructs to examine whether errors occurred.

### How You Can Use the Insert Statement

The insert statement adds new rows to a database table. The values that populate the new rows can come from any of the following places:

- **Variables associated with the current frame:**
  - Simple variables
  - Attributes of reference variables
  - Columns in a dynamic array
- **Global variables**
- **A subselect**
- **Constants**
- **Expression results**
How You Can Access a Database with Standard SQL Statements

Each insert statement adds one row to the database. For example, the following statement adds a single new customer record to the customer database table:

```sql
insert into customer (acctno, cphone, cname,
           addr, ccity, cstate, czip, cdistrict, cstatus,
           cacctbal)
values (:acctno, :cphone, :cname, :caddr,
           :ccity, :cstate, :czip, :cdistrict,
           :cstatus, :cacctbal);
```

You may insert several rows into a table using a loop. For example:

```sql
i = 1
for i = 1 to new_customers.lastrow do
   insert into customer
      (acctno, cphone, cname)
   values
      (
         :new_customers[i].acctno,
         :new_customers[i].cphone,
         :new_customers[i].cname
      )
endfor;
```

For a discussion of processing arrays using the row state attribute, see Working with Arrays, Table Fields, and Collections (see page 205).

How You Can Use the Update Statement

The update statement replaces values in specified columns of a database table with new values. These new values can come from any of the following places:

- Variables associated with the current frame
- Columns from another table
- The result of an evaluated expression
- Global variables
- Constants
- A subselect

You can update all the rows in a table. For example, assume that you want to change to five-digit account numbers for customers. The following statement updates all existing account numbers in the customer table by adding 10000 to each number:

```sql
update customer set acctno = acctno + 10000;
```
Using the where clause, you can update only selected rows. The following example updates only the customer who meets the criteria defined in the where clause:

/* Update customer's account balance */
update customer
set cacctbal = cacctbal - :charges
where acctno = :checkout_form.acctno;

There are also query object and cursor versions of the update statement. Both of these update operations modify the values in the current row when the statement is executed. For examples of query object and cursor update statements, see How You Can Use Query Objects (see page 177) and How You Can Use Cursors to Access the Database (see page 157).

**How You Can Use the Delete Statement**

The delete statement removes rows from a database table. You can delete all rows or, by using a where clause, only specified rows.

For example, assume that you want to remove all customers from the customer table whose accounts are closed. (The value for a closed account is stored in the CLOSED constant.) The following statement performs this task:

delete from customer where cstatus = CLOSED;

There are also query object and cursor versions of the delete statement. Both of these delete operations remove the current row when the statement is executed. For examples of query object and cursor delete statements, see How You Can Use Query Objects (see page 177) and How You Can Use Cursors to Access the Database (see page 157).

**How You Can Use Other SQL Statements**

There are two types of SQL statements that you can code directly in a 4GL script:

- **Data definition statements**

  These include statements that create data structures, such as tables or views, rules, and database events.

  All data definition statements participate in transactions. Using one of these statements initiates a transaction if none was open, and any work performed by these statements is affected by the commit and rollback statements.

  For more information about transactions, see How You Can Manage Transactions (see page 182).
- Statements that do not access database tables
  
  Two examples of this type are the set statement (including set autocommit and set lockmode) and the inquire_sql statement. The set statements do not take part in transactions and cannot be issued inside them.

  For a list of SQL statements and their syntax, see the Language Reference Guide online help.

How You Can Use Database Procedures

OpenROAD scripts can call database procedures, which are collections of statements managed as a single unit by the Ingres DBMS. Using database procedures provides another way for OpenROAD scripts to access the database.

For more information about calling a database procedure, see Writing Scripts and Procedures (see page 67).

Creating a database procedure within an OpenROAD application requires using the execute immediate statement.

For more information about using the execute immediate statement, see How You Can Use the Execute Immediate Statement (see page 155) and the Language Reference Guide online help.

How You Can Use the Execute Immediate Statement

You can use the execute immediate statement to perform any of the following tasks:

- Execute a variety of SQL statements that the 4GL code constructs at runtime.

- Execute SQL statements that contain legal SQL syntax that is not recognized by the OpenROAD compiler, such as:
  - Creating a database procedure
  - Issuing statements to a non-Ingres server or a version of the Ingres server that supports different SQL syntax
The syntax of the execute immediate statement is:

```sql
execute immediate statement_string [into variable {, variable}]
```

**statement_string**

Specifies a string constant, a varchar expression, or a StringObject reference variable value. It must adhere to the following restrictions:

- It must evaluate to a single SQL statement.
- It cannot contain an open or fetch statement or any of the exceptions for the execute immediate statement.

Assume, for example, that you want to create a temporary table with a unique name at runtime. Do the following to create the table:

1. Declare two variables, one to contain the table name (tname) and one to contain the I parameter (vstring):
   ```sql
tname = varchar(32);
vstring = varchar(500);
```
2. Build the temporary table name from the user's login, followed by numbers:
   ```sql
   select :user_name = dbmsinfo('username');
tname = user_name + '123';
   ```
3. Load the create table statement into vstring and run it using the execute immediate statement:
   ```sql
   vstring = 'create table ' + tname + '(name varchar(50), addr varchar(100))';
   execute immediate :vstring;
   ```

When the program runs the execute immediate statement in Step 3, it executes the statement contained in vstring, creating the specified table.

For more examples of using the execute immediate statement, see the *Language Reference Guide* online help.

**How You Can Use the Into Clause with a Select Statement**

You must use the into clause if the `statement_string` is a select statement. When the statement is executed, the values retrieved by the select statement are placed in the specified variables. The data types of the variables must match those in the target list of the select statement. The number of columns selected must also match the number of targets specified.

For example, assume that your application contains the following code:

```sql
statement = 'Select title, vidno from video';
```
The following execute immediate statement runs the select statement and puts the values returned in the vidtitle and vidno variables:

```sql
execute immediate :statement into :vidtitle, :vidno;
```

If you do not know the names and data types of the target variables until runtime, create an SQLSelect or a query object to build the retrieval statement dynamically. For more information about dynamic query creation, see How You Can Access a Database with DataStream Objects (see page 165) and How You Can Use Query Objects (see page 177).

### How You Can Use Cursors to Access the Database

Cursors are useful for:

- Bringing only part of a result set across the network where low communications bandwidth is an issue
- Performing an unrelated operation in the midst of processing returned rows (which can also be performed using a select loop)
- Performing operations on returned values that span two or more event blocks
- Determining which of several select statements in a fetch loop to execute at runtime, for example:

```sql
if frame_mode = MODE_SALES then
  open cursor1 for select salesname as name,
         salesquota as amount
  from sales where region = :region order
  by amount;
elseif frame_mode = MODE_MGRS then
  open cursor1 for select mgrname as name,
         budget as amount from managers
  where business_unit = :bus_unit
  order by name;
else
  open cursor1 for select staffname as name,
         salary as amount
  from staff where staff_code = :staff_code
  ORDER BY Amount:
endif;
```

```sql
i = 0;
curs_state = CS_CURRENT;
while (curs_state = CS_CURRENT) do
  i = i + 1;
  fetch cursor1 into :tf[i].name = name,
    :tf[i].amount = amount;
  curs_state = cursor1.state;
endwhile;
/* number of rows fetched = i-1 */
close cursor1;
```
Providing better performance than a select loop when the application plans
to limit the number of rows selected

Select loops cache every row that qualifies for the select. Setting
DBSessionObject's PreFetchRow attribute when using a read-only cursor
lets you limit the number of qualifying rows actually fetched.

For example, suppose you want to fill a tablefield with the top 10 salespeople
(by sales) from a particular region. Using a select loop, you would code:

```
i = 1;
select :tf[i].name = name, :tf[i].quota = quota,
    :tf[i].sales = sales
from salespeople where region = :region
order by sales desc
{i = i + 1;
    if i > 10 then
        endloop;
    endif;
}
```

This technique is inefficient because all salespeople rows for the specified
region are returned to OpenROAD from the database when only 10 are
desired. The following example is a more efficient way to code this situation:

```
CurFrame.DBsession.PrefetchRows = 10;
open cursor1 for select name as name,
    quota as quota,
    sales as sales
from salespeople where region = :region
order by sales desc for readonly;
i = 0;

CURS_STATE = CS_CURRENT;
while (CURS_STATE = CS_CURRENT) do
    i = i + 1;
    fetch cursor1 into :tf[i].name = name,
        :tf[i].quota = quota, :tf[i].sales = sales;
    CURS_STATE = CURSOR1.state;
    if i > 10 then
        endloop;
    endif;
endwhile
/* number of rows fetched = i-1 */
close cursor1;
```

The PreFetchRows attribute enables you to tune the performance of cursors
when you have an approximate idea of the number of rows the cursor will
return.
How You Can Use the CursorObject System Class

In OpenROAD, cursors are implemented as objects of the CursorObject system class. Using a cursor consists of the following steps:

1. Declare a CursorObject reference variable.
2. Open the cursor, specifying whether the cursor can update or delete rows that are fetched.
3. Fetch a row and move the values in the row into variables.
4. Optionally update or delete the row on which the cursor is positioned, using the values loaded into the variables.
5. Generally, repeat Steps 3 and 4 until all the rows in the result table have been processed.
6. Close the cursor.

It is only necessary to declare a cursor once in a program. You can open it more than once, if you close it between each opening.

Note: Opening a DataStream object in QY_CURSOR mode provides the same functions as using a cursor. For a discussion of using DataStream objects, see How You Can Access a Database with DataStream Objects (see page 165).

CursorObject Attributes

The CursorObject class has three attributes that provide status information:

**State**

Contains information about the state of the cursor, such as whether it is open or closed

**RowCount**

Indicates how many rows have been fetched successfully by the cursor since the cursor was opened. When a cursor is opened, RowCount is set to 0. Each subsequent successful fetch statement for that cursor increments RowCount by 1.

When you close the cursor, the RowCount attribute is not reset to 0; it retains whatever value it had before the close statement executed. It is reset only if you reopen the cursor.

**DBSession**

Identifies the session in which the cursor was opened. After you open a cursor, all subsequent cursor operations for that cursor automatically take place in the session in which the cursor was opened. If the frame is working in a different session, OpenROAD automatically switches sessions for the cursor operation and switches back when the cursor statement completes.
**How You Can Declare a Cursor**

Before you can open a cursor, you must create the CursorObject object for the new cursor by declaring a global or local reference variable of type CursorObject. The syntax is:

```java
cursor_var_name = CursorObject
```

For example, the following code line declares a reference variable named cust_cursor that points to a cursor object:

```java
cust_cursor = CursorObject;
```

**How You Can Open and Position a Cursor**

You can use the open statement to perform the following operations:

- Open a cursor in the database, specifying whether:
  - The cursor can update or delete rows that are fetched
  - Updates should be deferred or direct
- Associate a select statement with the cursor and position the cursor immediately before the first row in the select statement's result table
- Set the cursor's State attribute to CS_OPEN

To access the rows in the result table, use the fetch statement. This statement performs the following tasks:

- Advances the cursor one row in the result table
- Places the values in the indicated columns into the specified variables
- Sets the cursor’s State and RowCount attributes

Each execution of a fetch statement fetches one row of the result table. The syntax lets you retrieve the values from every column in the result table or from some subset of the columns.

If you fetch all of the retrieved values in the order specified in the select statement, you do not need to name the columns in the fetch statement.

For example, assume you issued the following open statement to retrieve the values in eight columns from the customer table:

```java
open cust_cursor for select acctno, cphone, cname, caddr, ccity, cdistrict, cstatus, cacctbal
from customer for direct update of acctno, cphone, cname, caddr, ccity, cdistrict, cstatus, cacctbal;
```
To fetch the values from all eight columns in the same order specified in the select statement, use the following statement:

```sql
fetch cust_cursor into :acctno, :cphone,
   :cname, :caddr, :ccity, :cdistrict,
   :cstatus, :cacctbal;
```

Because the variables in the fetch statement match the number, data type, and order of specification of the select statement's expressions, you need not specify column names. The fetch statement puts the retrieved values (one row at a time) into the specified variables.

Similarly, if you fetch a subset that begins with the first column returned by the select, continues in the order listed in the select, and stops before all are listed, it is not necessary to specify column names. For example, the following statement fetches the first, second, and third columns (acctno, cphone, cname) returned by the select:

```sql
fetch cust_cursor into :acctno, :cphone, :cname;
```

However, whenever you fetch column values out of order (that is, do not start with the first selected column and continue in the order of the select), you must identify the column. For example, to fetch only the customer's name and status using the example open statement, specify the column names:

```sql
fetch cust_cursor into :cname=cname,
   :cstatus=cstatus;
```

Whether you fetch all or some of the selected column values, the variables that receive the column values and their associated columns must have compatible data types.

For more information about the open and fetch cursor statements, see the Language Reference Guide online help.

**How You Can Update or Delete a Row**

After you have fetched a row, you can update values in the row or delete the row if the cursor was opened to allow for updates and deletes. There are versions of the update and a delete 4GL statement specifically for use with cursors.
For example, the following code updates the customer record on which the cursor is currently positioned:

```sql
on click update_button =
begin
/* Actually update the customer in the database */
if menu.get_menu.initial_account > 2 then
  update customer
    set acctno = :customer.acctno,
            cphone = :customer.cphone,
            cname = :customer.cname,
            caddr = :customer.caddr,
            ccity = :customer.ccity,
            cstate = :customer.cstate,
            czip = :customer.czip,
            cdistrict = :customer.cdistrict,
    where current of cust_cursor;
...
```

The following code provides an example of deleting the row on which the cursor is positioned:

```sql
on click delete_button =
begin
  if menu.get_menu.initial_account > 2 then
    delete from customer
    where current of cust_cursor;
...
```

For more information about the cursor version of the update and delete statements, see the Language Reference Guide online help.

### How You Can Close a Cursor

When the program has processed all the required rows in the result table, close the cursor. Closing a cursor:

- Sets the cursor’s State attribute to CS_CLOSED
- Makes changes visible to the rest of the program (if the cursor was a deferred update cursor)
- Releases any locks held by the select statement associated with the cursor (if autocommit is on)

  For a description of autocommit, see How Running with Autocommit On Works (see page 187).

If you reopen a closed cursor, the cursor is repositioned at the top of the result table.
The following statement provides an example of closing a cursor:

```sql
close cust_cursor;
```

In addition to being closed explicitly by the close statement, cursors are also closed implicitly when the variable representing them goes out of scope. For example, if the cursor object is referenced by a variable declared locally to a frame and the frame closes without explicitly closing the cursor object, the cursor object is closed automatically.

**How You Can Manage Transactions with Cursors**

Each cursor must be opened and closed within a single transaction. Moreover, the state of the transaction affects the state of the cursor. For example, a cursor is automatically closed in either of the following cases:

- Its transaction is closed by either a commit or rollback.
- Its transaction is aborted by an error.

Some errors (for example, deadlock, logfull, and disk full) do not abort an entire transaction but do close any open cursors. If an error occurs while a cursor is open, you should roll back the transaction and begin it again.

If you are running with autocommit on (each database statement is a separate transaction), a cursor is considered one transaction from the time it is opened until it is closed. Therefore, when autocommit is on and a cursor is open, you can issue only cursor statements to access the database. If you issue any other database access statements, the open cursor is closed automatically.

For an explanation of using autocommit, see How Running with Autocommit On Works (see page 187).

You can use the inquire_sql statement with the transaction parameter to determine the transaction state. For more information about the inquire_sql statement, see Handling Database Errors (see page 197) or the online help.

**How You Can Use Methods That Access the Database**

Three system classes, BitmapObject, StringObject, and DataStream have methods that access the database. The BitmapObject and StringObject methods are:

**InsertIntoDB**

Places the associated object in the database
UpdateInDB
Replaces the object in the database

DeleteFromDB
Removes the object from the database

Note: None of the BitmapObject and StringObject methods should be issued with autocommit set to ON.

The DataStream system class defines the following methods that access the database:

Open
Opens the query for SQLSelect and query objects

SetCols
Sets up the Columns array for an SQLSelect object

Close
Closes the query (only when the DataStream object was opened in QY_CURSOR or QY_DIRECT modes)

NextRow
Retrieves the next row of data (only when the DataStream object was opened in QY_CURSOR or QY_DIRECT modes)

The QueryObject system class, which is a subclass of DataStream, has the following additional database access methods:

DBInsert
Places a new row into the database

DBUpdate
Replaces a specified row in the database

DBDelete
Removes a specified row from the database

Because they access the database directly, these methods participate in transactions. Not only can using one of these methods initiate a transaction, but the methods are affected also by commit or rollback statements. Because the transaction semantics are not handled directly by these methods, you must write the transaction management code. For more information about handling transactions opened by system class methods, see How You Can Manage Transactions (see page 182).
For a discussion of using BitmapObject objects and StringObject objects, see the chapter Working with Images and Text Strings (see page 277). For a discussion of using DataStream objects, see How You Can Access a Database with DataStream Objects (see page 165) and How You Can Use Query Objects (see page 177).

How You Can Access a Database with DataStream Objects

OpenROAD provides the DataStream system class to enable applications to retrieve data from the database when you do not know the target fields or variables until runtime. In addition to allowing dynamic changes to the query, DataStream objects allow movement backward as well as forward through the retrieved data set.

The DataStream system class, in particular its SQLSelect and QueryObject subclasses, provide the only means in OpenROAD to make dynamic changes to a query or allow backwards movement in a data set. Standard select statements or those built using the execute immediate statement cannot be used to retrieve data when the columns are unknown until runtime.

The primary difference between the SQLSelect and QueryObject system classes is that the QueryObject class provides methods for adding, changing, and removing data in the tables underlying the query. SQLSelect objects are used only for retrieving data from the database.

These two system classes also differ in the way you create them:

- You provide an arbitrary SQL select statement as the query for an SQLSelect object.
- You provide a decomposed view of the query for a QueryObject object by setting various attributes; OpenROAD builds the SQL select statement.

Both the SQLSelect and QueryObject classes have many attributes, but the OpenROAD programmer uses only a subset of these attributes, as described in the following sections.

Using either of these classes requires the following steps:

1. Create the SQLSelect or QueryObject object.
2. Provide the query or set the necessary attributes to build the query.
3. Specify target fields or variables.
4. Invoke the Open method to retrieve the data.
5. Invoke additional methods to display the data.
6. Invoke the Close method to close the SQLSelect or QueryObject object.
How You Can Access a Database with DataStream Objects

The following subsections explain using statically created SQLSelect objects and QueryObject objects. For a discussion and example of using a query object, see How You Can Use Query Objects (see page 177).

How You Can Select a Mode for a DataStream Object

The DataStream's Open method performs the actual data retrieval. One of its arguments, the mode of operation, determines the query's behavior. The various modes are:

- **QY_CACHE**
  Specifies that the retrieved data set is cached, allowing previous navigation and random seek; requires the Load and NextRow methods to display data.

- **QY_CURSOR**
  Specifies that the retrieved data set uses a DBMS cursor, allowing nested queries; requires the Load and NextRow methods to display data.

- **QY_DIRECT**
  Specifies that the retrieved data set is returned as in a select loop; precludes nested queries because a single select statement is in effect between the Open and Close of the DataStream object; requires the Load and NextRow methods to display data.

- **QY_ARRAY**
  Specifies that the retrieved data set is immediately loaded into the specified array or table field by the Open method (making the Load and NextRow methods meaningless); requires the UpdField method to refresh the display if loading into a table field.

Using QY_ARRAY mode enhances performance if you are loading a table field because an entire array of data is loaded. However, to take advantage of the enhanced performance, the class of the array being loaded must have attribute names that match the names of the columns in the query (which can differ from the names of columns in underlying database tables due to “as” clauses).

How You Can Use SQLSelect Objects

The following text uses examples that might be included in a frame that retrieves information from a customer table.
How You Can Specify the Query

When creating an SQLSelect object, you must first specify the query, which involves:

- Creating the SQLSelect object
- Providing an SQL query
- Providing specifications for at least one target field or variable for each column selected

Except for building the where clause, you might set up the query in an initialize block.

The following code from the initialize block declares the necessary variables and assigns the query text to the selectstring variable:

```plaintext
initialize () =
  declare
    ss = SQLSelect;
    selectstring = varchar(200) not null;
    wclause = varchar(60) not null;
    ...
  enddeclare
begin
  selectstring = 'select acctno, cphone, cname, caddr, ccity,' +
                  ' cstate, czip, cdistrict, cstatus, cacctbal' +
                  ' from customer';
```

Because the ss SQLSelect object is declared without specifying default null, it is created at the same time that it is declared. If your application may never need the SQLSelect object, declare it with default null and later, when the SQLSelect object is required, explicitly create it using the Create method. For examples of using the Create method, see Creating Dynamic Frames (see page 363).

In addition to declaring necessary variables and providing the query text, specifying the query requires assigning the value in each database column to an SQLSelect target.

Each column's value must be assigned to the SQLSelect's Columns attribute, which is an array of the QueryCol system class. The Columns attribute must contain at least one row for each column to be retrieved from the database.

The syntax for assigning a column value to an SQLSelect target is:

```
sqlselect_name.Columns[n].Targets[m].Expression = field_name | var_name | value
```
When you assign each column participating in the query to a row in the Columns attribute, you assign it to at least one target in the Targets array (which is an attribute of the Columns array attribute). Specifying more than one target for a column lets you assign it to more than one field or variable in the frame.

For example, you can assign a column to different targets if you want to:

- Put a column's value into both a simple field and a table field
- Store a column's original value (for comparison to changes made later by the user)

Because the Targets attribute is an array of type QueryParm, it contains the following attributes, two of which are required for using an SQLSelect object. The following target attributes are required for SQLSelect objects:

**Expression**

Contains the name of the target field or variable, such as "field1" or "tbl[i].col1"

**IsDBHandleField**

Set to TRUE if the column's value is to be retrieved as a DBHandle and the target expression evaluates to a bitmap object or string object to be loaded using this DBHandle

**IsSelectTarget**

Set to TRUE if the column is used as a target for a select statement; defaults to FALSE

Because the IsDBHandleField attribute defaults to FALSE, specifying it is required only for columns whose values are retrieved as a DBHandle. Setting the IsSelectTarget attribute is required even though the select clause includes every column for selection.

The index of the Columns array is incremented for each column selected from the database. The index of the Targets array is incremented only if the column is to be sent to more than one target field or variable. Each column must be sent to at least one target.
Continuing our example, the following code provides the name of each target field to the Expression attribute and sets the IsSelectTarget attribute to TRUE for each column:

```csharp
/* Fill in the Expression and IsSelectTarget for every column to be retrieved. */
ss.Columns[1].Targets[1].Expression = 'customer.acctno';
ss.Columns[2].Targets[1].Expression = 'customer.cphone';
ss.Columns[3].Targets[1].Expression = 'customer.cname';
ss.Columns[4].Targets[1].Expression = 'customer.caddr';
...
for i = 1 to ss.Columns.LastRow do
    ss.Columns[i].Targets[1].IsSelectTarget = TRUE;
endfor;
end;
```

The where clause can be specified to select a subset of rows from the table. The following code illustrates getting either all accounts or only accounts that are closed (cstatus = 0):

```csharp
if (get_all_accounts = TRUE) then
    /* Fetch all customers */
    wclause = '  order by cname';
else
    /* Fetch customers whose accounts are closed. */
    wclause = '  where cstatus = 0 order by name';
endif;
```

The text of an SQLSelect object’s query is stored in its Query attribute. The data type of the Query attribute is StringObject. You can load the Query attribute directly with the text of a select statement or create the text from variables that contain pieces of the select statement.

The following statement sets the value of the Query attribute from its two varchar variables, selectstring and wclause:

```csharp
ss.Query.Value = selectstring + wclause;
```

Although the complete select statement is assigned to the Query's Value attribute, the actual data retrieval occurs only when you invoke DataStream's Open method.
How You Can Run the SQLSelect Query

To execute the query after specifying it, invoke the Open method. If you open the SQLSelect object in QY_ARRAY mode, each column's value goes directly to the Expression attribute of the SQLSelect object's target array. If you mapped the Expression attribute for each column to a specific field on the form, each column's value is displayed on the form.

If, however, you open the SQLSelect object in any mode but array mode, the following additional methods are required to display the data:

NextRow
Retrieve the data row by row from the database into internal buffers

Load
Moves data from the internal buffers into the target fields on the form

Opening an SQLSelect object requires setting several parameters in addition to specifying the query mode. The Open method has the following parameters:

QueryMode
Data Type: integer
(Required.) Specifies one of the following modes:
- QY_ARRAY
- QY_CACHE
- QY_CURSOR
- QY_DIRECT
Default: None

CheckCols
Data Type: integer
(Optional.) Checks the number of columns specified in the Query text with the number of columns in the Columns array. When FALSE, populates columns in the Columns array if they have not been explicitly set.

Default: FALSE

IsRepeated
Data Type: integer
(Optional.) Specifies whether to make a repeat query. Set to TRUE to make a repeat query.

Default: FALSE
MaxRows

**Data Type:** integer

(Optional.) Specifies a maximum number of array rows to populate (if greater than 0). Exceeding MaxRows causes an error (applies only to QY_ARRAY mode).

**Default:** 0

Scope

**Data Type:** Scope

(Required.) Specifies the scope for evaluation of expressions in the Columns array Targets attribute.

**Default:** Value of the current Scope attribute

It is always necessary to set the scope for an SQLSelect object or QueryObject object. Scope can be set as a parameter to the Open method or by setting the DataStream’s Scope attribute directly. The following code sets the SQLSelect object's Scope attribute to the scope of the current frame:

```plaintext
ss.Scope = CurFrame.Scope;
if ss.Scope != CurFrame.Scope then
    /* error */
endif;
```

**Note:** Because setting the scope causes compilation of expressions in the SQLSelect object, errors during compilation could prevent the scope from setting as expected. It is recommended, therefore, that you check for appropriate setting of the scope while you are developing the application and make the necessary changes to your 4GL code.

Rather than setting the scope by setting the Scope attribute, you can set it using the Open method. The following code opens the SWLSelect object in cache mode, with CheckCols set to TRUE, and specifies the current frame as the scope:

```plaintext
status = ss.Open
(    
        querymode = QY_CACHE,
        checkcols = TRUE,
        scope = CurFrame.Scope
    )
```
if (status != ER_OK) then
    /* There was an error. */
    rollback;
    /* Notify user of error. */
    ...
    resume;
endif;
commit;

if (ss.ErrorNo != 0 and ss.IsDBError = TRUE) then
    /* There was a database error */
    rollback;
    /* Notify user of error. */
    ...
    resume;
endif;

The value returned by the Open method is ER_OK if no errors occur in executing the query. If no errors have occurred, the example code commits the transaction opened by executing the query.

Like any SQL statement, SQLSelect objects (and QueryObject objects) participate in a transaction. Because closing the SQLSelect object does not close the transaction, your application must explicitly commit or roll back the transaction.

Because all the data retrieval is done at once when the Open method is invoked, the code in the previous example commits the transaction immediately.

After committing the transaction, the sample code checks for errors using the following DataStream attributes:

**DataStream attributes**

**ErrNo**

Provides the error number of the last operation, generated either by:

- The DBMS—for example, a column specified as a target does not exist in the table
- The application—for example, the 4GL code provides an illegal scope or contains expressions that do not compile

**IsDBError**

Set to TRUE if the error is a DBMS error.

If the SQLSelect has been opened successfully, this frame uses the SQLSelect’s MaxRow attribute to determine the next operation. The MaxRow attribute contains the number of rows retrieved. If the value of the MaxRow attribute is not zero, you can invoke the NextRow and Load methods to retrieve data.
The following code performs these operations:

```csharp
if (MaxRow > 0) then
    status = ss.NextRow();
    status = ss.Load();
endif;
```

Because the ss SQLSelect object was opened in QY_CACHE mode, data was retrieved from the database table into the cache. The data is not directly displayed on the form. Therefore, the example code invokes the NextRow method to move the next row (in this case the first row) from the cache to internal buffers, and the Load method to move it from internal buffers to fields (or variables) in the frame.

**How You Can Get Next and Previous Rows**

Suppose that your frame has buttons labeled Next and Previous. The following code shows how you can use these buttons to fetch the next and previous rows:

```csharp
/*
** Initialize the buttons.
*/
if (ss.CurRow >1) then
    field(prev_btn).CurBias = FB_CHANGEABLE;
else
    field(prev_btn).CurBias = FB_DIMMED;
endif;
if (ss.CurRow <ss.MaxRow) then
    field(next_btn).CurBias = FB_CHANGEABLE;
else
    field(next_btn).CurBias = FB_DIMMED;
endif;

on click next_btn =
{
    ss.NextRow();
    ss.Load();
    /*
** Display the data.
*/
    ...
    /*
** See if there are more rows.
*/
    if (ss.CurRow >= ss.MaxRow) then
        field(next_btn).CurBias = FB_DIMMED;
    else
        field(next_btn).CurBias = FB_CHANGEABLE;
    endif;
}
```
on click prev_btn =
{
    ss.PrevRow();
    ss.Load();
    /*
     ** Display the data.
     */
    if (ss.CurRow <= 1) then
        field(prev_btn).CurBias = FB_DIMMED;
    endif;
    field(next_btn).CurBias = FB_CHANGEABLE;
}

A third DataStream method used to get rows is the FetchRow method. This method sets the current row in the cache to the row specified in the RowIndex parameter. The value of this parameter must be an integer.

For example, the following code sets the fourth row in the cache as the next row to be loaded:

ss.FetchRow(rowindex = 4);

To position the current row on the first row of the cache, set the RowIndex parameter to "1", for example:

ss.FetchRow(rowindex = 1);

Both the PrevRow and FetchRow methods are valid only in QY_CACHE mode.

**How You Can Close the SQLSelect Object**

To close an SQLSelect object, invoke DataStream's Close method. It is necessary to close the SQLSelect object before changing any of the query parameters and reexecuting it. Attempting to close an SQLSelect object that is not open, or open one that is still active, causes an error.

The following code closes the SQLSelect object:

if ss.State = QS_ACTIVE then
    ss.Close();
endif;

When you close an SQLSelect object that was opened in cache mode, you clear it of all data. Therefore, do not close the query until the user has selected an operation that uses different data.
How You Can Use Parameterized Queries

To allow greater flexibility, the text of the SQLSelect's Query attribute can contain parameter substitutions. There are two types of query parameters:

**Parameters evaluated before the query is sent to the DBMS server**

These parameters involve straight textual substitution and can appear anywhere in the text, even inside quoted strings and substituting for SQL keywords and operators. They cannot, however, be used with repeat queries. These parameters begin with a leading ampersand (&).

The following example contains parameterized column names and a parameterized table name:

```plaintext
ss.Query.Value = 'select &(col1), &(col2),
            &(col3) ' + 'from &tablename';
```

The variables representing the column names and table name must be visible in the scope specified in the SetCols or Open method. When the ampersand is followed by an expression, the expression must be enclosed within parentheses as shown in the following example:

```plaintext
ss.Query.Value = 'select &(col1 + col2/2) ' +
            'from &tablename';
```

**Parameters interpreted by the Ingres server**

These parameters can appear only where the Ingres DBMS accepts parameters (that is, in the same places as constants). They begin with a leading colon (:).

The following example evaluates an expression and uses the result as the value for the where clause:

```plaintext
select number from infotbl
   where number = :(var1 + var2);
```

The following example shows how you can set the Value attribute of an SQLSelect with a parameterized where clause:

```plaintext
initialize () =
declare
    ss = SQLSelect;
    wclause = varchar(60) not null;
...
enddeclare
begin
    ss.Query.Value = 'select * from customer
            &(wclause) ' + 'order by cname';
```

You can set the where clause later. The value of the where clause is assigned after the SQLSelect is opened and evaluated and before the query is sent to the DBMS server.

For a discussion of setting the Query's Value attribute, see How You Can Specify the Query (see page 167).
How You Can Use the SetCols Method to Specify the Query

Note that in addition to using parameter substitution, the preceding query differs from previous examples in that it does not specify the columns to be retrieved. Instead it uses the "select *" syntax to retrieve all columns from the customer table.

After loading the SQL text into the Query attribute, you can use the SetCols method to get relevant column information for each column and to initialize the Columns array. The SetCols method also checks the query for syntax errors.

The SetCols method gets information from the database table and assigns it to relevant attributes in the SQLSelect’s Columns array. As an array of type QueryCol, the Columns attribute contains the following information:

**AsName**
- Specifies the column name used in the select statement to see the retrieved column (for example, select name as EmpName ...); used in group by or order by clauses

**ColumnName**
- Specifies the name of the database column; required for QueryObject objects but not SQLSelect objects

**DataType**
- Specifies the data type of the database column; supplied or checked by the SetCols and Open methods

**DataTypeLength**
- Specifies the length of the data type; supplied or checked by the SetCols and Open methods

**DataTypeNullable**
- Set to TRUE if the data type is nullable; supplied or checked by the SetCols and Open methods

**Targets**
- Specifies an array of field mappings for each column

**Value**
- Specifies the current column value (valid only if a row has been fetched)

The attributes that contain data type information are assigned directly by invoking the SetCols or Open methods. In addition, if it has not already been filled in, the AsName attribute is also assigned a value by invoking these two methods.
How You Can Use Query Objects

To allow users to change, delete, or add data to a database, use a query object rather than an SQLSelect object. Unlike the process for creating an SQLSelect object, which requires you to provide the SQL select statement and the target information, you cannot create the select statement for a query object (QueryObject.Query.Value is read-only).

Creating a query object requires you to specify data to be retrieved and manipulated. The query object builds the select statement for you.

Creating a query object requires setting attributes and invoking methods of the same system classes involved in creating an SQLSelect object. The system classes used in creating both SQLSelect and QueryObject objects are:

- DataStream
- QueryCol
- QueryParm
- SQLSelect

In addition, creating a query object involves the following two system classes:

- QueryObject
- QueryTable

For a description of these system classes, see the Language Reference Guide online help.

Moreover, setting the Targets attribute for a query object is somewhat more complicated than setting it for an SQLSelect object because there are more relevant attributes of the Targets array to be set.

The Targets attribute, which is an array of type QueryParm, contains the following attributes, all of which are required for creating a query object:

**Expression**

Contains the name of the target field or variable, such as “field1” or “tbl[i].col1”

**Default:** None

**IsFileHandleField**

Set to TRUE if the column’s value is to be retrieved as a FileHandle and the target expression evaluates to a bitmap object or string object to be loaded using the FileHandle

**Default:** FALSE
**IsDBHandleField**
Set to TRUE if the column's value is to be retrieved as a DBHandle and the target expression evaluates to a bitmap object or string object to be loaded using this DBHandle

**Default:** FALSE

**IsInsertTarget**
Set to TRUE if the column is used as a target for an insert statement

**Default:** FALSE

**IsSelectTarget**
Set to TRUE if the column is used as a target for a select statement

**Default:** FALSE

**IsUpdateTarget**
Set to TRUE if the column is used as a target for an update statement

**Default:** FALSE

**IsDeleteWhere**
Set to TRUE if the column is used as a where clause element for a delete statement

**Default:** FALSE

**IsUpdateWhere**
Set to TRUE if the column is used as a where clause element for an update statement

**Default:** FALSE

### Required Operations

The following operations are required to create a query object:

1. Provide the parameters for building the select clause:
   a. Load the name of each database column into the ColumnName attribute of DataStream's Columns array attribute.
   b. Supply the name of the target field or variable to the Expression attribute of the Targets attribute of the Columns array attribute (for every column selected from a database).
   c. Set the value of IsSelectTarget to TRUE for every column selected from the database.
d. Set the value of QueryParm's IsUpdateTarget to TRUE for every database column that allows updates.

e. Set the value of QueryParm's IsInsertTarget to TRUE for every database column that allows insertions.

2. Provide the parameters for building the from clause:

a. Specify the name of the table or tables being accessed in the QueryTable's TableName attribute.

b. Supply tables to the FromTable array attribute of the Columns array attribute (required only if a column appears in more than one table).

3. Provide the parameters for building the where clause (if desired):

a. For update operations, set the value of QueryParm's IsUpdateWhere to TRUE for every column that is to be part of the where clause.

b. For delete operations, set the value of QueryParm's IsDeleteWhere to TRUE for every column (usually PrimaryKey columns) that is to be part of the where clause.

c. For select operations, provide values for QueryObject's RunTimeWhere attribute (if desired).

4. Provide the parameters for building the order by clause (if desired):

a. Set the value of the Columns array attribute's OrderBy attribute to an appropriate sequence number; that is, set the primary column to 1 and the secondary column to 2, and so on. If the sort order for a particular column should be descending, then the sequence number's sign should be changed to negative (for example, -1 or -2).

b. Supply the name or names of database columns to the AsName attribute of the Columns array attribute for all columns to be used in the order by clause.

5. Provide the parameters for building the group by clause (if desired):

a. Set the value of the Columns array attribute's GroupBy attribute to an appropriate sequence number, such as 1 for the primary column and 2 for the secondary column.

b. Supply the name or names of database columns to the AsName attribute of the Columns array attribute for all columns to be used in the group by clause.

6. Provide the parameters for building the having clause by entering an appropriate string into QueryObject's HavingClause attribute (if desired).

To execute the query and manipulate the data, you use most of the same DataStream methods used in running SQLSelect objects. However, because you do not specify the select query, you cannot use the SetCols method in building a query object.
The relevant DataStream methods are:

**Open**
- Opens the query in the specified mode

**FetchRow**
- (Valid only for cache mode.) Moves to a specified row in the cache and puts its data into internal buffers

**NextRow**
- Retrieves the next row of data from the cache into internal buffers

**PrevRow**
- (Valid only for cache mode.) Backs up to the previous row in the cached sequence, positioning that row to be loaded into fields or variables

**Load**
- Moves one row of data from internal buffers into target variables, objects, or fields on the form

**Close**
- Closes the query

In addition to using methods defined for the DataStream system class, using a query object for database inserts, updates, and deletions requires methods defined for the QueryObject system class. These methods are:

**CommitToCache**
- Keeps data in the cache consistent with data in the database when insert, update, or delete operations are performed on cached data

**DBDelete**
- Performs deletions on database tables with columns mapped to variables or fields specified for deletion

**DBInsert**
- Performs insertions on database tables with columns mapped to variables or fields specified for insertion

**DBUpdate**
- Performs updates on database tables with columns mapped to variables or fields specified for update

When running in cache mode, the cached data set is frozen to the state of the data at the time the query was executed. If you make changes to the data in the database tables, you must also update the cache to keep data synchronized. To update the cache after changing data in the database, use the CommitToCache method.
How You Can Update Data with a Query Object

To update underlying database tables whose columns have been specified for update and mapped to the Columns array attribute, use QueryObject's DBUpdate method.

This method takes the following parameters:

**RowNumber**
- Specifies the row of the cache updated by the operation (used only if opened in QY_CACHE mode)
- **Default**: the current row

**IsRepeated**
- Set to TRUE if repeat queries are to be used
- **Default**: FALSE

**ZeroRowsIsError**
- If set to TRUE, a zero row result indicates an error; otherwise OK is returned
- **Default**: FALSE

**MaxRows**
- If nonzero, specifies the maximum number of rows that may be affected by the query. If more rows are affected than is specified by MaxRows, returns an error.
- **Default**: zero (unlimited)

How You Can Delete Data with a Query Object

Deleting data with a query object is more complicated than updating it. After removing the deleted row from the database, the code must check whether any rows remain in the cached data set.

For example, assume the data set originally contained four rows and the user just deleted the third row. After the third row is deleted from the cache, the fourth row becomes the third row and CurRow becomes 3. You can display this row directly by invoking the Load method. Note that you do not want to invoke the NextRow method because the deletion from the cache automatically performs that operation.

Assume instead that the user deleted the fourth and last row in the cache. In this case, you do not want to invoke the Load method because there is no current row. Instead, you want to make the current row point to an actual row.
How You Can Manage Transactions

A transaction is one or more database statements processed as a single, indivisible unit. A transaction begins with the first database statement executed after the application is started or after a previous commit or rollback statement. By default, the transaction continues until one of the following occurs:

- An explicit commit or rollback statement occurs
- A deadlock or log full error occurs
- The application stops

All locks taken during the transaction are held until the transaction is completed, and any changes made to a database by the transaction are not visible to other users until the transaction is committed.

How You Can Control Concurrency

No matter how many frames are opened or closed during a transaction, all locks are held until the transaction is completed. It is important, therefore, to design the application and its individual transactions to provide the best possible concurrency control.

Concurrency control refers to the management of shared data so that simultaneous users can access and update the data with little or no wait time while data integrity is maintained.

In general, data access and data integrity exist in an inverse relationship. If an application allows all users to access the data at will, then data integrity is severely threatened. If the application holds locks for long periods of time, for example by running as one continuous transaction, then it is difficult for users to share the data. Most application developers try to find a balance that lets multiple users access the data as quickly as possible yet maintain data integrity.

Sample Concurrency Control Application

This section presents an example application with code samples that illustrate the choices inherent in maintaining concurrency control. For instance, the application selects rows from a part table and, after users make their changes, updates the database. The entire application is one transaction, because the commit statement is not executed until the table has been updated. Keeping a single transaction throughout the application ensures data integrity, because the locks taken by the select statement are held until the changes are committed.
Although integrity is maintained by holding a single lock, concurrency is poor because no other user can access the data used by this application until the application has finished executing. For example, if the user running the application decided to go to lunch after loading the table field and before making any changes, the parts table would be inaccessible to any other business that may need it.

The following examples demonstrate two alternative approaches to this application.

**Coding the frame as a single transaction**

The following code illustrates this approach:

```plaintext
on click load_button,
on userevent 'load_data'=
begin
commit work; /* Start a new transaction */;
parttable.Clear(); /* Clear out array */
i = 1;
select :parttable[i].partno = partno,
    :parttable[i].partno_save = partno,
    :parttable[i].short_desc = short_desc
from part
begin
    i = i + 1;
end;
end;

on click update_button =
begin
    i = parttable.FirstRow();
    /* No error checking is done for simplicity of example */
    while i <= parttable.LastRow() do
        if parttable[i]._rowstate = RS_DELETED then
            repeated delete from part
            where partno = :parttable[i].partno;
        elseif parttable[i]._rowstate = RS_NEW then
            repeated insert into part
            (partno, short_desc)
            values
            (:parttable[i].partno,
             :parttable[i].short_desc);
        elseif parttable[i]._rowstate = RS_CHANGED
        then
            /* Does not allow direct update of key value */
            repeated update part
            set short_desc = :parttable[i].short_desc,
            partno = :parttable[i].partno
            where partno = :parttable[i].partno_save
        endif;
        i = i + 1;
    endwhile;
commit work;
end;
```
Dividing the operation into two transactions

The following code illustrates this approach:

```plaintext
on click load_button,
  on userevent 'load_data' =
  begin
    parttable.Clear(); /* Clear out array */
    i = 1;
    select  partno as :parttable[i].partno,
        partno as :parttable[i].partno_save,
        short_desc as
        :parttable[i].short_desc
    from part
    begin
      i = i + 1;
    end;
    commit; /* Commit added here */
  end;

on click update_button =
  begin
    i = parttable.FirstRow;
    /* No error checking is done for simplicity */
    /* of example */
    while i <= parttable.LastRow do
      ...
      <data integrity checks must be added here
      before rows are written back to the database>
    endwhile;
    commit;
  end;
```

The second example provides better concurrency by dividing the operation into two transactions. This is achieved by placing a commit statement after the select statement and another commit statement after the loop that updates the table. More than one user can access the data simultaneously, because the commit statement after the select statement has released the locks held by the select statement.

However, because the program must ensure that a second user has not changed the data already retrieved into the application's data set, making the application's data inconsistent with the data stored in the database table, you would need to add code to the second transaction to ensure data integrity. For example, adding a timestamp or a version column to the database table lets you verify that the data is the same when you update as when you retrieved.

You could enhance concurrency further by issuing a commit statement after each insert or update statement. However, committing after each database modification is useful only if the set of items changed are not considered to be a true logical unit of work.
How You Can Manage Multiple Sessions and Transactions

Using the commit statement to break a single long transaction into separate, shorter transactions is an important method for increasing concurrent user access to data. Another way to increase concurrency is to open separate database sessions for concurrent frames in an application. For more information about using multiple sessions, see How Programming for Multiple Sessions Works (see page 188).

In OpenROAD, transactions can span event blocks and frames but not database sessions. For example, assume a user begins a long transaction on a table in one frame and then needs to open a second frame to perform an update on a different table. The transaction in the first frame remains open while the user performs the update in the second frame.

If an application has multiple database sessions, each session has a separate transaction. Transactions in different database sessions can be open simultaneously.

Transactions continue across event blocks and frames until an explicit commit or rollback statement is issued in the same session in which the transaction was opened, or until the application is terminated.

For example, assume that an application has a frame used to update purchase orders while the data entry clerk is talking to the customer on the telephone. All line items are locked for the duration of the phone call. This frame also lets the clerk update information about the customer, such as changing a phone extension or contact.

If the frame opens two database sessions, one to update purchase order information and the other to update customer information, it is not necessary for the frame to hold locks on the customer record while the purchase order is updated.

Any database work performed after the frame switches to the second session either begins a new transaction associated with the second session or participates in an on-going transaction associated with that session. The transaction begun in the first session is still open, but there is no activity associated with it until the frame returns to that session.

Assume further that the frame calls another frame while the first frame is working in the second session. The work performed in the called frame is part of the transaction associated with the second session. If the called frame issues a commit or rollback statement, all the work of the transaction is committed or rolled back, respectively, including the transaction work that originated in the calling frame.
To close either transaction, either the main frame or the called frame must issue a commit or rollback statement while working in the session associated with the transaction. If the application is terminated, OpenROAD automatically commits both transactions.

How You Can Manage Transactions with Bitmaps, String Objects, Long Byte Objects, Long Vchar Objects, and Data Streams

Three methods are associated with the BitmapObject and StringObject system classes that interact directly with the database:
- InsertIntoDB
- UpdateInDB
- DeleteFromDB

Two methods are associated with the LongByteObject and LongVCharObject system classes that interact directly with the database:
- FetchFromDB
- UpdateInDB

Four methods are defined by the DataStream system class:
- Open
- SetCols
- Close
- NextRow

Three additional methods are defined for the QueryObject subclass:
- DBInsert
- DBUpdate
- DBDelete

If no transaction is open when one of these methods is executed, its execution initiates a transaction. If a transaction is open, then the work performed by the method becomes part of the transaction.

Like any other work performed by a database statement, the work is not committed to the database until the transaction of which it is a part is committed. If the program issues a rollback statement, any work performed by one of these methods in the affected transaction is also undone.
Similarly, if you set the DBHandle attribute of either a bitmap or StringObject object, you access the database. Setting the DBHandle can initiate a transaction, and commit and rollback statements affect the statements that set a DBHandle attribute.

For more information about setting a DBHandle directly, see How You Can Use the ErrorStatus Attribute (see page 201). For more information about detecting and handling DBMS errors in DataStream objects, see How You Can Handle DataStream Errors (see page 199).

**How You Can Manage Transaction Errors**

Deadlock and log file full errors cause the system to roll back the entire transaction. By default, other errors only roll back the statement that caused the error. If you want to roll back an entire transaction whenever any database error occurs, you must explicitly issue the rollback statement.

If you want the entire transaction rolled back in case of error, instead of writing code that handles each statement separately, you can set transaction management for the entire session using the SQL set session with on_error statement. In addition to requiring less code, this statement requires fewer writes to the log file.

For example, to ensure that every transaction in the Videos application is rolled back in the event of any database error, the application’s top frame (the main_control frame) includes the following statement:

```
set session with on_error = rollback transaction;
```

To turn off this option, causing rollback of only the current database statement, issue the following statement:

```
set session with on_error = rollback statement;
```

For more information about determining the status of the current transaction, see How You Can Use the Inquire_sql Statement (see page 201).

**How Running with Autocommit On Works**

`Autocommit on` describes a transaction management state in which each database statement is an individual transaction automatically committed when it is successfully executed. By default, autocommit is off. To turn autocommit on, use DBSessionObject's SetAutoCommit method.
When autocommit is on, the database session can have only one cursor open at a time (including a query object opened in cursor mode). When a cursor is open, the only valid database statements that the application can issue in the cursor’s database session are cursor statements. For example, if autocommit is on and the application is running a single database session, you cannot open a cursor in one frame, call or open another frame, and then perform another unrelated database operation in the new frame before returning to the first frame and continuing the cursor operation.

How You Can Use the Exit Statement with Transactions

The effect of the exit statement on an open transaction is undefined. Before exiting an application, your application should check for open transactions and provide some procedure for any that are found. Any open cursors are closed on exit.

How Programming for Multiple Sessions Works

Developing an application with multiple concurrent frames raises issues about transaction management. If operations in the frames are related, you might want to maintain a single transaction for the entire application. A single transaction forces a commit or rollback statement issued in any one of the open frames to affect all open transactions. Designing applications whose transactions do not span frames or event blocks is recommended, however, to improve concurrency.

However, maintaining one transaction throughout several concurrent frames also can cause locking conflicts. If operations in the frames are not related, you might want to open a different database session (each with its own transaction) for each frame.

In a single OpenROAD application, you can use multiple sessions to provide separate database sessions for concurrent transactions (with the same or different flag settings) in the same database or to connect each session to a different database.

Although you can open any number of sessions, you should limit the number of open sessions according to your transaction processing needs. If you need to access data from different databases, you need multiple connections to support this access. However, you should not open more sessions than you can easily manage because:

- You might lose track of them
- There is a DBMS limit to the number of concurrent sessions (exceeding the limit prevents other frames or applications from connecting)
When using database events, register them for every open session that is to receive them. Each frame receives database events only on its current connection. If you register a database event and then switch to another session that did not register the event, the frame will not receive the database event.

**How You Can Keep Track of the Current Session**

The DBSession attribute, defined for the ProcExec class and inherited by the GhostExec, FrameExec, and MethodExec classes, identifies the current database session for the application. This attribute contains the DBSessionObject object.

For more information about using this attribute, see How You Can Switch Database Sessions (see page 193).

**How You Can Access the DBSessionObject Object**

Each open database session in an application has an associated DBSessionObject object. The attributes of this object identify the database associated with the session, the session's ID, and the DBMS flags with which the session was opened. There are also DBSession Object attributes that provide status information about the session:

**Database**

Contains the name of the database accessed by the session

**DBMSError**

Contains the database-specific error number associated with the most recently executed SQL statement in the session

**ErrorNumber**

Contains the generic error numbers of any error resulting from the most recently executed SQL statement in the session

**RowCount**

Contains the number of rows affected by the most recently executed SQL statement in the session

**State**

Contains the connection state of the session (DS_CONNECTED or DS_DISCONNECTED)

For more information about the DBSessionObject's attributes, see the Language Reference Guide online help.
You can access the DBSessionObject object through the following ways:
- DBSession attribute of the FrameExec object—for running frames
- ProcExec object—for 4GL procedures
- MethodExec—for user class methods

For example, the following statement assigns the name of the current session's associated database to the variable dbname:

```java
dbname = CurFrame.DBSession.Database;
```

OpenROAD automatically assigns the DBSessionObject object for the application's starting session to the DBSession attribute of the application's starting component. Subsequent frames or procedures inherit the DBSessionObject from their calling component, unless you explicitly change sessions for that component.

For instructions to change sessions, see Connect Method (see page 191) and How You Can Switch Database Sessions (see page 193).

How You Can Use Concurrent Frames and Multiple Sessions

Each running frame in an OpenROAD application has its own FrameExec object. Because you can set the DBSession object for each FrameExec object, concurrent frames in the application can run in different database sessions.

For example, assume that an application has two concurrent frames running and that the DBSession attribute for Frame_A is set to first_session and for Frame_B to second_session. The variables first_session and second_session are two reference variables of type DBSessionObject.

When a user works in Frame_A, the session represented by first_session is the application's current session. When the user moves to Frame_B, the application's current session becomes the session represented by the variable second_session. Each time a user switches frames, OpenROAD switches the session in which the user is working.

When you use concurrent frames in this manner, remember that transactions and cursors do not cross sessions. In the example just mentioned, if you issue a commit statement in Frame_A, it has no effect on database statements executed in Frame_B's script.
How You Can Use Cursors, DataStream Objects, and Multiple Sessions

Every cursor or DataStream object opened in QY_CURSOR mode has an associated CursorObject object. One of the attributes of the CursorObject system class is DBSession. This attribute contains a DBSessionObject, which identifies the database session in which the cursor or DataStream object was opened.

After you open a CursorObject object, OpenROAD uses the session associated with the DBSessionObject object in the CursorObject’s DBSession attribute for any subsequent cursor statements.

Therefore, when you issue a fetch, update, delete, or close cursor statement, or a FetchRow, DBInsert, DBUpdate, DBDelete or Close method on a DataStream object, the specified operation takes place in the session in which the cursor was opened, regardless of the session in which the frame is running.

It is not necessary to switch explicitly to the session with which the CursorObject object is associated to perform a cursor operation. After the cursor statement completes, OpenROAD automatically returns to the session in which the frame was working before the cursor statement.

How You Can Open a New Database Session

Before you can open a session other than a frame’s starting session, you must create the DBSessionObject object for the new session by declaring a global or local reference variable of type DBSessionObject.

Declaring the variable globally lets you open the session represented by the variable in any frame of the application. If you declare a local variable, you can reference that variable in the frame in which the variable is declared or pass it to a frame or procedure called by that frame. You can open the session only within the frame in which you declared it or in subsequent frames or procedures called by that frame.

The DBSessionObject system class provides the following two methods for opening a new connection to a database:

- The Connect method
- The OpenNew Connection method

Connect Method

Use this method to start a session that is different from any of the currently open sessions, for example, a session with a different database or a different set of DBMS flags.
The syntax is:

```
integer = DBSessionObject.Connect(database = varchar(256), [flags = varchar(256)])
```

The DBSessionObject object must not be open already (the state must be DS_DISCONNECTED) when you invoke this method. You can specify any database, including one to which the application already is connected.

**flags**

Lets you define the runtime environment for the session. For example, you can open a session as another user by specifying the `-u` flag.

If the Connect method succeeds in opening the connection, it returns ER_OK. If it does not succeed, it returns an error code.

**OpenNew Connection Method**

This method opens a connection that is identical to a previously opened connection. Use this method to have two or more concurrent transactions against the same database with the same parameter settings.

The syntax is:

```
DBSessionObject = DBSessionObject.OpenNewConnection()
```

The DBSessionObject object on which the method is invoked must represent an open session, because this method uses the existing session as a template for the new session.

When the method completes, it returns a DBSessionObject object representing the new session. The new session accesses the same database and runs under the same flags as the existing session. If the method fails, it returns null.

For more information about these statements, see the Language Reference Guide online help.
Examples of Opening New Database Connections

The following examples show various ways to open a new database connection.

Using the OpenNewConnection method in the initialize block

The following initialize block locally declares the reference variable second_session for the DBSessionObject and uses the OpenNewConnection method to open a database session that is identical to the one for the calling frame:

```
initialize (second_session = DBSessionObject;) =
begin
  second_session = CurFrame.DBSession.OpenNewConnection();
... 
end;
```

Using the Connect method in the initialize block

The following initialize block uses the Connect method to open a connection to the videos database, specifying the user as marg and setting the arithmetic-handling mode of the session. The statement uses a globally-declared reference variable third_session for the DBSessionObject:

```
initialize (status = integer;) =
begin
  status = third_session.Connect(database = 'videos',
                                flags = '-umarg -xw');
end;
```

Using the OpenNewConnection method in a frame-invoking statement

The following openframe statement in a calling frame establishes a new, identical database session for the called frame (newframe):

```
newframe = openframe newframe with dbsession =
  CurFrame.DBSession.OpenNewConnection();
```

How You Can Switch Database Sessions

After you have opened additional database sessions for an application, you can switch between open sessions by setting the DBSession attribute. This attribute is available for both the FrameExec object and the ProcExec object, letting you switch sessions in a frame or in a 4GL procedure.

You can switch to an open session in any of the following locations:

- The initialize block of the called frame
- A frame-invoking statement (callframe, openframe, or gotoframe) in the calling frame or procedure
The following sub-sections describe how to switch databases in each of these situations.

How You Can Switch Sessions in a Called Frame

In addition to opening a new database session for a called frame in the initialize block of the called frame or the frame-invoking statement of the calling frame or procedure, you also can switch sessions for a called frame in either of these locations.

By default, a called frame inherits the value of the DBSession attribute from its calling frame or procedure. To switch sessions as you invoke the frame, you must change the value of this attribute in the frame-invoking statement. To switch sessions in the called frame, you use its initialize block.

The following examples demonstrate each of these methods:

Frame-invoking statement

You can use the with clause with the openframe, gotoframe, or callframe statement to place the application in a different open session. For example, the following statement opens the newframe frame and places the application in the session represented by the DBSessionObject third_session:

```plaintext
field1 = callframe newframe with dbsession = third_session;
```

Initialize block

You can use a statement in the called frame’s initialize block to place the application in a different open session. If the reference variable that represents the DBSessionObject for the target session is a global variable, you can simply assign that variable value to the DBSession attribute of the called frame, for example:

```plaintext
initialize() =
begin
  CurFrame.DBSession = second_session;
  ...
end;
```
If the reference variable was declared as a local variable in the calling
frame, you must pass the variable to the called frame as a parameter and
then assign the value to the DBSession attribute, for example:

/* in the calling frame */
field1 = callframe newframe
   (sessno = second_session);

/* in the called frame */
initialize (
   sessno = DBSessionObject;)
   begin
      CurFrame.DBSession = sessno;
      ...
   end;

If you use FrameExec's ParentFrame attribute to assign a called frame to a
parent other than the calling frame, the called frame has the same session
as the calling frame, not of the assigned parent. The following illustration
of DBSessions and ParentFrames shows this process:

How You Can Switch Sessions in an Event Block

To change to another session only for one operation in a specific event block,
set the DBSession attribute of the frame's FrameExec object in the event
block. For example, the following event block switches to another session if the
specified condition is true:

on add_customer =
begin
   /* If customer is rejoining the club, switch
      ** to session connected to the archived
      ** customer database. */
   if old_customer = TRUE
      CurFrame.DBsession = second_session;
   ...
   /* Switch back before exiting event block */
   CurFrame.DBsession = first_session;
end:
It is not necessary to switch back to the original session before the event block completes. Event block execution can begin in one session and end in another. However, if your frame has event blocks defined for any database events, the frame receives these events only when the application is running in the same session in which you registered the events for the frame. If you switch sessions in an event block to perform a task, be sure to switch back to the session in which the database events are expected.

For example, assume that session1 has registered for a database event and the current event block has changed sessions to session2. If a database event arrives while session2 is active, session1 does not receive the event.

For more information about database events, see Inter-Frame Communication Techniques (see page 333).

If you open a new database session for the current frame in an event block, you must explicitly switch the current frame to the newly opened session. The following example opens a connection to the videos database and switches to the session for the current frame:

on click chg_session_btn =
begin
  status = videos_session.Connect (database = 'videos');
  Curframe.DBsession = videos_session;
end;

How You Can Switch Sessions for a 4GL Procedure

To switch sessions from within a local 4GL procedure, you must set the CurProcedure.DBSession attribute for the ProcExec object. For example, the following statement switches the current procedure to the session represented by the DBSessionObject third_session:

CurProcedure.DBSession = third_session;

Note: In contrast to the CurProcedure.DBSession attribute, setting the CurFrame.DBSession attribute from within a local procedure will not change the local procedure's session. Instead, it changes the calling frame's session after the local procedure returns.
How You Can Switch Sessions for 3GL and Database Procedures

To use 4GL code to switch the session for a 3GL procedure, you must do so before you call the procedure. For example, the following code could appear in an event block:

```plaintext
on click add =
begin
    CurFrame.DBSession = second_session;
    status = callproc check_name
        (name = cus_name);
    CurFrame.DBSession = first_session;
end;
```

You can use Embedded SQL to switch sessions within a 3GL procedure. However, when the procedure completes and control returns to OpenROAD, the application automatically returns to the session in which it was running when the procedure was called.

You cannot switch sessions within a database procedure. If you want the tasks in a database procedure to be performed in a different session, you must change the session before you call the procedure.

How You Can Disconnect a Session

OpenROAD provides the Disconnect method for closing open connections. The syntax is:

```plaintext
integer = DBSessionObject.Disconnect()
```

**DBSessionObject**

Specifies the object that represents the session you want to close

OpenROAD also disconnects an open session if you remove all references to the session (set all references to the DBSessionObject for the session to null). In either case, closing a session closes any open cursors and query objects in that session and commits any transaction open in the session.

Handling Database Errors

This section describes the tools available in OpenROAD to detect and trap DBMS errors. Using these tools, you can develop an error-handling strategy appropriate for your application's needs. Several possible strategies are described, with an example of each.
In OpenROAD, you must check for DBMS errors after any database statement or method invocation that issues database statements. Use the tools described in the following lists to check for errors.

Two attributes defined for the DBSessionObject system class provide primary error-checking capability:
- ErrorNumber
- DBMSError

DataStream objects provide their own error-checking tools:
- ErrorNo attribute
- IsDBError attribute
- Status returned by methods that access the database

Ingres also provides the inquire_sql statement as an alternative tool.

Errors that originate in other portions of the application, for example, from a called procedure, are discussed in Errors Originating in Called 3GL Procedures (see page 151).

**How You Can Use the ErrorNumber and DBMSError Attributes**

Every database session has access to errors that occur during the execution of a database statement by accessing the values in the ErrorNumber and DBMSError attributes.

**ErrorNumber**
- Contains the generic error number associated with the error

**DBMSError**
- Contains the database-specific error number associated with the error

A generic error represents a class of error, and the database-specific number identifies a specific error associated with a specific DBMS within that class.

The values in ErrorNumber and DBMSError reflect the status of the last database statement that the application attempted to execute in the current session. If the statement was executed, each of these attributes is set to zero. If the statement did not execute because of an error, then these attributes contain the appropriate error numbers.

These attributes are also set when an error occurs setting the DBHandle attribute of a bitmap or string object.
As an example, the 4GL procedure dbms_error_message demonstrates one way to use these variables. This procedure displays an error message to the user when a DBMS error occurs:

```c
procedure dbms_error_message () =
begin
  if CurFrame.DBSession.ErrorNumber != 0 then
    return 'Error code is' +
      varchar(CurFrame.DBSession.ErrorNumber) +
      '. DBMS specific error number is' +
      varchar(CurFrame.DBSession.DBMSError) +
      '.';
  else
    return '';  
  endif;
end;
```

### How You Can Handle DataStream Errors

All DataStream methods that access the database return status information. These methods are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Objects Returning Status Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>SQLSelect and QueryObject objects</td>
</tr>
<tr>
<td>SetCols</td>
<td>SQLSelect objects</td>
</tr>
<tr>
<td>Close</td>
<td>SQLSelect and QueryObject objects (this method accesses the database only when the object was opened in QY_CURSOR or QY_DIRECT mode)</td>
</tr>
<tr>
<td>NextRow</td>
<td>SQLSelect and QueryObject objects (this method accesses the database only when the object was opened in QY_CURSOR or QY_DIRECT mode)</td>
</tr>
<tr>
<td>DBInsert</td>
<td>QueryObject objects</td>
</tr>
<tr>
<td>DBUpdate</td>
<td>QueryObject objects</td>
</tr>
<tr>
<td>DBDelete</td>
<td>QueryObject objects</td>
</tr>
</tbody>
</table>

Two methods are exceptions to the general descriptions of return status. One is the Close method, which does not return a status. The other is the NextRow method, which returns either ER_OK when successful, if no breaks have been set, or a number that represents the break level if breaks have been added to the DataStream. If the NextRow method is unsuccessful, it returns -1 and sets the ErrorNo attribute either to the error that was encountered, or 0 if there is no next row to retrieve.
If a database operation was successful, the value of the returned status and DataStream's ErrorNo attribute is ER_OK. If the operation was unsuccessful, the ErrorNo attribute contains the triggering error number. Because the ErrorNo attribute can be set by operations that fail in the client as well as the DBMS, use the value in the IsDBError attribute to determine whether the error occurred in the DBMS.

In addition to the ErrorNo and IsDBError attributes available to all DataStream objects, the QueryObject subclass provides the ZeroRowIsError parameter to the methods that access the DBMS (DBInsert, DBUpdate, and DBDelete).

By default, not finding any rows that satisfy an insert, update, or delete query (and therefore not changing the queried table) does not cause an error. If you want this situation to be considered an error, set the ZeroRowIsError parameter to TRUE.

For an example of error-checking while opening a DataStream object, see How You Can Run the SQLSelect Query (see page 170). For an example using the ZeroRowIsError parameter, see How You Can Update Data with a Query Object (see page 181).

**How You Can Use BitmapObject and StringObject Methods**

The BitmapObject and StringObject system class methods that access the database (InsertIntoDB, DeleteFromDB, and UpdateInDB) indicate errors in two ways:

- The return value is set to a nonzero error code if the method does not execute successfully.
- The DBSessionObject attributes ErrorNumber and DBMSError are set to the appropriate error number if a DBMS error occurs during the execution of the method.

**How You Can Use LongByteObject and LongVCharObject Methods**

The LongByteObject and LongVCharObject system class methods that access the database (FetchFromDB and UpdateInDB) indicate errors in the following ways:

- The return value is set to a nonzero error code if the method does not execute successfully.
- The DBSessionObject attributes ErrorNumber and DBMSError are set to the appropriate error number if a DBMS error occurs during the execution of the method.
How You Can Use the ErrorStatus Attribute

The ErrorStatus attribute of the BitmapObject and StringObject system classes is set to a nonzero value if an error occurs when you explicitly set the FileHandle or DBHandle attribute of either object. The following are ways to do this in a program:

- In an assignment statement
- In the target list of a statement, such as a select statement
- By passing the attribute by reference to a procedure

Indirectly setting either attribute by executing one of the methods associated with BitmapObjects or StringObjects (InsertIntoDB, UpdateInDB, DeleteFromDB) does not affect the ErrorStatus attribute. For example, assume that your application uses the InsertIntoDB method to add bitmap images to the database. This method constructs a DBHandle for each image that it inserts into the database. If an error occurs and the DBHandle is incorrectly set, the ErrorStatus attribute is not set; instead, an error message is generated, the ErrorNumber and DBMSError attributes of the DBSessionObject system class are set, and the DBHandle is set to null.

For more information about using these attributes, see How You Can Use the ErrorNumber and DBMSError Attributes (see page 198).

How You Can Use the Inquire_sql Statement

The inquire_sql statement returns runtime diagnostic and status information about database statements in an application. In OpenROAD, this statement can be most useful for retrieving error text or determining the transaction state. The syntax of this statement is:

```
inquire_sql (:variable_name = constant 
       . :variable_name = constant)
```

To retrieve error text, specify the errortext constant; to determine the transaction state, use the transaction constant. Errortext returns a varchar string and transaction returns an integer (TRUE or FALSE).
Handling Database Errors

For example, your error handling procedure might use inquire_sql to determine if a transaction is open:

```sql
procedure error_handler(
    ...
) =
begin
    ...
    inquire_sql (in_xact = transaction);
    if in_xact = FALSE then
        ...
end;
```

Choosing an Error Handling Strategy

This section discusses some common strategies for detecting and handling DBMS errors. These include:

- Using the DBMSErrorPrinting attribute
- Calling a general error-handling procedure
- Detecting errors when manipulating BitmapObject and StringObject objects

For a discussion of strategies for detecting and handling DBMS errors in DataStream objects, see How You Can Handle DataStream Errors (see page 199).

How You Can Use the DBMSErrorPrinting Attribute

Another method of handling errors is to call a general error-handling procedure each time an error is detected. For example, the script for the check_out frame in the Videos application calls an error handling procedure after each database statement to detect and handle any DBMS errors. The following code fragment from the check_out frame script calls the error handler:

```sql
/* Update video information */
update v_video
set last_checkout =
    :checkout_form.checkout[i].date_out,
    no_checkouts = no_checkouts + 1
where vid_no = :checkout_form.checkout[i].vid_no;

if error_handler(frm = CurFrame, commit_evt = 'Commit') != ER_OK then
    resume;
endif;
```
The if statement in the example calls the error-handling procedure. This procedure first checks to see if there was a DBMS error. If none occurred, it returns ER_OK to the calling program. If an error occurred and the error was not a deadlock error, then the routine displays an error message that includes the generic and local DBMS error number. If the error was a deadlock, the procedure sends an event back to the application to retry the DBMS statement.

The following code is the error handling procedure:

```java
/*
** Name: error_handler -- Detects and handles DBMS errors.
**
** Description:
** This procedure first checks if an error occurred in the previous DBMS statement. If so, it pops up an error message. If the error was deadlock or another error for which retry is appropriate, it sends an event back to the calling frame to retry the DBMS statement. No need to roll back the transaction in either case because the main_control frame set the application's database session to roll back automatically for all DBMS errors.
**
** Called By:
** check_out frame.
**
** Parameters:
** frm          Frame which called this procedure.
** commit_evt   Event sent to calling frame when there is a deadlock to retry DBMS statements.
**
** Returns:
** DBSession.ErrorNumber set by previous DBMS statement; ER_OK (0) if success */

procedure error_handler
(  frm = FrameExec,            /* Calling frame */
   commit_evt = varchar(32),   /* Commit event for retries if a deadlock occurred. */
) =

declare
  err_no = integer not null /* Error number */
enddeclare
```
begin
  err_no = CurProcedure.DBSession.ErrorNumber;
  if err_no = 49000 then
    /* We have encountered a deadlock or log
** full. */
    message 'Deadlock error occurred while updating the database.';
    frm.SendUserEvent(commit_evt);
  elseif err_no != ER_OK then
    message 'Error occurred while making changes in the database.' +
    dbms_error_message();
  endif;

  return err_no;
end:

Note: The error-handling procedure calls another procedure, dbms_error_message, when a non-deadlock DBMS error occurs. This called procedure returns the generic and local error numbers. For the code for the dbms_error_message procedure, see How You Can Use the ErrorNumber and DBMSError Attributes (see page 198).

How You Can Detect Errors When Working with Bitmaps, String, Long VChar, and Long Byte Objects

How you handle errors when you manipulate bitmaps, long byte, long varchar, or string objects depends on how you have stored these objects and what operations you are performing.

For example, if your bitmaps, long byte, long varchar, or string objects are stored in a file, whenever you use the WriteToFile method you can check the method’s return value to detect errors. The WriteToFile method returns 0 if it was successful or a 1 if not.

If your bitmaps and string objects are stored in the database, use the InsertIntoDB, UpdateInDB, and DeleteFromDB methods to manipulate these objects in the database. If your long byte and long varchar objects are stored in the database, use the FetchFromDB or UpdateInDB methods to manipulate these objects in the database. When you do, your error checking should check not only the return status of the method but also the value of DBSessionObject’s ErrorNumber attribute because these methods access the database, and it is possible for a DBMS error to occur.

Finally, you can set a FileHandle or DBHandle directly, for example, by an assignment statement. When you do so, you can check the ErrorStatus attribute for errors. For more information about this attribute, see How You Can Use the ErrorStatus Attribute (see page 201).
Chapter 7: Working with Arrays, Table Fields, and Collections

This section contains the following topics:

- Arrays and Table Fields—Conceptual Background (see page 205)
- Arrays (see page 207)
- Table Fields (see page 227)
- Table Field Operations (see page 236)
- Collections (see page 254)

The OpenROAD Workbench lets you create table fields dynamically by first creating and mapping arrays. This chapter describes the components of arrays and table fields and the syntax used to access these components. It also explains how to work with sets of data using dynamic arrays and table fields, and how to perform tasks that are commonly performed on arrays or table fields.

Arrays and Table Fields—Conceptual Background

An array is a named set of rows in which each row is a reference variable that points to an object of a given system or user class.

Arrays are arranged in rows and columns. The attributes of the underlying class determine the columns of the array. Each column is the collection of values in one specific attribute in all the rows of the array. The data in each column is of the same type (such as all names or all telephone numbers), while each row contains a single object composed of every column (such as one name and one telephone number).

The following diagram illustrates this concept:
Arrays and Table Fields—Conceptual Background

A table field is the means by which an OpenROAD application displays an array to users. A *table field* is a composite field comprised of active fields arranged in rows and columns. Each column in the table field maps to one attribute of the class associated with the array and each table field row displays one row of the array. The fields are usually entry fields, but can be of any kind, such as image or toggle fields. A table field can have columns for all or some of the class's attributes; however, it cannot have columns that are not array attributes.

By default, table fields appear on forms with table titles, column titles, control buttons, and scroll bars. For a detailed description of a table field, see Table Fields (see page 227).

In OpenROAD, you can use an array with or without an associated table field. A table field provides a convenient way to display the data in an array.

**Ways to Create an Array**

There are several ways to create an array in OpenROAD:

- **When you create global array variables**
  You create these variables using OpenROAD Workbench.

- **When you create local array variables**
  You create these variables with a declaration statement in a frame or field script.

- **When you create a table field in OpenROAD Workbench, an associated array variable is automatically created.**
  This default array has the same name as the table field and its attributes are identical to the columns in the table field. You can specify additional array attributes by specifying an underlying user class.

For more information about creating variables and table fields, see the *User Guide*. 
Manipulating Arrays and Table Fields

OpenROAD lets you add, delete, or change values in an array. You can also manipulate such aspects of a table field as:

- **Its visual characteristics**
  You can, for example, change the background color of the table field or any of its columns or cells. You can also modify any of the table field's trim.

- **The operations on the control button**
  You can delete or modify default operations or add your own. You can also remove the control button from display. You can dynamically enable or disable menu items.

- **The scroll bar**
  You can remove the scroll bar from display.

- **Individual cells or columns**
  You can highlight individual cells or columns or make a given column invisible.

How You Can Reference Table Fields

Throughout this chapter, references to `arrayname` and `tablefield_name` in the syntax are synonymous. Both refer to the array itself, because the table field and its associated array (ArrayObject) are identically named.

To refer to the TableField object, use the field function with the table field's name. The syntax is:

```
field(tablefield_name)
```

For more information about the field function, see How You Can Reference Objects (see page 114).

Arrays

You can refer to an array as a whole or to an individual row or cell. (Individual columns are referenced only through the table field associated with the array.)

For the syntax to reference each part of an array, see How You Can Reference an Array (see page 210).
Because an individual attribute of an array object can itself be an array variable or another reference variable, you can nest arrays and objects inside arrays. The following illustration shows an array containing a nested object: a reference variable with two attributes, represented by the fields Vendor# and Vname on the form. The array that contains this object has four attributes, represented by the headers Vendor_subform, Part#, Catalog#, and Price.

The following graphic illustrates the array as it would appear in a table field:

<table>
<thead>
<tr>
<th>Vendor_subform</th>
<th>Part#</th>
<th>Catalog#</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vendor#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vname</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vendor#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vname</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following graphic illustrates a nested array. The outermost array has two attributes, shown by the headings Vendorno Col and Pricetbl. The attribute that appears in the table under the heading Pricetbl is an array of three attributes, Partno_col, Catno_col, and Price_col.

How You Can Declare an Array

If the array variable is not associated with a table field, you must declare it in your application before you can reference the variable (or its component parts) in the 4GL code. The syntax for declaring an array variable is:

```
arrayname = array of classname [default null]
```

The class can be any system class or user class. The following statement declares an array of user class VIDEO_ROW. The video_list reference variable defaults to null:

```
video_list = array of VIDEO_ROW default null;
```

If the array's class is not known until runtime, declare the type of the array to be Object. For an example of using an array generically, see How You Can Sort Table Field Data Generically (see page 252). For more information about specifying the default null clause, see How You Can Reference Objects (see page 114).
Declaring an array variable does not place any actual rows into the array; instead, it:

- Specifies the type of object that is to comprise the array
- Initializes an object of type ArrayObject that contains a description of the array and provides access to methods for manipulating the array (if it is not declared with default null)

For more information about these methods, see How You Can Manipulate Arrays (see page 212).

**How You Can Reference an Array**

OpenROAD provides syntax for referencing each of the following array components:

- An array as a single entity
- Individual array rows
- A member of a nested structure
- Individual columnfields if the array is associated with a table field of the same name
- Individual data items in a row

The syntax for each of these references is summarized in the following table:

<table>
<thead>
<tr>
<th>Array Component</th>
<th>Reference Syntax and Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td><code>arrayname</code></td>
</tr>
<tr>
<td></td>
<td>Example: <code>callproc update_db(vid_cust=custtable);</code></td>
</tr>
<tr>
<td>Row</td>
<td><code>arrayname[n]</code></td>
</tr>
<tr>
<td></td>
<td>Example: <code>callproc update_acct(account = custtable[2]);</code></td>
</tr>
<tr>
<td>Nested Object's Attribute</td>
<td><code>arrayname[n].nestedobject.attribute</code></td>
</tr>
<tr>
<td></td>
<td>Example: <code>custtable[5].address.custcity = 'Daly City';</code></td>
</tr>
<tr>
<td>Nested Array's Attribute</td>
<td><code>arrayname[n].nestedarray[m].attribute</code></td>
</tr>
<tr>
<td></td>
<td>Example: <code>custtable[4].vid_rent[5].title = Kismet';</code></td>
</tr>
</tbody>
</table>
Arrays

Working with Arrays, Table Fields, and Collections  211

<table>
<thead>
<tr>
<th>Array Component</th>
<th>Reference Syntax and Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column field</td>
<td>arrayname[*].attribute</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>on childexit custtable[*].acctbalance</td>
</tr>
<tr>
<td></td>
<td>This syntax is usable only if custtable is a table field.</td>
</tr>
<tr>
<td>Individual attribute</td>
<td>arrayname[n].attribute</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>custtable[3].custzip = zip;</td>
</tr>
<tr>
<td></td>
<td>This syntax refers to an individual attribute. If we had tablefield[n]. attribute, it would refer to an individual table field cell.</td>
</tr>
</tbody>
</table>

To refer to elements of an array underlying a dynamically created table field, use a dynamic expression (DynExpr). For more information about creating a table field dynamically, see Creating Dynamic Frames (see page 363).

References to nested arrays and objects proceed left to right from the general (outermost array row) to the specific (innermost array, row, or data item). For example, the order of precedence for the following reference:

custtable[4].vid_rent[5].title

is:
1. Row 4 in the custtable array
2. The attribute vid_rent (itself an array) in row 4
3. Row 5 of the vid_rent array
4. The attribute title (of the vid_rent array)

You cannot change the order of precedence in nested syntax.

Restrictions on Array Structure

Because each row in an array points to an object of a given system or user class, an array cannot consist of a single attribute whose data type is simple, such as an integer or varchar. To create a single-column array that contains integers, for example, you must use the IntegerObject system class. To access the value of the IntegerObject, use its Value attribute.
The following statement puts the value of the third row of a single-column array of IntegerObjects into an integer variable named intvar:

```
intvar = intarray[3].intcol.Value;
```

To create a single-column array that contains varchars, point to the StringObject system class.

For more information about the Value attribute of these object types, see the Language Reference Guide online help.

### How You Can Manipulate Arrays

An application can perform the following array operations:

- Add rows
- Change the values in all or some of the rows
- Delete all or some of the rows in an array

You can choose from a variety of ways to accomplish these tasks:

- For a discussion of adding rows, see How You Can Add Rows to an Array (see page 216).
- For a discussion of changing rows, see How You Can Change Values in an Array (see page 219).
- For more information about deleting rows, see How You Can Delete Rows in an Array (see page 221).

There are several methods of the ArrayObject class that manipulate arrays (such as inserting, removing, or sorting rows) or provide information (such as the number of rows in the array). For a detailed description of these methods, see How You Can Retrieve Array Information (see page 221), or see the ArrayObject section of the Language Reference Guide online help.

### How You Can Use the _RowState Attribute

To determine changes made directly to an array by a user editing a table field, you can use the _RowState attribute for each row in an array. This attribute contains information about the state of each row, indicating the last action performed on the row by the user or by the SetRowDeleted method. Changes made to an existing row by a 4GL script (except any changes resulting from the execution of the SetRowDeleted method) do not affect this attribute.
There are five valid row states:

**RS_UNDEFINED**

Indicates that:

- A row was automatically added to the end of an array when a user, in a table field, set the input focus to the row just beyond the last row or used the down arrow key to move past the last row.
- A user selected the Insert Before Current Row operation from a table field's default control button. A row whose state is RS_UNDEFINED cannot change to a state of RS_DELETED. When these rows are deleted by any means, they disappear from the array.

**RS_NEW**

Occurs when a user types in or otherwise changes an undefined (RS_UNDEFINED) row.

A row whose state is RS_NEW cannot change to a state of RS_DELETED. When these rows are deleted by any means, they disappear from the array.

**RS_UNCHANGED**

Indicates a row that was added to the array by a 4GL script (by the InsertRow method or assignment statement) and has not yet been changed by the user.

**RS_CHANGED**

Occurs when a user makes a change, other than a deletion, to an RS_UNCHANGED row. If the user changes the data and then sets it back to its original value, the row's state remains RS_CHANGED.

**RS_DELETED**

Indicates a row that was initially RS_UNCHANGED or RS_CHANGED and then was marked deleted, either by the program (using the SetRowDeleted method) or by a user (using the table field control button's Delete Current Row or Delete All Rows operation).

A row that has the state RS_DELETED remains in the array with a non-positive row number.

You can force a row to reflect a different state than would occur by default by directly assigning a value to the _RowState attribute. For example, assume a frame remains open after the user clicks the Save button. This frame lets the user continue making changes to values in a table field after saving existing changes. To prevent the same changes from being written to the database the next time the user clicks the Save button, you can set the state of all rows back to RS_UNCHANGED. The following code fragment checks the row state of every row in the custtable array, makes appropriate changes to the database, and resets the value of all row states to RS_CHANGED:
for i = custtable.FirstRow to custtable.LastRow do
  if custtable[i]._rowstate = RS_DELETED then
    repeated delete from customer where acctno = :cust_table[i].acctno:
  elseif custtable[i]._rowstate = RS_NEW then
    repeated insert into customer ...
  elseif custtable[i]._rowstate = RS_CHANGED then
    repeated update customer...  
  endif;
  commit;
  /*Reset rowstate to prevent handling this row
   ** again on next Save unless data changes
   ** again. */
  custtable[i]._RowState = RS_UNCHANGED;
endfor;

Fence Diagram for Row States and Transitions

The following illustration summarizes the effects of programming statements and user actions on array row states:

- The vertical lines represent the starting and ending row states.
- The arrows are labeled to indicate the statement or action that causes the state to change.
- The numbers in parentheses to the right of each row state indicate the numeric value of the row state constants.

The following statements illustrate a few possibilities:

- If a user opens a new row by selecting the Insert New Row operation of a default control button, the row state changes from nonexistent to RS_UNDEFINED.
  - If the user changes the newly inserted row, the row state changes from RS_UNDEFINED to RS_NEW.
  - If the 4GL code makes an assignment to the newly inserted row, the row state does not change; therefore the arrow in the fence diagram begins and ends at the same position.
- If a row with a row state of RS_NEW is deleted, its row state becomes nonexistent.
- If a row with a row state of RS_UNCHANGED is deleted, its row state becomes RS_DELETED.
Arrays
How You Can Add Rows to an Array

Because creating an array does not populate it, adding rows to the array may be the first task to perform after declaring the array. There are several ways that you can populate an empty array or add additional rows to a populated array:

- Add rows by first assignment
- Use the InsertRow method
- Assign the contents of one array to another
- Create a populated array by duplicating an existing array
- Allow the user to append to a table field

The following sections describe each of these methods in more detail.

Add Rows by First Assignment

When you make an assignment to a nonexistent row in your 4GL code, OpenROAD creates that row if it is the next row in the array sequence; if it is not, an error occurs. This is an easy way to fill an array.

For example, the following select loop fills the custacct array:

```
i = 1;
select :custacct[i].acctno = cacctno,
    :custacct[i].custname = cname,
    :custacct[i].acctbalance = acctbal
from customer
begin
    i = i + 1;
end;
```

These statements perform the following operations:

1. Place the first row that the select returns into the first row of the array.
2. For each row retrieved, increment i (the index into the array) by one.
3. Place the second returned row into the second row in the array (in the second iteration of the loop) and again increment i.
4. Continue this process for each row returned by the select statement.

If any of the array’s rows were nonexistent prior to executing the select statement, they are automatically created when the statement is executed.
This method only allows rows to be added to the end of an array or to an empty array. Also, the index must be incremented by one; this method does not work if row numbers are skipped.

When you use this method to add rows to an array that is displayed in a table field, OpenROAD updates the display when the event block containing the select loop completes.

**Use the InsertRow Method of the ArrayObject Class**

The syntax for this method is:

```
integer = arrayname.InsertRow([rownumber = integer]
                           [, rowobject = object] [,_rowstate = integer]);
```

- `arrayname` specifies the name of the array
- `object` specifies a reference variable that points to an object that is either of the same class as, or is a subclass of, the array's class. When the statement is executed, OpenROAD inserts the object into the array at the specified position.

  If you do not specify an object, OpenROAD constructs an object whose attributes contain default values and inserts the object into the array. Default values are null for nullable fields, zero for numeric fields and blanks for character fields.

  Another way to set default values for a column is by setting default values for the ProtoField of the column. For more information, see TableField, ColumnField, and ProtoField Objects (see page 228).

The InsertRow method inserts the new row ahead of the row specified by the rownumber parameter and adjusts the numbers of the rows following the new row. For example, if you insert a row into the third row position in the array, the current row 3 becomes row 4. If you do not specify the rownumber parameter, OpenROAD inserts the new row as row 1. To insert a row at the end of the array, set rownumber = `arrayname.LastRow` + 1.

By setting _rowstate, you can set the _rowstate to other than the default RS_CHANGED.

Inserting a default object is the best way to insert an empty row into an array. For example, the following code inserts a blank row at the specified position:

```
custtable.InsertRow(rownumber = row);
```
The inserted row is blank because the rowobject parameter is not specified. This technique is especially useful if you display an array in a table field and want the user to enter new data in a particular position in the array.

If you set rowobject to null, 4GL creates the row number in the array, but no object is associated with the new row. This feature provides useful program-control capabilities but should not be used if the array is displayed by a table field.

**Assign the Contents of One Array to Another Array**

Assume that you have an array variable, custtable, that contains customer addresses and an empty array variable, addresses. The following statement assigns the contents of custtable to addresses:

```plaintext
addresses = custtable;
```

This statement does not create two distinct objects, but rather two array variables that point to the same set of objects, that is, to the same array. Both custtable and addresses now reference the same array.

**Create a Populated Array by Duplicating an Existing Array**

If you want two separate array variables so that you can modify the data in one array without affecting the other, you can create a second copy of the array by using the Duplicate method defined for the Object system class.

You can make an exact duplicate of an array, placing a reference to the new array in a new reference variable, provided the new reference variable is of the same class, or a superclass of, the array being copied.

The following example creates a new array object called array2 which is an exact duplicate of array1:

```plaintext
declare
    array1 = array of myclass;
    array2 = array of myclass;
enddeclare
begin
    /* load values into array1 */
    /* then duplicate array1 */
    array2 = array1.Duplicate();
    ...
```

**Allow the User to Append to a Table Field**

A table field may or may not allow the user to add new rows at the bottom of the array. For more information about this, see How You Can Append Rows to End of Table Field (see page 243).
How You Can Change Values in an Array

If an array is not displayed in a table field, the only way to change one of its values is by assignment. For example, the following statement updates the account balance of the customer in the fifth row:

```plaintext
custtable[5].acctbalance = custtable[5].acctbalance + credit - debit;
```

If your array's data type is a dynamically created user class, you must use a DynExpr to change values programmatically. For more information about changing values in dynamically created arrays, see Creating Dynamic Frames (see page 363).

If an array is displayed in a table field whose bias allows data modification, the user can edit and change the displayed data. For a description of the biases for table fields, see Creating Dynamic Frames (see page 363).

If two or more array variables point to the same data set and you change a value in the data set, the change is reflected in all the array variables that point to that data set. If any of the array variables is displayed in a table field, refresh the display to make the change visible if you want it to appear prior to the end of the event block. To refresh the display, use the UpdField method defined for the ActiveField system class.

The following statement updates display of the custtable table field:

```plaintext
field(custtable).UpdField();
```

How You Can Specify an Array Row to be Deleted

The SetRowDeleted method sets the _RowState attribute of a specified row to RS_DELETED. Marking a row deleted is useful to allow the application to access the data in the deleted row before the data is lost.

For example, assume a user edits the data displayed in a table field, changing data in some rows and deleting a few rows. When finished, the user clicks the Save button. The event block associated with this button checks the _RowState of each row in table field's array and, based on the row state, performs the appropriate operation in the database.
The following code fragment illustrates such row checking:

```plaintext
on click menu.save_menu =
begin
  i = custtable.FirstRow;
  while i <= custtable.LastRow do
    if custtable[i]._rowstate = RS_DELETED then
      repeated delete from customer
      where acctno = :cust_table[i].acctno;
    elseif custtable[i]._rowstate = RS_NEW then
      repeated insert into customer ...
    elseif custtable[i]._rowstate = RS_CHANGED then repeated update
      customer...
    endif;
    i = i + 1;
  endwhile;
commit;
end;
```

**Note:** If the user had actually removed rows from the array, rather than marking them as deleted, the application would not be able to apply those deletions to the database.

The syntax for the SetRowDeleted method is:

```plaintext
integer = ArrayObject.SetRowDeleted(rownumber = integer)
```

**rownumber = integer**

(Required.) Identifies the row to be deleted

The effect of marking a row as deleted depends on the initial state of the row:

- **Original row state of RS_CHANGED or RS_UNCHANGED**
  Marking the row as deleted gives the row a non-positive sequence number in the array but does not remove it from the array. The first row deleted becomes row 0, the next row deleted becomes -1, and so on. If, for example, 10 rows were marked RS_DELETED, the FirstRow method would return -9.

  After a row is marked as deleted, OpenROAD moves the rows that follow the deleted row down in the array's row sequence. For example, if you mark row 3 deleted, then the current row 4 becomes row 3.

- **Original row state of RS_NEW or RS_UNDEFINED**
  Marking the row as deleted actually removes the row from the array.

In all cases, if the array is displayed in a table field, the row is removed from the table field display.
How You Can Delete Rows in an Array

When you want to delete rows from an array and do not need to keep track of the deleted rows, use either of the following methods:

- The RemoveRow method to delete rows one row at a time

  The syntax for the RemoveRow method is:

  \[
  \text{integer} = \text{ArrayObject.RemoveRow(rownumber} = \text{integer})
  \]

  You must include the rownumber parameter to identify the row to be deleted.

  If the row is deleted successfully, the method returns ER_OK. If the row is not found, the method returns ER_ROWNOTFOUND.

  After the specified row is deleted, OpenROAD adjusts the numbers of the rows that followed the deleted row. For example, if you remove row number 5, then the current row 6 becomes the new row number 5.

- The Clear method to delete all the rows at once

  The syntax for the Clear method is:

  \[
  \text{integer} = \text{ArrayObject.Clear()}
  \]

  Both of these methods actually delete rows from the array (and any associated table field display), rather than just marking the rows deleted. Rows that have a _RowState value of RS_DELETED as well as those with positive sequence numbers are deleted.

How You Can Retrieve Array Information

It is useful for the control flow of a program to know how many rows are in an array or whether or not all rows have been processed. The ArrayObject class has three attributes that contain quantitative information about an array. Each of these attributes is an integer value that provides some information about the specified array.
These attributes are:

**AllRows**

Returns the total number of rows in the array, including rows marked deleted. A zero value indicates that there are no rows, not even any marked deleted.

**FirstRow**

Returns the number of the first row in the array, where first is defined as the row with the lowest number. This attribute is useful for determining whether any rows in the array have been marked deleted.

If this attribute returns zero or a negative number, it indicates that there are rows marked deleted in the array, because rows with a _RowState of RS_DELETED are given non-positive numbers, starting with zero.

For example, assume that an array has six rows, two of which are marked deleted. The array's row numbers are -1, 0, 1, 2, 3, and 4. If you use the FirstRow attribute against this array, it returns the value -1.

If there are no rows having a row state of RS_DELETED, then FirstRow returns 1.

**LastRow**

Returns the highest non-negative sequence number in the array. (Rows with positive sequence numbers are rows that are not marked deleted.) For example, the value returned by the LastRow attribute on the previously mentioned array is 4. If the value returned by LastRow is zero, there are no undeleted rows in the array, although there may be deleted rows.

These attributes are most often used to control loops in an application, for example:

```plaintext
on click menu.save_menu =
begin
  i = custtable.FirstRow;
  while i <= custtable.LastRow do
    if custtable[i]._rowstate = RS_DELETED then
      ...
    endif;
    i = i +1;
  endwhile;
  commit;
end:
```

You must be careful if you delete rows in a for loop. The following code is correct:

```plaintext
for i = 1 to array.LastRow do
  if (condition) then
    array.RemoveRow(rownumber = i);
  endif;
endfor;
```
However, the following code is not correct:

```csharp
n = array.LastRow;
for i = 1 to n do
    if (condition) then
        array.RemoveRow(rownumber = i);
    endif;
endfor;
```

In the preceding example, if a row is deleted, there will no longer be \( n \) rows in the array. You can remedy this problem by using the downto form of the for statement.

```csharp
n = array.LastRow;
for i = n downto 1 do
    if (condition) then
        array.RemoveRow(rownumber = i);
    endif;
endfor;
```

### How Sorting Arrays Works

The Sort method sorts the array according to values in specified columns. The syntax for this method is:

```csharp
status = arrayobj.sort(attname = direction {, attname = direction})
```

Any attribute in the array with a simple data type or one of the allowable object types can be used as a sort key. Sort keys are listed in the order of significance, with the primary sort key listed first. Sort direction is specified as AS_ASC or AS_DESC.

The value of status is either of the following:

- **ER_OK**
  - Specifies that the sort succeeded
- **ER_FAIL**
  - Specifies that the sort failed. Failure is caused by such problems as specifying an incorrect attribute or sort order or attempting to sort non-sortable data.

The following example sorts the vlist array in descending order, using the category attribute as the primary sort key and title as the secondary sort key:

```csharp
vlist.sort(category = AS_DESC, title = AS_DESC);
```
Attributes used in a sort can be of any simple data type, such as varchar or integer. The Sort method uses the object's Value attribute if the attribute is an object of one of the following types:

- DateObject
- FloatObject
- IntegerObject
- MoneyObject
- StringObject

No other object types are permitted as sort attributes.

If the array is displayed in a table field, the table field is automatically updated after the sort. For an example of sorting a table field, see How You Can Sort Table Field Data Generically (see page 252).

How You Can Find Values in an Array

The Find method can locate an element of the array based on the contents of a single attribute of the array's class. The syntax for this method is:

```plaintext
status = ArrayObject.find(attributename = string,
                          value = expression | searchstring=string
                          [.format = string] [.casesensitive = integer]
                          [.rownumber = byref (integer_variable),
                          [.usedeleted = integer] [.startrow =integer]
                          [.endrow=integer])
```

**attributename**

Specifies the attribute being searched for. Attributes can be of any simple data type, such as varchar or integer. The Find method inspects the object's Value attribute if the attribute is an object of one of the following types:

- DateObject
- FloatObject
- IntegerObject
- MoneyObject
- StringObject
Other object types are permissible for this parameter, but only for a search by exact value. When you use the Find method to locate a value in an array that has no named attributes, the attributename should be set to 'value'. For an example of using the Find method in conjunction with a pop-up frame that lets the user specify the search conditions, see How You Can Scroll Indirectly with the Find Method (see page 240).

For more information about the Value attribute of these object types, see the *Language Reference Guide* online help.

There are two ways of finding specific rows in an array:

- **By value**, using `value = expression`
- **By searchstring**, using `searchstring = expression`

When you specify an expression whose type is compatible with the specified attribute, the Find method searches for an exact match between this expression and the specified attribute.

Numeric values always fall into this category.

By searchstring, using `searchstring = expression`

When you specify a string expression (and optional format), the Find method searches for a match between the string and specified attribute, converting the attribute's value to a string using the specified (or a default) format.

The searchstring parameter lets you use any of three wildcard characters:

- `%` (percent sign)—represents a character string, possibly a null string
- `_` (underscore)—represents exactly one character
- `\` (backslash)—represents the escape character

The following table provides an example of string matching:

<table>
<thead>
<tr>
<th>Actual String</th>
<th>Search Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>%abc</td>
</tr>
<tr>
<td>Abc</td>
<td>matches</td>
</tr>
<tr>
<td>1abc2</td>
<td>matches</td>
</tr>
<tr>
<td>%abc</td>
<td>matches</td>
</tr>
<tr>
<td>Aabc</td>
<td>matches</td>
</tr>
</tbody>
</table>
The format parameter can be any of the legal OpenROAD format templates for the attribute type for which you are searching. For a list of the format templates, see the User Guide. For more information about specifying the format parameter, see the Language Reference Guide online help. If the format attribute is omitted, the default format is used for the specified attribute type.

If the Find method finds a row, the rownumber parameter (if specified) is set to the row number of the found row. Table fields displaying the array are not automatically scrolled to the row found. To move the input focus to this row, you must write your own 4GL code, using SetInput Focus method on the table field.

The other parameters for the Find method are:

**casesensitive**
- Specifies whether case is significant
- **Default:** TRUE

**usedeleted**
- Specifies whether to search through deleted rows
- **Default:** FALSE

**startrow**
- Always specifies a positive number unless usedeleted is TRUE, in which case startrow defaults to array.FirstRow
- **Default:** 1

**endrow**
- Always specifies a positive number unless usedeleted is TRUE. If endrow is less than startrow, the search wraps
- **Default:** array.LastRow

The value of status is one of the following:

**ER_OK**
- Specifies that the row was found

**ER_ROWNOTFOUND**
- Specifies that the row was not found

**ER_FAIL**
- Specifies that the search failed. Failure is caused problems such as attempting to compare incompatible data types or specifying incorrect attributes.
Table Fields

A table field is a composite field comprised of active fields arranged in rows and columns. Each column in the table field maps to one attribute of the class associated with the array and each table field row displays one row of the array. The fields are usually entry fields, but can be of any kind, such as image or toggle fields.

A column is a stack of identical active fields: each has the same data type and contains the same type of information, for example, customer names or telephone numbers. A row contains one field from each column in the table.

The following graphic illustrates these concepts:

A table field can be oriented in either of two ways:

- Vertically
  A vertical table field displays its columns vertically and its rows horizontally.

- Horizontally
  A horizontal table field displays its columns horizontally and its rows vertically.
Table Fields and Associated Arrays

There are three ways to create a table field in OpenROAD Workbench:

- You can create an activefield for each column, select them all, and then group them into a Tablefield.
- You can Create Fields from Database Table, and select the table field option.
- You can Create Fields from User Class, and select the table field option.

If you do not create the table field from a UserClass, OpenROAD provides a default unnamed user class for the underlying array. If you created the table field using a Group, the attribute names correspond to the names of the activefields that were grouped and have data types corresponding to the activefield's data types. If you created the table field from a database table, the names and values of the attributes correspond to the column names and data types of the table.

The data type of the array variable associated with the table field is the UserClass associated with it, either a named one, or the default unnamed one. The correspondence between array attributes and table field columns facilitates selection of multiple rows from a database table into the field display.

TableField, ColumnField, and ProtoField Objects

When you create a table field, OpenROAD initializes a TableField object for the table field and one ColumnField object for each column in the table field. The TableField object provides access to the following table field components:

- Visual, such as the width of the outline around the table field
- Operational, such as setting the input focus to a specific cell
- Textual, such as the text of its title

The ColumnField objects give access to the following column field components:

- Visual, such as the background color of its prototype field, ProtoField
- Textual, such as the text of its title
Just as each table field and each column in the table field has an associated object, each cell in a table field also has an associated CellAttribute object. However, because of their negative impact on performance, CellAttributes are by default turned off when a table field is created. CellAttributes can be turned on explicitly with the HasCellAttributes property. Or they can be turned on implicitly by changing a single cell’s attribute (see Individual Cells (see page 233)). They can be freed up by turning off HasCellAttributes, but this resets all attributes back to the protofield’s attributes.

For more information about setting CellAttribute objects on, see How You Can Turn CellAttributes On (see page 246). For more information about TableField, ColumnField, and CellAttribute objects, see the Language Reference Guide online help.

**Table Field Components**

A table field is made up of many individual parts, such as its scroll bar, its title, its columns, and its outlines. OpenROAD lets you change many of the characteristics of these parts so that you can customize a table field as needed.

This section describes the parts of a table field, tells you what kinds of changes you can make to each part, and shows you the correct 4GL syntax to reference each part.

In most cases, you must use the field function in your 4GL code to access the properties of a table field (created in OpenROAD Workbench) or its parts. Using the field function accesses the reference variable that points to the TableField object. Referring to the table field without using the field function refers to the array displayed by the table field. For a full description of the field function, see Working with Classes (see page 113).
The following application window shows the parts of a table field. These parts are described in the sections that follow.

**Table Field Background**

The table field background represents the space on the form occupied by the table field. You can change the color and pattern of this space or the width, color, and pattern of its bounding line. To access these attributes, use the following syntax:

```
field(tablefield).attribute = value
```

*attribute*

Specifies any attribute of the TableField object. For example, the following code changes the background of the specified table field to light blue:

```
field(custtable).bgcolor = CC_PALE_BLUE;
```

**Table Title**

You can change the table title's text and appearance, for example, its typeface or the color of its background.

You access the title's text with the Title attribute of the TableField object. This attribute has a data type of varchar(256). The syntax is:

```
field(tablefield).Title = title_string
```

*title_string*

Specifies a quoted literal or a varchar variable
For example, the following statement changes the title of the custtable table field:

```
field(custtable).title = 'Customer Information';
```

To change the visual properties of the title, use the TitleTrim attribute, which has a data type of FreeTrim object. The syntax is:

```
field(tablefield).TitleTrim.attribute = value
```

**attribute**

Specifies an attribute of the FreeTrim object

For example, the following statement sets the background color of the table's title:

```
field(custtable).titletrim.bgcolor = CC_PALE_BLUE;
```

### Table Header

The table header displays the column titles of the table field. OpenROAD lets you display or remove this header and change its appearance and boundary line.

To access the visual characteristics of the table header, use the TableHeader attribute of the TableField object. The data type of this attribute is StackField. The syntax is:

```
field(tablefield).TableHeader.attribute = value
```

**attribute**

Specifies an attribute of the StackField object. For example, the following statement sets the pattern of the table header's bounding line:

```
field(custtable).tableheader.outlinecolor = CC_PALE_BLUE;
```

For more information about working with column headers, see How You Can Display and Remove Column Headers (see page 245).

### Column Titles

You can change a column's title text or its visual appearance. To change the text, use the following syntax:

```
field(tablefield[*].column).Title = title_string
```
This syntax references the Title attribute of the column's ColumnField object.

**Title = title_string**

*Data type:* varchar(256)

Specifies a title, which can be a quoted literal or a varchar variable.

Use the TitleTrim attribute of the ColumnField object to access the visual attributes of the title. TitleTrim is a FreeTrim object. The syntax is:

```plaintext
field(tablefield[*].column).TitleTrim.attribute = value
```

**attribute**

Specifies an attribute of the FreeTrim object. For example, the following statement changes the title of the custname column to boldface:

```plaintext
field(custtable[*].custname).TitleTrim.IsBold = TRUE;
```

**Column Field**

You can change a variety of a column's visual characteristics. To access these characteristics, use the following syntax:

```plaintext
field(tablefield[*].column).attribute = value
```

**attribute**

Specifies an attribute of the ColumnField object. For example, the following statement sets the background pattern of the custname column in the custtable table field:

```plaintext
field(custtable[*].custname).bgpattern=FP_SHADE;
```

**Prototype Field**

The prototype field is not an actual physical area on a table field, but rather a template for all the fields in the column. You can access this field in the Frame Editor by choosing the first row of the table field. You can access this template in 4GL through the ProtoField attribute of the ColumnField object. The ProtoField attribute's data type is FormField.
The FormField object is high in the system class hierarchy. Most of its useful attributes are associated with actual field objects, such as entry field or list field, that are subclasses of the FormField object. Therefore, you must cast references to the ProtoField attribute to the appropriate field type. The syntax is:

\[
\text{fieldtype(field(tablefield[*].column).ProtoField).attribute} = \text{value}
\]

**fieldtype**

Specifies the class of the column's fields.

**attribute**

Specifies any appropriate attribute of that class.

For more information about casting, see How You Can Work with Attributes (see page 140).

When you change a ProtoField attribute, the change appears in all the cells in the column. The following statement causes all of the values in the custname column (a column of entry fields in the table field custtable) to appear in boldface:

\[
\text{entryfield(field(custtable[*].custname).protofield)}
\]

\[
\text{isbold} = \text{TRUE};
\]

You can hide a table field column by setting the bias of its protofield to FB_INVISIBLE:

\[
\text{entryfield(field(tf[*].name).protofield).CurBias} = \text{FB_INVISIBLE};
\]

**Individual Cells**

You can access some of the visual characteristics of individual cells of an unnested table field by using the CellAttribute object. This object has a variety of attributes that control several of the visual characteristics of a cell. Note that because the size of a cell is always identical to the size of all cells in the same column, you cannot change any attributes that would affect the size of a CellAttribute attribute.

The syntax for changing individual cell attributes is:

\[
\text{field(tablefield[n].column).attribute} = \text{value}
\]

**tablefield[n].column.attribute**

Identifies a specific cell in row \(n\) of the array for the table field, and \(attribute\) is an attribute of the CellAttribute object.
For example, the following statement changes the background color of the fourth cell in the name column of the custtable table:

```java
field(custtable[4].name).bgcolor = CC_LIGHT_BLUE;
```

Because of the significant overhead entailed in their use, CellAttribute objects are not automatically created when a table field is created. CellAttributes can be turned on explicitly with the HasCellAttributes property. Or they are turned on implicitly by changing a single cell's attribute. They can be freed up by turning off HasCellAttributes, but this resets all attributes back to the protofield's default attributes.

There are no CellAttribute objects for the cells of nested table fields.

**Table Body**

The table body represents the set of columns in the table field. To access this part of a table field, use the TableField object's TableBody attribute. This attribute's data type is StackField. You can use this attribute to change visual properties of the table body's line boundary and background. The syntax is:

```java
field(tablefield).TableBody.attribute = value
```

**attribute**

Specifies an attribute of the StackField object. For example, the following statement sets the width of the outline of the table:

```java
field(custtable).tablebody.outlinewidth = LW_THIN;
```

The TableBody attribute is also useful in obtaining information about a table field (such as column names and data types) that is accessed at runtime but unknown at development time.

For a discussion and example of finding which table field a field is in, and then using the TableBody attribute to iterate through the Child Fields, see WhichTableField Method (see page 249).

**Control Button**

By default, each table field has a control button that provides the user with a menu of operations that can be performed on the table field. This menu appears when the user clicks the control button.

You can remove and add this menu. You can also change which operations appear on the menu. For instructions about performing these tasks, see How You Can Remove or Add the Control Button (see page 241).
Scroll Bar

The scroll bar lets users scroll up or down in a table field display to bring undisplayed rows in the array into view. A scroll bar is present by default on every table field when it is created.

To scroll horizontally (in a vertical table field), you can put the table field into a view port. A vertical table field can also have its own horizontal scrollbar. If a table field has \( n \) columns, and you set the ColumnsDisplayed attribute to a number less than \( n \), only the leftmost columns are displayed. However, if you set the HasHorizontalScrollBar attribute to TRUE, a scrollbar is displayed at the bottom of the table field that lets the user scroll to see all columns.

You can remove the scroll bar from the display or bring it back if it is not displayed. For more information, see How You Can Display and Remove the Vertical Scroll Bar (see page 237).

Summary of Table Field Access Syntax

The following table provides a quick reference for the correct syntax to access each part of a table field:

<table>
<thead>
<tr>
<th>Table Field Component (Data Type)</th>
<th>Reference Syntax and Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table field (TableField)</td>
<td>field(tablefield)</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>field(custtable)</td>
</tr>
<tr>
<td>Title</td>
<td>field(tablefield).title</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>field(custtable).title = 'New Customers';</td>
</tr>
<tr>
<td>- Text (varchar(256))</td>
<td>field(tablefield).titleTrim.attribute</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>field(custtable).titletrim.bgcolor = CC_PALE_BLUE;</td>
</tr>
<tr>
<td>Table header (StackField)</td>
<td>field(tablefield).TableHeader</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>field(custtable).tableheader.bgpattern= FP_SHADE;</td>
</tr>
<tr>
<td>Column (ColumnField)</td>
<td>field(tablefield[*].column)</td>
</tr>
<tr>
<td>Table Field Component (Data Type)</td>
<td>Reference Syntax and Example</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Example of setting a columnfield attribute:</td>
<td>field(custtable[*].custname).bgcolor = CC_RED;</td>
</tr>
<tr>
<td>ProtoField (FormField)</td>
<td>fieldtype(field(tablefield[*].column).ProtoField)</td>
</tr>
<tr>
<td>Example of setting a protofield attribute:</td>
<td>entryfield(field(custtable[*].custname).protofield).bgcolor = CC_RED;</td>
</tr>
</tbody>
</table>

**Column Header**

- **Text (varchar(256))**
  - Example:
  ```
  field(custtable[*].custname).title = 'Customer Name';
  ```
- **Visual characteristics (Free Trim)**
  - Example:
  ```
  field(custtable[*].custname).titletrimbgcolor = CC_RED;
  ```

**Table body (StackField)**

- Example:
  ```
  field(custtable).tablebody.outlinewidth = LW_THIN;
  ```

**Individual cell (CellAttribute)**

- Example:
  ```
  field(custtable[3].custname).bgcolor = CC_RED;
  ```

---

**Table Field Operations**

This section describes how to perform a variety of operations on a table field:

- Displaying and removing the vertical scroll bar
- Scrolling through data
- Scrolling to a specific row
- Displaying a horizontal scrollbar
- Removing or adding the control button
- Enabling and disabling user modifications to table field data
- Copying the contents of a table field to the Clipboard
- Column operations
- TableField methods
- Controlling multiple table fields with a single control button
- Building sort and find into every table field menu

**How You Can Display and Remove the Vertical Scroll Bar**

When you create a table field, OpenROAD automatically gives your table field a vertical scroll bar. You can remove this scroll bar by setting the HasScrollBar attribute of the TableField object to FALSE. The syntax for this operation is:

```java
field(tablefield).HasScrollBar = FALSE;
```

To redisplay a scroll bar, set the HasScrollBar attribute to TRUE.

**How You Can Scroll Through Data**

A user can scroll the displayed data in a table field in one of the following ways:

- Vertical scroll bar
- Navigation keys: arrow keys, Page Up, Page Down, Home, and End keys
- Single-character find

**Note:** Scrolling through data changes the current row to one of the visible rows and may also change selections. For more information about how selections are affected, depending on how the SelectionType and ScrollingChangesSelection attributes are set, see the *User Guide*.

**How You Can Use the Vertical Scroll Bar**

The scroll bar can be used to scroll line by line or page up or page down. You also can drag the thumb of the scroll bar for longer scrolls.
How You Can Use Navigation Keys to Scroll

The up and down arrow keys select a new row and cause scrolling if that row is not visible. Page Up and Page Down scroll a single page. The Home and End keys scroll to the first or last page of the table field.

**Note:** If the current field is changeable, the navigation keys apply to that field, provided the field is of a type that uses that particular key.

How You Can Use Single-character Find

When a user positions the cursor in a column that has an entry field as its protofield and types a letter or number, the cursor moves to the first row whose value in that column begins with the specified letter or number. Each time the user types the character, the cursor moves to the next row whose column value begins with that character.

In searching for a specified character, when the cursor reaches the last row of the table field, it loops back to the first row.

Because typing in a field whose mode is changeable actually displays the value in the field, you can use single-character find only when the column is Landable but not Changeable.

You can disable this feature for a column in either of the following ways:

- Use the Property Inspector to set the HasSingleCharacterFind attribute.
- Set the HasSingleCharacterFind attribute in your 4GL code.

For example, the following code disables this feature for the vid_no column of the vlist table field:

```plaintext
field(vlist[*].vid_no).HasSingleCharFind = FALSE;
```

To implement a custom find operation for a table field whose bias is changeable, use the ArrayObject's Find method. For an explanation about using the Find method, see How You Can Scroll Indirectly with the Find Method (see page 240).

How You Can Trigger SetValue, Exit, and Scroll Events

When a user scrolls through data using the single character find feature or an arrow key, the action triggers a SetValue event, an Exit event, or both, for the field being exited.

When a user uses the scroll bar, a SetValue or Exit event is triggered only if the row that had the current input focus scrolls out of the display. If the current row remains displayed, you can determine whether the user scrolled through the data by using a Scroll event.
The Scroll event is triggered when the user clicks the scroll bar of a table field with an interactive bias (FB_LANDABLE or FB_CHANGEABLE). The Scroll event is also triggered when scrolling occurs due to use of a scrolling key, Page Down, Page Up, Home, or End.

If a SetValue or Exit event is triggered by moving the scroll bar, these events are executed before the Scroll event. Because the Scroll event is not chained off the SetValue or Exit events, a resume statement in either of those events does not affect the running of the Scroll event.

For more information about Scroll events, see the Language Reference Guide online help. For more information about event chains, see Managing Event Queues (see page 317).

**How You Can Scroll to a Specific Row**

There are two ways to allow direct scrolling to a particular row:

- The SetInputFocus method of the TableField system class
- The ActiveRow attribute of the TableField system class

In addition, you can use ArrayObject's Find method to provide the user with the opportunity to specify a row for scrolling. After the user specifies the row, use the SetInputFocus method or the ActiveRow attribute to scroll the table field to the desired row.

**How You Can Use the SetInputFocus Method to Scroll**

You can use the SetInputFocus method to set the input focus of a frame to a specific cell of a table field. Its syntax is:

```
integer = tablefield.SetInputFocus(row=integer), [column=varchar(256)]
```

- **row**
  - Identifies the row in the array to which to move the input focus
  - **Default:** row 1

- **column**
  - Identifies the name of the column in the table field to which to move the input focus
  - **Default:** the first displayed column

The SetInputFocus method returns ER_OK (value of 0) if the parameters are valid, and it returns a nonzero number if there is an error.
When you use the SetInputFocus method, the change in focus occurs at the end of the current event block. The method triggers any appropriate Exit, SetValue, and Entry events. For example, the following event block accepts a number as input from the user and then scrolls the table field display to that row:

```plaintext
declare
    resp = varchar(80);
...

on click menu_button.ChangeRow =
begin
    resp = prompt 'Enter the number of the row '+
        'you want to see.:
    status = field(custtable).SetInputFocus(row =
        int4(resp));
    ...
end;
```

When you use the SetInputFocus method to specify a column, the column name must be quoted or contained in a varchar variable.

**How You Can Use the ActiveRow Attribute to Scroll**

The ActiveRow attribute identifies the array row that has the current input focus. If you set this attribute directly, OpenROAD scrolls the data display to the specified row, moves the input focus to the first Landable field in that row, and sets the CurRow attribute to the same specified number.

Setting ActiveRow attribute has no immediate effect. Instead, an internal event is queued that ultimately generates any necessary SetValue and Exit events as the focus attempts to move. To prevent getting incorrect results, therefore, do not set the ActiveRow attribute in an event block and then attempt to use its value in the same event block.

**How You Can Scroll Indirectly with the Find Method**

You can use the Find method of the ArrayObject class to find the row you want to scroll to, followed by setting ActiveRow to that row to do the actual scrolling.

**Horizontal Scroll Bars**

When the ColumnsDisplayed attribute of a vertical table field is smaller than the total number of columns, the table field may have a horizontal scroll bar. You can enable the horizontal scroll bar by setting the HasHorizontalScrollBar attribute to TRUE.
How You Can Remove or Add the Control Button

When you create a table field in OpenROAD Workbench, a control button containing the table field's operations menu is automatically associated with the table field. You can use the Property Inspector to remove the control button by setting the HasControlButton attribute to FALSE. You can also set the HasControlButton attribute in your 4GL code:

```c
field(tablefield).HasControlButton = FALSE;
```

Alternatively, you can set the table field's ControlButton attribute to NULL:

```c
field(tablefield).ControlButton = NULL;
```

If a table field does not currently contain a control button and you want to add one, you must specify an existing object of the ControlButton class. The syntax is:

```c
field(tablefield).ControlButton = control_button;
```

**Note:** If you remove the default control button from a table field and then decide later to add a control button containing the default operations, you must point to a control button that contains these operations.

How You Can Enable and Disable User Modifications to Table Field Data

You can modify data by doing the following:

- Updating existing data in a table field row
- Inserting new rows before existing rows
- Deleting rows
- Appending rows to the end of the table field

The following sections describe these operations in more detail.

How You Can Update Existing Data in a Table Field Row

The CurBias attribute of a field in the table field determines whether a user can directly edit that field. A field can be edited if its CurBias is FB_CHANGEABLE. The CurBias of the field is determined by the CurMode of the frame and the biases of the table field, column fields, and protofield fields.

You can use the Property Inspector to initialize the biases for each column of a table field and for the table field itself. By default, when you create a table field, the bias is FB_CHANGEABLE for each frame mode except FM_READ. The default bias for the FM_READ mode is FB_LANDABLE.
Biases can be dynamically changed in OpenROAD by using the CurBias, UpdateBias, QueryBias, ReadBias, User1Bias, User2Bias, User3Bias, or AllBias attributes of FormField objects. For more information about biases and frame modes, see Creating Dynamic Frames (see page 363).

You can disable updates to the whole frame by setting CurMode to FM_READ. You can disable updates to the entire table field by setting the table field’s bias to be a bias other than FB_CHANGEABLE. Or you could keep the table field's bias as FB_CHANGEABLE, and restrict a particular column by setting its bias to a bias other than FB_CHANGEABLE. For more information about CurBias, see the Language Reference Guide online help.

**How You Can Insert New Rows Before Existing Rows**

The user can insert rows before existing rows by using the Insert Before Current Row menu item of the table field’s control button.

Enabling or disabling Inserts can be done by enabling or disabling this menu item from OpenROAD. You can use the Option Menu Editor to determine the name of an item in the control button menu. To open the Option Menu Editor from the Frame Editor, select the control button and then click Edit, OptionMenu. For example, if the table field's control button is named tblfld_controlbutton, the following code can be used to disable the menu:

```c
field(tblfld_controlbutton.insertbefore).CurBias = MB_DISABLED;
```

The following table shows the menu text and menu button names for the table field control button:

<table>
<thead>
<tr>
<th>Menu Text</th>
<th>Menu Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Before Current Row</td>
<td>insertbefore</td>
</tr>
<tr>
<td>Delete Current Row</td>
<td>deletecurrent</td>
</tr>
<tr>
<td>Delete All Rows</td>
<td>deleteall</td>
</tr>
<tr>
<td>Copy Table</td>
<td>copytable</td>
</tr>
</tbody>
</table>

To enable the menu, use bias MB_ENABLED instead of MB_DISABLED. To remove this item from the menu, use bias MB_INVISIBLE.

By default, the insertbefore menu item is enabled only in FM_UPDATE and FM_QUERY modes, and disabled in FM_READ, FM_USER1, FM_USER2, and FM_USER3 modes.
How You Can Delete Rows

The user can delete rows by using the Delete Current Row or Delete All Rows menu items of the table field's control button.

You can enable or disable deletions by enabling or disabling these menu items. The following code exemplifies disabling the delete menu items:

```plaintext
field(tblfld_controlbutton.deletecurrent).
   CurBias = MB_DISABLED;
field(tblfld_controlbutton.deleteall).
   CurBias = MB_DISABLED;
```

By default this menu item is enabled only in FM_UPDATE and FM_QUERY modes, and disabled in FM_READ, FM_USER1, FM_USER2, and FM_USER3 modes.

How You Can Append Rows to the End of a Table Field

The user can append a row at the end of the table field by choosing the row below the last row with data, and then directly editing it (if appending is enabled). The CurOps attribute of the table field determines whether appending is enabled.

To enable a row to be appended, the CurOps attribute is set to OP_APPEND. To disable this capability, set CurOps to OP_NONE. CurOps can be set directly, or you can use one of the following attributes:

- ReadOps
- UpdateOps
- QueryOps
- User1Ops
- User2Ops
- User3Ops

You can use the Property Inspector to initialize these attributes. Also, they can be changed dynamically in 4GL.
The following settings determine whether a user can append rows to a table field:

**OP_APPEND**
Specifies that the user can append rows to the end of the array by moving the cursor past the last row displayed in the table field

**OP_NONE**
Specifies that the user cannot append rows to the end of the array by moving the cursor past the last row displayed in the table field

**OP_APPENDINSERT**
Specifies that the user can append or insert rows to the table field

**OP_APPENDEDDELETE**
Specifies automatic append at the end of the table field and that the user can delete rows

**OP_APPENDINSERTDELETE**
Specifies that the user can perform all operations on the table field

The setting that you specify with these attributes occurs only when the frame is in the corresponding mode. For example, if you set the QueryOps attribute while the frame is in Read mode, the QueryOps setting does not take effect until the frame is displayed in Query mode.

To change the setting immediately, regardless of the frame's mode, use the CurOps attribute of the TableField system class. This attribute accepts the same two settings as do the other operations attributes.

OpenROAD changes the operation setting for the current mode of the frame to the setting specified for the CurOps attribute. For example, if a frame is in Update mode and you set CurOps to OP_APPEND, then OpenROAD automatically sets the UpdateOps attribute to OP_APPEND also.

**How You Can Copy Data to PC Clipboard**

Copying to the PC Clipboard can be accomplished from OpenROAD with the TableField CopyToClipboard method. For more more information, see online help.

CopyTable is also an option on the default menu of a table field's control button. Data is copied in a format compatible with formats that many spreadsheet programs can use to import data.
Column Operations

This section describes the following operations that affect one or more columns in a table field:

- Displaying and removing column headers
- Formatting a multiline header
- Using hidden columns
- Turning CellAttributes on
- Using the HasDataChanged attribute

How You Can Display and Remove Column Headers

By default, OpenROAD puts a table header on each table field that you create. This area contains the column titles. You can remove the table header and, consequently, the column titles, by setting the HasHeader attribute of the TableField object to FALSE:

```csharp
field(tablefield).HasHeader = FALSE
```

If the table header has been removed and you want to display it, set the HasHeader attribute to TRUE.

How You Can Format a Multiline Header

If you want the title of a column to appear on multiple lines, use the OpenROAD system constant for a new line, HC_NEWLINE. For example, the following statement results in a two-line column title:

```csharp
field(movie[*].director).title = 'Director''s' + HC_NEWLINE + 'Name';
```

The result looks like this:

```
Director's
Name
```

How You Can Use Hidden Columns

Hidden columns are columns in a table field that are invisible to the user. They have a variety of uses, for example, as recordkeeping or status fields.
There are two ways to implement hidden columns:

- Set the bias of the column(s) to FB_INVISIBLE.

  You may want some table field columns to be visible to some users but not others or invisible only some of the time. For example, it might be acceptable for a supervisor to see a salary column but unacceptable for a clerk to see that information. To change a column to invisible, set the CurBias attribute of the ColumnField object to FB_INVISIBLE.

  The following code in a frame's initialize section determines who is using the application and sets the CurBias attribute for the columns in the frame's table field accordingly:

  ```
  select  :user_name = dbmsinfo('username'),
          :dba_name = dbmsinfo('dba');
  commit;
  if user_name != dba_name then
    field(custtable[*].acctbal).CurBias = FB_INVISIBLE;
  ```

  Use this strategy only if you want to change the visibility of the column during the running of frame, since it adversely affects performance.

- Define a user class that contains hidden columns as well as those displayed in the table field.

  This is the most efficient way to implement hidden columns. This way is useful for keeping a record of data that might be changed by the user.

**How You Can Turn CellAttributes On**

The CellAttribute object gives you access to the attributes of an individual cell in a table field column. Because of the impact on performance, OpenROAD does not automatically create CellAttribute objects when you create a table field. Instead, if you want to use CellAttribute objects, you must explicitly or implicitly turn them on.

CellAttribute objects are turned on by column rather than on a cell-by-cell basis. Turning on CellAttributes for a column sets it on for all cells in that column.

The easiest way to turn on CellAttributes for a column is to use the Property Inspector to set the column's HasCellAttributes attribute to TRUE. This involves the least performance overhead. However, you can choose to turn CellAttributes on at runtime by using one of the following techniques:

- Explicitly, in the initialize portion of a frame script

  To turn on the CellAttributes explicitly, set the HasCellAttributes attribute of the ColumnField object to TRUE. The syntax is:

  ```
  field(tablefield[*].column).HasCellAttributes = TRUE
  ```
Implicitly, by referencing one of the CellAttribute’s attributes in your script
If they are not already on, the CellAttributes are implicitly turned on when your script sets a cell attribute for a cell. For example, the following event block changes a cell’s background color to red when a user tabs into a column. The first time the user enters the specified column, the cell attributes are turned on for that column.

```plaintext
on childentry custtable[*].custname = 
    begin
        field(custtable[].custname).bgcolor = CC_RED;
    end;
```

Setting CellAttributes implicitly has the advantage of creating CellAttributes only when needed. You cannot turn on CellAttributes for the cells of a nested table field.

**How You Can Use the HasDataChanged Attribute**

If cell attributes are turned on for a given column field, you can fetch the CellAttribute object for a cell by using the column field CellAttribute method. The following example shows the syntax for fetching the CellAttribute object that corresponds to the cell in a column field:

```plaintext
cellattribute = columnfield.CellAttribute(record = integer)
```

*integer*

Specifies the row index of the cell

Having fetched the CellAttribute for a cell, you can use its HasDataChanged attribute to see whether the user has made any changes to the data in the cell. The HasDataChanged attribute is:

- TRUE if the user types in an entry field cell or changes a cell of any other type
- FALSE when the value of the cell is changed by a 4GL script

A 4GL script can also set the HasDataChanged attribute of a CellAttribute, a column field, or an entire table field. For example, you might set HasDataChanged to FALSE after processing changes.

If you intend to use this attribute, ensure that its information is accurate by turning on the cell attributes explicitly in either of these ways:

- In the Property Inspector when you create the frame
- By setting the HasCellAttributes attribute when the frame is initialized at runtime
Table Field Operations

TableField Methods

This section describes the methods defined for the TableField object. For more information about these methods, see the Language Reference Guide online help.

InsertColumn and DeleteColumn Methods

These methods are used by the Frame Editor for constructing table fields dynamically. The InsertColumn method inserts a new column into a table field, while the DeleteColumn method removes an existing column.

Do not use the DeleteColumn method to make a column invisible. For more information about hiding a column, see How You Can Use Hidden Columns (see page 245).

WhichRow Method

A table field generally displays a subset of the rows in its associated array. For example, the table field may display only four rows at a time, although the underlying array contains 20 rows. Therefore, a data item currently displayed in the third row of the table field might actually belong to the thirteenth row of the array.

To determine which array row contains the value in a specific cell in a table field, use the WhichRow method. The syntax for WhichRow is:

```plaintext
integer = tablefield.WhichRow(cellfield = FormField)
```

The return value identifies the array row that contains the data displayed in the specified cell field. If the specified cell field is not part of the table field or is part of an empty row, this method returns zero.

One common use of this method is in ChildExit event processing, to determine whether the user has exited a row within a table field rather than moved to another column within the same row:

```plaintext
if field(custtable).currow !=
   field(custtable).WhichRow(cellfield=
       curframe.targetfield)
then
   /* The row is being exited */
endif;
```
This expression on the left side of the if statement:

\[ \text{field(cust_table).currow} \]

returns the number of the array row that last had the input focus. The expression on the right side, because the specified cell field is the target field, returns the array row number of the row that gets the input focus next. If the row numbers returned by these two expressions are not the same, then the user has moved the cursor up or down a row.

**WhichTableField Method**

WhichTableField, a method of the FormField class, indicates if a field is in a table field, and if so, which one.

Because a single control button can manipulate the data in more than one table field, it is important to be able to determine which table field is to be manipulated. You can use the WhichTableField method to determine whether a field is in a table field and, if it is, to return the table field object to a variable. Using this method, you can access all the information about a specific table field, such as its name, column names, and data, without knowing at development time which field is to be accessed.

The syntax for WhichTableField is:

\[ \text{table_field_var = formfield.WhichTableField()} \]

This method returns one of the following values:

- Null if the field is not in a table field
- The table field in which the form field resides
- The innermost table field if the field is in a nested table field

When a control button is part of a table field and the user invokes one of its menu operations, OpenROAD uses the WhichTableField method to identify the table field. The control button’s script invokes the WhichTableField on the TriggerField. Because the control button's menu field triggers the event, and because the control button is directly associated with the table field, OpenROAD can identify the table field.

However, if the control button is independent (not attached to a specific field or table field), you must write your own code to determine which field triggered the control button invocation. If you are dealing with non-nested table fields, you can use FrameExec’s InputFocusField attribute in conjunction with the WhichTableField method to determine the table field to be handled.
For example, the following code first checks whether the cursor is on a form field. If it is, the code checks whether the current field is a table field. If the current field is not a table field, the code sends an appropriate message to the user.

```plaintext
table_field = null;
if CurFrame.InputFocusField is not null then
    table_field = CurFrame.InputFocusField.WhichTableField();
endif;
if table_field is null then
    message 'You must be in a table field to sort it.,'
    resume;
endif;
```

If the input focus is not on any field when the WhichTableField method is invoked, an error message is written to the trace window and the log file.

This code checks that the field is not null before proceeding, thus preventing the trace window message.

If the field is a table field, the WhichTableField method returns the table field to the user-defined variable `table_field`. You can obtain information about this table field using attributes and methods provided by the OpenROAD system classes. For example, you can:

- **Access array values**
  
  To manipulate values in the underlying array, you need a reference variable that points to it. To load values from the underlying array into the array reference variable, use the GetFieldValue method of the FieldObject system class, for example:

  ```plaintext
  ... 
  tf_array = ArrayObject default null;
  ...
  table_field.GetFieldValue
  (value = byref(tf_array));
  ```
Get information about table field and array attributes

The InputFocusField attribute makes it easy to determine the name of the column from which the user invoked a control button operation (provided the column does not display a nested CompositeField). The following code gets the name of the current column into the col_name variable:

```plaintext
col_name = CurFrame.InputFocusField.Name;
```

To obtain information about all of the table field's columns, such as their names, data types, and current biases, you must loop through the table field's column fields. For example, the following code obtains and displays the names and data types of all the columns in table_field:

```plaintext
i = 1;
while i <= table_field.TableBody.ChildFields.LastRow do
    message 'column name:' +
    table_field.TableBody.ChildFields[i].Name;
    message 'col_title:' +
    columnfield(table_field.TableBody.ChildFields [i]).Title;
    i = i + 1;
endwhile;
```

The TableBody attribute of the TableField system class contains the set of column fields for the table field. Because TableBody is of type StackField, it has access to the ChildFields attribute defined for its parent, CompositeField.

To iterate through the columns of a table field, use the following syntax:

```plaintext
tablefield_refvar.TableBody.ChildFields[x]
```

To apply ColumnField attributes (such as ProtoField and Title) to the value returned by this iteration process requires casting the returned value to an object of the ColumnField data type.

For more information about using the TableBody attribute, see the Language Reference Guide online help. For more information about casting, see How You Can Work with Attributes (see page 140).

For an example of using the WhichTableField method, see How You Can Sort Table Field Data Generically (see page 252).
Controlling Multiple Table Fields with a Single Control Button

You can use one control button to provide operations for multiple table fields. In fact, if you have multiple table fields on a frame, it is necessary to use a single ControlButton so that the Control Button accelerator keys will be applied to the table field that currently has input focus. Assume you have a frame with two table fields, for example:

- **Checkouts**
  This table field tracks DVDs as they are checked out and in by each customer.
- **Reservations**
  This table field tracks reservations held by customers for specific DVDs.

As customers check out a DVD, they also can reserve another disc.

You could add operations to the control buttons provided by default with each of these table fields. However, to reduce duplication of code, you can write generic code for operations common to both table fields and provide a single control button for both.

Independent control buttons are created with no menu operations. To include the standard table field operations (Insert Before Current Row, Delete Current Row, and Delete All Rows) that are provided by default with every table field control button, you can copy the 4GL code for these operations into the independent control button. You can find this code using View Processed Script in the Frame Editor. These three default table field operations are excellent examples of coding table fields generically.

How You Can Sort Table Field Data Generically

Sorting array data by specifying a column in its associated table field is a common operation. You can put a generic sort operation into a control button that is attached to a specific table field or is an independent control button field.
The following event block for a Control Button's Option Menu's Menu List lets users perform this operation, determining the names of the table field and column at runtime:

/* Script for MENULIST */

on click =

declare
table_field = TableField default null;
tf_array = ArrayObject default null;
col_name = varchar(32) not null;
sort_order = integer not null;
enddeclare

begin
/* Find out what table and column we are in, ** if any. */
if CurFrame.InputFocusField is not null then
table_field = CurFrame.InputFocusField.WhichTableField();
endif;
if table_field is null then
message 'You must be in a table field to sort it. ';
resume;
endif;

col_name = CurFrame.InputFocusField.Name;
/* Point tf_array to the array underlying ** the tablefield */
table_field.GetFieldValue(value = byref(tf_array));
/* ** Use CurEnumValue to specify sort order. ** Note that the first item in the menu list ** is labeled sort in ascending order, and the ** second item to sort in descending order. */
if MenuList(CurFrame.TriggerField).CurEnumValue = 1
then sort_order = AS_ASC;
else
sort_order = AS_DESC;
endif;
/* Note the colon in front of col_name is required ** to indicate that col_name is a variable, not the ** literal column name. */
tf_array.sort(col_name = sort_order);
end;

For more information about the Sort method of the ArrayObject system class, see How Sorting Arrays Works (see page 223). For information about using the WhichTableField method of the TableField system class, see WhichTableField Method (see page 249).
A **collection** is an object that contains a group of related objects. For example, the Excel Worksheets collection object in the following sample code contains Worksheet objects. Each object within a collection is called an element of that collection. Because collections are objects, they have properties and methods, just as single objects do.

Declared collections can use the OpenROAD array syntax for indexing. Undeclared collections must use the standard item method for indexing. The following sections provide examples of how this is done. For the collections syntax, see the *Language Reference Guide* online help.

### How You Can Index into a Declared Collection

The following code shows how to index into a collection when the collection has been defined to OpenROAD as a collection:

```plaintext
initialize()=
defclare
    app = application;
    wbs = workbooks collection of workbook;
    x = integer;
    y = varchar(20);
    awbs = array of workbook;
enddeclare

begin

    // Show Microsoft Excel
    app.visible = 1;

    // Get the workbooks collection
    wbs = app.workbooks();

    // Open a workbook and put it into an OpenROAD
    // array
    awbs[1] = wbs.open
        ('c:\msoffice\excel\examples\samples.xls');

    // Index into the OpenROAD array to access a workbook
    // and set an attribute.
    [1].author = 'John Smith';

    // Index into the workbooks collection through an integer
    // constant and set an attribute of the workbook
    // being indexed.
    wbs[1].author = 'Jane Smith';
```
In the preceding code, the Workbooks method returns a Workbooks collection of Workbook objects. The following sections show various ways of indexing into this collection.

### How You Can Index into an Undeclared Collection

The following code shows how to index into a collection when the collection has not been defined to OpenROAD as a collection:

```plaintext
initialize()=
  declare
    app = application;
    wbs = workbooks;
    x = integer;
    y = varchar(20);
    awbs = array of workbook;
  enddeclare

begin
  // Show Microsoft Excel
  app.visible = 1;

  // Get the workbooks collection
  wbs = app.workbooks();
```

How You Can Index into an Undeclared Collection
// Open a workbook and put it into an OpenROAD array
awbs[1] = wbs.open('c:\msoffice\excel\examples\samples.xls');

// Index into the OpenROAD array to access a workbook and set an attribute.
awbs[1].author = 'John Smith';

// Index into the workbooks collection through an integer constant and set an attribute of the workbook being indexed.
Wbs.item(1).author = 'Jane Smith';

// Index into the workbooks collection through a string constant and set an attribute of the workbook being indexed.
wbs.item('samples.xls').author = 'Sam';

// Index into the workbooks collection through an integer variable and set an attribute of the workbook being indexed.
x = 1;
wbs.item(x).author = 'Sally';

// Index into the workbooks collection through an integer variable and set an attribute of the workbook being indexed.
y = 'samples.xls';
wbs.item(y).author = 'unknown';

// Index into the workbooks collection through a string constant and call a method of the workbook being indexed.
wbs.item('samples.xls').close(TRUE,'c:\save.xls',FALSE);

// Quit Excel.
app.quit();
end
Chapter 8: Working with List Views and Tree Views

This section contains the following topics:

- How You Can Use Listview and Treeview Fields (see page 257)
- Listview_Treeview_Demo (see page 257)
- Listview Fields (see page 260)
- TreeNode and Tree Classes (see page 266)
- Treeview Fields (see page 269)

This chapter describes what you need to know when using listview fields and treeview fields in your OpenROAD applications.

A list view field lets the end user view and optionally manipulate or edit a list of items. A tree view field presents the end user with a hierarchical list of items in a tree structure that can be expanded or collapsed.

How You Can Use Listview and Treeview Fields

Listview fields and treeview fields provide different ways to visually represent data in your application. You use the ListviewField class to create a list view. For example, OpenROAD Workbench uses a listview field to display the lists of applications and their components. A treeview field visually represents a tree. It is created using the TreeviewField class and other related classes.

The following sections describe how to create listview and treeview fields for use in your application.

For a complete description of the attributes and methods of the ListviewField and the TreeViewField classes, see the Language Reference Guide online help.

Listview_Treeview_Demo

Before proceeding with this chapter, you should import the Listview_Treeview_Demo as an application into OpenROAD Workbench. It is the basis for much of the discussion in this chapter. The following sections describe how to import it and access it.
Import the Demonstration Application

Before you can use the Listview_Treeview Demo application, you must import it into OpenROAD Workbench.

To import the Listview_Treeview Demo application

1. Start OpenROAD Workbench.
2. Select a database you can import the application into on the Connect tab.
3. Click the Develop tab and click the Applications portlet to make it active.
4. Click File, Import.
   The Import an Application dialog appears.
5. Click Browse and navigate to the %II_SYSTEM%\ingres\w4glsamp\lvtvdemo directory.
6. Select the file lvtvdemo.exp as the input file and click Open.
7. The path and file name are displayed in the Input File field.
8. Enter Listview_Treeview Demo in the Application Name field and click Go.

   The demonstration application is imported and displayed in the Applications portlet.

   To run the Listview_Treeview Demo, see Run the Demonstration Application (see page 259).
Run the Demonstration Application

To run the imported demonstration application

1. Select the Listview_Treeview_Demo application in the Applications portlet of the Develop tab.
2. Click Run, Run.

The Run the Examples dialog is displayed:

The demonstration dialog presents three choices of example frames that demonstrate listview and treeview fields. These frames are:

Listview Example

Displays a list of names and telephone extensions. For more information about this example, see Listview Fields (see page 260).

Treeview Example 1

Displays a simple, fixed tree. It shows the basic elements involved in creating a tree display. For more information about this example, see How You Can Build a Tree (see page 270).

Treeview Example 2

Displays a complex tree. It shows the hierarchy of fields in a frame. For more information about this example, see How You Can Attach Information to a Tree Node (see page 273).

To exit from this dialog or from any of the examples, click the standard close button (X).
A list view field lets the end user view and optionally manipulate or edit a list of items. A listview field provides a versatile way to display data and offers several display styles. When you run the Listview Example in the demonstration application, it displays an example of how a listview field can be used:

![Listview Example](image)

**Note:** The first column of a listview must be left-aligned.

The Listview Example displays a list of names and telephone extensions. You can sort the list by clicking a column header. You can also use the Style menu to change the style of the listview display.

**How the Listview Example Works**

The remainder of this section refers to the listview_example frame of the Listview_Treeview_Demo application. Use the Frame Editor to examine the listview_example frame online and its 4GL script as you read.
In the Frame Editor, the listview_example frame is displayed:

The listview_example frame contains an empty listview field named lview and a matrix field that contains a number of image trim fields. The bitmaps of the image trim fields are used with the listview. When the frame runs, the matrix field is hidden.

Use the Property Inspector to examine the ColAttributes attribute of the listview field. ColAttributes is an array of ListviewColAttr objects, and determines the number and order of columns in the listview field. The ListviewColAttr contains the column width and header text, and lets you specify how the text is aligned in the column header.

The listview_example frame uses a user class, name_and_extension. This class has the following attributes:

**last_name**
- Specifies a person's last name

**first_name**
- Specifies a person's first name
**extension**

Specifies a telephone extension

**image_index**

Specifies an integer

Now, to display the frame script to understand the following discussion, click Tools, Script on the floating menu bar.

The following script is displayed:

```
**
** The Style menu allows the user to display the listview in one of the four available styles.
**
*/

initialize()=
declare
    lv = listviewfield;
    names = array of name_and_extension;
/
** sort_column is the number of the column on which the listview is currently sorted. sort_direction is the current sort direction (either AS_ASC or AS_DESC).
*/
    sort_column = integer not null;
    sort_direction = integer not null;
    get_image_index = procedure returning integer not null;
    load_listview = procedure;
    set_image_indexes = procedure;
    set_listview_indexes = procedure;
    sort_names_array = procedure;
enddeclare
```

The script declares a variable, names, as an array of name_and_extension. The populate_name_array procedure loads the names array with data.
The set_listview_images procedure sets up the SmallImageList and LargeImageList arrays of the listview field. SmallImageList and LargeImageList are arrays of bitmap objects. In this example, the bitmaps display letters, and the \textit{nth} item of each array is a bitmap representing the \textit{nth} letter of the alphabet.

The next step in setting up the listview field is to associate bitmaps with each item in the names array. This example uses the bitmap corresponding to the first letter of the last name. The set_image_indexes procedure calculates the appropriate bitmap index for each item in the names array and stores the index in the image_index attribute.

Initially, the name array is sorted by last name. The sort_names_array procedure sorts the array using the values of sort_column and sort_order to determine how the array is sorted.

Finally, the load_listview procedure is called to load the listview field with data.

**How You Can Load the Listview Field with Data**

A listview field is a choice field, and like other choice fields has a ValueList attribute. The ValueList attribute is a ChoiceList. Unlike the other choice fields, the ChoiceItems attribute of the ValueList of a listview field is an array of ChoiceDetail objects. Each ChoiceDetail object contains information about a row in the listview field.

The ChoiceDetail class has the following attributes:

- **EnumDisplay**
  - Contains the text, which is displayed in the first column of the listview
- **EnumText**
  - Contains a text string
- **EnumValue**
  - Contains an integer value that you may use to identify an item in the list
- **isSelected**
  - Indicates at runtime whether the item is currently selected
- **EnumBitmap**
  - Is not used by listview fields
- **EnumSubText**
  - Specifies an array of StringObjects. The \textit{nth} item in the array contains the text that is displayed in column \(n + 1\) of the listview.
**ImageIndex**

Specifies an index into the arrays SmallImageList and LargeImageList that are associated with the listview field. These arrays are arrays of BitmapObjects. The ImageIndex attribute let you specify a bitmap to be displayed with a listview item.

When you examine the load_listview procedure, you can see how data is loaded into the listview field. The first step clears the ChoiceItems array. The procedure uses the AddItem method of the ChoiceList to add each new ChoiceDetail object to the ChoiceList. The AddItem method adds a ChoiceDetail object to the array and returns the position of the new item in the array. After using AddItem to create a ChoiceDetail object and add it to the ChoiceList, it sets the first name and extension in the EnumSubText array and sets the ImageIndex attribute using the previously calculated value.

Finally, after adding the items to the ChoiceList, it invokes the UpdChoiceList method of the listview field and causes the new data to be displayed.

**Listview Field Styles**

You can experiment with various field styles by selecting different styles from the Style menu of the listview_example. The Style attribute of a listview field determines how the listview is displayed. The possible styles are:

LVFS_DETAIL

Specifies that the data should be displayed in columns

LVFS_LIST

Specifies that only the values from the first column of the detail view (EnumDisplay) should be displayed. The data is arranged in columns, and ordered so that you read from top to bottom of the first column and then continue to read from top to bottom of all subsequent columns, left to right.

LVFS_ICON

Specifies that the data should be displayed with large icons (bitmaps). Only the values from the first column of the detail view are displayed. The data is arranged in rows so that you read across the first row, then across the second row, and continue to read across subsequent rows, top to bottom.
LVFS_SMALLICON

Specifies that the data should be displayed with small icons (bitmaps). Only the values from the first column of the detail view are displayed. The data is arranged in rows so that you read across the first row, then across the second row, and continue to read across subsequent rows.

The sizes of the icons are determined by the following listview field attributes:

- SmallIconHeight with a default of 16
- SmallIconWidth with a default of 16
- LargeIconHeight with a default of 32
- LargeIconWidth with a default of 32

Listview Field Events

Several useful events are associated with listview fields. The script for listview_example illustrates how you can use them. The script also shows how you can resize a listview field on a WindowResized event.

If you run OpenROAD Workbench with the trace window enabled, the listview_examples frame posts messages to the trace window when one of the following events occurs:

- Click
- DoubleClick
- HeaderClick
- SetValue

The messages in the trace window display the order in which events are generated. These are:

**Click and DoubleClick Event**

The first click causes a SetValue event. A DoubleClick event occurs when the user double-clicks the text in the first column or on the associated bitmap. The second click, if done quickly, causes a DoubleClick event. In listview_examples, a message is displayed in the frame's status bar when a DoubleClick event occurs. You might use the DoubleClick event to do something such as updating another frame.

**Note:** You may lose a DoubleClick event if too much time is spent processing the preceding SetValue event.
HeaderClick Event

Occurs when the user clicks one of the column headers of the listview field when the style is LVFS_DETAIL. The ReasonCode attribute of the FrameExec contains the number of the column. In the listview_example, the listview data is sorted on the HeaderClick event.

SetValue Event

Occurs when the style is LVFS_DETAIL and the user clicks the text in the first column or on the associated bitmap. In the other modes, a SetValue event occurs when the user clicks either the text or the bitmap. In listview_examples, a message is displayed in the frame’s status bar when a SetValue event occurs.

TreeNode and Tree Classes

The TreeviewField class has two related classes that you must use to work with it—Tree and TreeNode. Because they are interrelated, before you can understand treeview fields you must first understand the Treenode and Tree classes as described in the following sections.

TreeNode Class

The TreeNode class represents a node that contains data for a single node in a tree. The following sections describe how its attributes affect a treeview field.

Appearance

When displayed in a treeview field, the following attributes of a treenode affect the appearance of the node:

TextLabel

Specifies the text that is displayed for the node

BmpLblIndx

Specify an index into the ImageList array, and specifies a bitmap label for the node. A treeview field has an attribute ImageList, which is an array of bitmap objects.

SelBmpLblIndx

Specifies an index into the ImageList array, and specifies a bitmap label, which is displayed when the node is selected.
Node Relationships

When a treenode is part of a tree, there are a number of attributes that provide information about the node's relationship to other nodes in the tree. These are all read-only attributes, and cannot be set directly.

**ParentNode**
Indicates the node's parent node. If the node has no parent (that is, it is the root node of the tree), the ParentNode attribute is NULL.

**Children**
Indicates the number of immediate child nodes

**Descendants**
Indicates the number of descendant nodes

**FirstChild**
Indicates the first child node (NULL if there is no child node)

**LastChild**
Indicates the last child node

**NextSibling**
Indicates the next sibling node

**PrevSibling**
Indicates the previous sibling node

Tree Class

The *Tree class* is used to manage data that has an hierarchical structure. A *tree* contains a collection of treenode objects. The top treenode in the hierarchy is the root node. The following sections describe its attributes and methods.

Attributes

The Tree class has two attributes:

**RootNode**
Specifies the tree's root node

**Nodes**
Specifies the total number of nodes in the tree
TreeNode and Tree Classes

Maintenance Methods

Use the following methods to maintain the tree:

**AddNode**

Adds a node to the tree

**DeleteNode**

Deletes a node from the tree

Node Location Methods

Use the following methods to find nodes in a tree:

**NodeByName**

Finds a node with a given name, or TextLabel

**NodeByKey**

Finds a node with a given key. When you add a node to a tree, you may specify a unique key. The NodeByKey method is useful when the nodes in a tree do not have a unique name, and the NodeByName method does not find a unique node.

Traversal Methods

The following methods let you traverse the entire tree:

**StartTraverse**

Lets you traverse the entire tree

**NextNode**

Returns the next node after you have started to traverse the tree

**StopTraverse**

Stops the traverse operation

AddNode Method

The AddNode method creates a node and adds it to the tree. It is the basic method for building a tree.

This method has the following syntax:

```
treenode = tree.AddNode
(   name         = varchar(256),
    [keylabel     = varchar(256),]
    [relative     = treenode,]
    [relation     = integer]
);```
The AddNode method has one mandatory parameter, and three optional parameters:

**name**
(Required) Specifies the node's TextLabel attribute

**keylabel**
(Optional) Lets you specify a unique string that is associated with the new node. If you specify a key label, you may use the NodeByKey method to search for the node.

**relative**
(Optional) Specifies an existing node in the tree. If relative is not specified or is null, the new node becomes the root node of the tree; the old root node, if it exists, becomes a sibling of the new root node.

**relation**
(Optional) Specifies the relation of the new node to relative. The possible values of relation are:

- **TN_LASTCHILD**
  Specifies that the new node is the last child of relative. This is the default if relation is not specified.

- **TN_FIRSTCHILD**
  Specifies that the new node is the first child of relative

- **TN_NEXTSIBLING**
  Specifies that the new node is the next sibling of relative

The AddNode method returns the newly created node.

---

**Treeview Fields**

A *tree view field* presents the end user with a hierarchical list of items in a tree structure that can be expanded or collapsed. The Tree attribute of a treeview field is a tree object.
How You Can Build a Tree

Example 1—Treeview:

This example illustrates how you can build a fixed tree when you do the following:

1. Select Treeview Example 1 on the Run the Examples dialog.
2. Expand the first node of U.S. Government by clicking the first box that contains a plus sign.
3. Continue to expand each of the U.S. Government nodes.

The following displays:
If you look at the script for the frame tree_example_1, you can see how the build_simple_tree procedure uses the AddNode method to build the tree. Fixed trees are not often needed in applications, however. In most cases, the contents of the tree depend on the data available at runtime, and you must build the tree dynamically.

**Example 2—Treeview:**

This example shows how you can build a tree dynamically. To use this example, do the following:

1. Select Treeview Example 2 on the Run the Examples frame.

Two windows are displayed. The first is the Sample Frame:
The second window is the Treeview Example 2:

2. Expand as many nodes as you want to display on Treeview Example 2.

   If you expand all the nodes, it displays a tree showing the hierarchy of fields in the first frame:
3. Select a field in the tree. For example, stk1(stackfield).

The background of the corresponding stack field in the first frame changes to a different color:

![Sample Frame](image)

The next time you select the item in the tree, the background of the corresponding field reverts to its original color. You can continue to select items in the tree, or click the standard close button (X) to exit the example.

In this example the tree is based on the hierarchical relationship of the fields in the frame sample_frame. The tree_example_2 is called with a FrameExec as a parameter. The tree is built with the recursive procedure, add_a_node. Given a field, add_a_node adds a node corresponding to the given field, and then calls itself to add nodes for any of the field's children.

**How You Can Attach Information to a Tree Node**

Every class has an attribute, ClientData, of type Object. The ClientData attribute provides a convenient way to attach data to a tree node. In tree_example_2, the add_a_node method attaches a tree_ex2_class object to each tree node.
The tree_ex2_class has the following attributes:

**fld**
Specifies a formfield, the field that corresponds to the node

**originalbgcolor**
Specifies an integer, the original background color of the field

**color_changed**
Specifies an integer, a flag to indicate that the background color was changed

The fld attribute allows access to the field on the frame sample_frame. How You Can Use Bitmap Labels (see page 274) shows how the other two attributes are used.

### How You Can Use Bitmap Labels

A treeview field has an attribute, ImageList, which is an array of bitmaps. In tree_example_2, the set_up_bitmaps procedure sets up the ImageList. In this case, there are three bitmaps: a gray circle, a blue circle, and a red circle. Each of the variables gray, blue, and red contains the index in the ImageList array of the corresponding bitmap.

A tree node has attributes BmpLblIndx and SelBmpLblIndx. You may specify bitmap labels to be displayed with the node by setting BmpLblIndx and SelBmpLblIndx. If BmpLblIndx is a valid index in the array ImageList, the corresponding bitmap label is displayed when the node is not selected; if SelBmpLblIndx is a valid index, the corresponding bitmap label is displayed when the node is selected.

The first treeview example uses no bitmap labels. The second example illustrates the use of bitmap labels. When a node is selected, its bitmap label is blue. Initially, the bitmap label is gray when a node is not selected. When you select a node with a gray bitmap label, the background color of the corresponding field is changed to red, and BmpLblIndx is changed to red. When you select a node with a red bitmap label, the background color of the corresponding field is changed back to its original color, and BmpLblIndx is set to gray. The code that changes the bitmap labels is in the event block on setvalue tvf.

### How You Can Delete Nodes

You can use the DeleteNode method to delete a node from a tree.
Treeview Field Events

The SetValue event occurs when the user clicks the text or the bitmap label of a node; when this happens, the node's IsSelected attribute is set to TRUE.

Two other events that you may find useful are the Expanded and Collapsed events. These events occur when the user expands or collapses a node by clicking the box that contains a plus sign or a minus sign. The second example displays a message in the frame's status bar when either of these events occurs.
Chapter 9: Working with Images and Text Strings

This section contains the following topics:

- How You Can Work with Images (see page 277)
- How You Can Work with Text Strings (see page 287)

This chapter describes how to use images and text strings in an OpenROAD application.

The first section, How You Can Work with Images (see page 277), covers the following topics:

- Using image fields, image trim, and palette fields
- The BitmapObject class and how to use it to store images
- The ImageField object and how to use it to display bitmap images

The second section, How You Can Work with Text Strings (see page 287), includes the following topics:

- Storing strings in files and in a database
- Displaying strings from a database
- Updating strings in a database
- Deleting strings from a database
- Creating a string storage table

How You Can Work with Images

OpenROAD lets you display two-color (monochrome) or full-color bitmap images using the following field objects:

- Image fields
- Image trim
- Palette fields

The following sections describe these field types.
Image Fields

An image field is a rectangular boundary, like a picture frame, in which you can display bitmap images that are stored either in files or in the database. Image fields differ from most other field types in the way they store data.

For most field types, the data to be displayed in the field is contained in the variable that is associated with the field. For example, the integer variable associated with a slider field contains the current value for the slider. However, the data for an image field consists of bitmap images.

Instead of actually containing the data for the field, the reference variable associated with the image field points to an object of the BitmapObject class. In addition to containing the bitmap image that is the current data for the field, the bitmap object referenced by the image field's variable provides attributes and methods that enable:

- Loading an image from a file into OpenROAD
- Updating the images in the database or files

How You Can Load a Bitmap Image into an Image Field

To load a bitmap image that is stored in a file into OpenROAD and display it in an image field, set the FileHandle attribute of the BitmapObject, for example:

```csharp
bmo.FileHandle = file_name.Value;
```

For more information about using the BitmapObject system class, see BitmapObject Class (see page 282).

Relationship Between BitmapObject and ImageField Object

Although the BitmapObject object has information about the image displayed in the image field, the ImageField object has information about the field itself, such as the size of the field. The most important attribute of the ImageField class is the DisplayPolicy attribute, which determines how the image fits into the field. For more information about the DisplayPolicy attribute, see ImageField Object (see page 286).
The following illustration of an ImageField and a BitmapObject displays the relationship between the two variables associated with the image field, the BitmapObject and the ImageField. For more information about using the ImageField system class, see ImageField Object (see page 286).

Image Trim

Unlike an image field, image trim displays a single bitmap image. When you create image trim in OpenROAD Workbench, you specify the name of the file that contains the bitmap image. At this point, OpenROAD copies the bitmap image from the file into the bitmap object associated with the image trim. OpenROAD saves this image as part of the form.

Although it is possible to replace this image at runtime by setting the FileHandle attribute of the BitmapObject class (see How You Can Store Images in Files (see page 283)), generally you do not need to do so.

Other Fields

You can use bitmap images also on button fields, toggle fields, bar fields, and as a background image for all active fields on a form including the form itself.
Palette Fields

PaletteField objects display a list of values, represented as images, from which the user selects a single value. Like radio fields, all choices are always displayed in a palette field. Therefore, the list should contain few choices.

Each value in a palette field has text, an associated numeric value, and an image. PaletteField objects are mapped to variables of either varchar or integer data type. However, regardless of the variable mapping, the palette field always displays the list of choices as images.

How You Can Insert an Image into a Palette Field

To insert a bitmap object programmatically into a palette field, use the AddBitmapItem method defined for the ChoiceList system class. You would follow these basic steps:

1. Declare a variable of type BitmapObject.
2. Load a DBHandle or FileHandle into the bitmap reference variable.
3. Insert the image into the palette field using the AddBitmapItem method.
4. Refresh the display using the UpdChoiceList method defined for the ChoiceField system class.

For example, the following statement from an initialize block adds a bitmap image from the v_video_graphics table to the first item in a palette field named vidchoices:

```plaintext
... declare
   bo = BitmapObject;
enddeclare
begin
   bo.DBHandle = 'v_video_graphics:3';
   field(vidchoices).ValueList.
      AddBitmapItem
         (enumvalue=1, textvalue = 'comedy',
          bitmapvalue = bo);
...```

To insert an image into each value of a palette field, declare an array of BitmapObject and load the appropriate handle into each row of the array.

How You Can Arrange the Display of Palette Fields

Because palette fields (like radio fields) display all choices on the window simultaneously, they can be arranged to display in several columns. To arrange the display in the Frame Editor, specify the number of columns on the field's Property Inspector. To arrange the display programmatically, use the Columns and Orientation attributes defined for the PaletteField system class.
The Columns attribute stores the number of columns that can be displayed horizontally or vertically (depending on the value of the Orientation attribute). Vertical columns display their values from top to bottom. Horizontal columns display from left to right. The values of the Columns and Orientation attributes together determine the style of the field.

The order of the displayed values is always along the primary axis. For example, a five-cell palette field displaying the numbers 1, 2, 3, 4, and 5 displays as follows when organized into two vertical columns (by setting the Columns attribute to “2” and the Orientation attribute to Vertical):

```
| Columns = 2 |
| Vertical   |
| 1  | 4 |
| 2  | 5 |
| 3  |   |
```

The same palette field displays as follows when organized into two horizontal rows (by setting the Columns attribute to “2” and the Orientation attribute to Horizontal):

```
| Columns = 2 |
| Horizontal  |
| 1  | 2  | 3 |
| 4  | 5  |   |
```

Only packed columns are allowed, that is, the number of empty cells allowed in any arrangement of palette or radio fields is always less than half the height of the column. Also, empty cells are always displayed at the end.
For example, the following illustration shows some of the possible configurations for a sixteen-cell palette field:

The five-column display was compressed to four columns because four empty cells are not allowed for a column that is only five values high. Note that the child order is along the major axis.

**Note:** Display is based on actual data as well as the values of the Columns and Orientation attributes. Therefore, when building a palette field dynamically, verify the number of actual cells by modifying the palette field's contents and calling the UpdChoiceList method prior to setting the number of columns.

### BitmapObject Class

The **BitmapObject class** provides attributes and methods that enable you to manipulate the images displayed in an image field. For example, when you create an image field on a form, it is similar to an empty picture frame. To display an image in that picture frame, use the BitmapObject's FileHandle attribute to display an image stored in a file.

Setting a BitmapObject attribute does more than change the image field's value. Each time you change the value, OpenROAD executes the implied procedures. For example, when you set the FileHandle attribute to a new value and the BitmapObject is associated with an image field, OpenROAD displays the new image in the field. This simplifies your coding, because you only need to set the attribute.
The BitmapObject class lets you store images either in standard files or in a database. Using your native file system to store images in standard files provides performance advantages. You can use SQL to store and retrieve the file names in a database table. You can then use the BitmapObject attributes and methods that deal with files to manipulate the images from your application.

The disadvantage of storing images in files is that you are responsible for maintaining them (because the images are stored outside of the system). You must ensure that any files you reference from the application actually exist, and you must back up and update the files appropriately. In addition, if you are creating a portable application or using a network, your program must take into account the different conventions for file specifications.

Storing images in the database enables you to take advantage of the DBMS recovery and transaction facilities. In addition, when you use networks, there are no file name transparency problems. However, this approach also has its disadvantages. When images are stored in a database, it is difficult for other programs to access the images and there is extra overhead in using the database instead of the native file system.

For more information about these two BitmapObject features, see Store Images in a Database (see page 284) and How You Can Store Images in Files (see page 283).

**How You Can Store Images in Files**

You can use your file system to store and retrieve images from standard files. To do this, you must create and maintain the bitmap images in files. OpenROAD reads image files in the following formats:

- **xbm**
  Specifies X standard monochrome bitmap format

- **rast**
  Specifies Sun raster format for color or monochrome bitmaps

- **gif**
  Specifies GIF portable format for color or monochrome bitmaps

- **tiff**
  Specifies color or black and white bitmap (including TIFF FAX Group 3 or 4 format files)

- **cur**
  Specifies CUR format for Windows cursor files
How You Can Work with Images

**bmp**
Specifications BMP format

**ico**
Specifies ICO format for Windows icon files

If you have an image stored in another format, you must use a conversion utility to convert the file into one of these valid formats before you use it with OpenROAD.

**How You Can Store Images in a Database**

To store images in a database, use OpenROAD to add them to a table. OpenROAD automatically creates a special bitmap storage table called ii_stored_bitmaps. However, you can create your own bitmap storage table to improve distribution of the images in the database and avoid transaction bottlenecks. For more information about creating your own bitmap storage table, see How You Can Create a Bitmap Storage Table (see page 285).

Each image in the bitmap storage table is associated with a handle, called a DBHandle, which you use to access the image. For example, to display an image stored in the database, you can set the value of the DBHandle attribute to the DBHandle for that image. OpenROAD then displays the image that is associated with that particular DBHandle. You must store the DBHandles for the images in a separate table from the images themselves.

Use the InsertIntoDB method to add the image to the database table. The InsertIntoDB method adds the image to the table you specify or, if you do not specify a table, to the system table. (To use a bitmap storage table other than the system table, you must set it up as described in How You Can Create a Bitmap Storage Table (see page 285).)

To add the images to the database, you would perform the following basic steps:

1. Define a varchar(76) column to hold the DBHandles in the table that contains the data for your application.
2. Decide whether to store the images in the system table or your own table.
3. Set the FileHandle attribute of the BitmapObject to the file name. OpenROAD copies the image from the specified file and stores the copy in the bitmap object.
   - If there is an image field on your form, you can use the BitmapObject associated with the image field. If there is no image field, declare a variable of type bitmap object to use for loading the images.
4. Use the InsertIntoDB method to insert into the database table the image that is now in the bitmap object. After this method completes, OpenROAD stores the new DBHandle for the image in the DBHandle attribute.
5. Add the current value of the DBHandle attribute to your database table so that you can access the image the next time you need it.

6. Commit the changes to the database.

The image is now stored in the database. You can display the image by setting the DBHandle attribute, and you can replace or delete the image, if necessary, with the UpdateInDB or DeleteFromDB methods, respectively.

For a description of updating images in the database, see Update Bitmaps in a Database (see page 285). For more information about error handling for the InsertIntoDB method, see Working with a Database (see page 147).

**Note:** Do not use these database methods for BitmapObjects with autocommit turned on because the multiple database statements involved in accessing the BitmapObject data can produce incorrect results.

### How You Can Display a Bitmap from a Database

To display an image stored in a database, use the DBHandle attribute of the bitmap object that is associated with the image field. Set the value of BitmapObject’s DBHandle attribute to the value of the DBHandle for the particular image.

### Update Bitmaps in a Database

To remove an image from the database, use the DeleteFromDB method. The DeleteFromDB method deletes the image that has the current value of DBHandle attribute.

**To delete a bitmap from a database**

1. Set the value of the DBHandle attribute to the DBHandle for the image you want to delete.
2. Use the DeleteFromDB method to remove the image.
3. Commit the changes to the database.

The DeleteFromDB method returns ER_OK if successful or a nonzero value if there is an error.

For more information about error handling for the DeleteFromDB method, see Working with a Database (see page 147).

### How You Can Create a Bitmap Storage Table

To store bitmaps in a database, use OpenROAD to add them to a table. OpenROAD automatically creates a special bitmap storage table, called ii_stored_bitmaps. You can also create your own storage table.
If you create your own bitmap storage table, it must be owned by someone with authority to grant user permissions to use it, and it must have the following columns and data types:

**picture_id**
- **Data Type:** i4 not null
- Specifies the ID of the bitmap in the table. This value is a sequential number, starting from 1, that is established when the bitmap is inserted into the database. In the DBHandle for this bitmap, this ID appears in textual form after the table name.

**row_sequence**
- **Data Type:** i4 not null
- Specifies the sequence number for bitmap in picture_id. It is numbered starting from 1.

**text_total**
- **Data Type:** i4 not null
- Specifies the total number of bytes in the full encoding of the image.

**text_value**
- **Data Type:** varchar(1786) not null
- Specifies one 1786-byte piece of encoding string

Create this table as a B-tree structure (compressed or not), with the unique key values on picture_id and row_sequence. Be sure to grant the correct permissions.

**ImageField Object**

The *ImageField object* describes the image field that you use to display the bitmap images. The ImageField object specifies how to display the image.

The most important attribute of the ImageField class is the DisplayPolicy attribute. Because the bitmap image may be a different size than the field, you must specify what happens when the image is displayed. The DisplayPolicy attribute has the following settings:

**DP_CLIP_IMAGE**
- Specifies that OpenROAD clips (or crops) the image to fit it into the field

**DP_AUTO_SIZE_FIELD**
- Specifies that OpenROAD expands or shrinks the field as necessary to fit the image
How You Can Work with Text Strings

The **StringObject class** is an in-memory text string of any length. It provides methods that enable you to manipulate a text by extracting, truncating, and concatenating strings. It also provides the attributes and methods you use for storing and updating the strings, either in standard text files or in the database. The StringObject class makes it easy to build applications that let users create, modify, and store textual data of indeterminate size.

When you create a multiline entry field in OpenROAD Workbench, you can choose between two types of variables, a varchar data type and a StringObject data type. When you select the StringObject data type, the variable associated with the entry field is a reference variable that points to an object of the class StringObject. The StringObject class is also useful for local and global variables, which you can use to store text or pass as parameters between frames or 4GL procedures.

StringObject objects are very effective for storing moderately sized text. However, because all manipulation is done in dynamically allocated memory, using StringObject objects to store large text strings is not very efficient. For text that is always less than 2000 bytes, the varchar data type is the most efficient format.

Setting a StringObject attribute does more than change the attribute's value. Each time you change the value, OpenROAD executes implied procedures. For example, when you set the FileHandle attribute to a new value and the StringObject is associated with an entry field, OpenROAD displays the new string in the field. This process simplifies your coding because all you need to do is set the attribute.

The StringObject class lets you store strings either in standard text files or in a database. Using your native file system to store strings in standard files provides performance advantages. You can use SQL to store and retrieve the file names in a database table.

**DP_SCALE_IMAGE_HW**

Specifies that OpenROAD scales the height and width of the image to fit the field, possibly causing the image to distort.

**DP_SCALE_IMAGE_H**

Specifies that OpenROAD scales the image vertically to fit the field. The aspect ratio of the image is maintained.

**DP_SCALE_IMAGE_W**

Specifies that OpenROAD scales the image horizontally to fit the field. The aspect ratio of the image is maintained.
For example, assume your application uses standard text files to store sections of a document and stores the names of the files in a database table. You can use the StringObject attributes and methods that deal with files to manipulate the strings from your application.

The disadvantage of storing strings outside of the database is that you are responsible for maintaining the string files. You must ensure that any files you reference from the application actually exist, and you must back up and update the files appropriately. In addition, if you create a portable application or use a network, your application must take into account the different conventions for file specifications.

Storing strings in the database offers the following advantages:

- You can take advantage of the DBMS recovery and transaction facilities.
- If you use networks, there are no problems with file name transparency.

Storing strings in the database, however, has the following disadvantages:

- It is difficult for other programs to access the data.

  To alleviate this problem, after loading the text into the database, use the WriteToFile method to transfer the text back out to a system file for manipulation outside of the database.

- There is extra overhead in using the database instead of the native file system.

The following sections describe how to work with strings stored in files and in the database.

**How You Can Store Strings in the Database**

To store strings in a database, use OpenROAD to add them to a table. OpenROAD automatically creates a special string storage table, called ii_stored_strings. However, you can create your own storage table if you want to improve distribution of the strings in the database and avoid transaction bottlenecks.

Each string in the storage table is associated with a handle, called a DBHandle, which you use to access the string. For example, to display a string stored in the database, set the value of the DBHandle attribute to the DBHandle for that string. OpenROAD then displays the string that is associated with that particular DBHandle. You must store the DBHandles for the strings in a separate table from the strings themselves.

The strings that you load into the database can come from text files, from the user (who enters them into multiline entry fields), or from StringObject variables in your 4GL code.
How You Can Load a String into the Database

Use the InsertIntoDB method to add a string to the table you specify or, if you do not specify a table, to the system table.

Note: To use a string storage table other than the system table, you must set it up as described in How You Can Create a String Storage Table (see page 292).

To load a string into the database, you would perform the following basic steps:

1. Define a varchar(76) column to hold the DBHandles in the table that contains the data for the application.
2. Decide whether to store the strings in the system table or your own table.
3. If the text string is in a text file, set the FileHandle attribute of the StringObject to the file name. (If the user has entered the text into an entry field, the string is already stored in the StringObject object and it is not necessary to set the FileHandle attribute.)
   For this step, if there is an entry field on your form, you can use the StringObject object associated with the entry field. If there is no entry field, declare a variable of type StringObject to use for loading the text.
4. Use the InsertIntoDB method to insert the string in the StringObject object into the database table.
   After this method completes, OpenROAD stores the new DBHandle for the string in the DBHandle attribute.
5. Add the current value of the DBHandle attribute to your database table so that you can access the string the next time you need it.
6. Commit the changes to the database.

For more information about error handling for the InsertIntoDB method, see Working with a Database (see page 147).

The string is now stored in the database. You can display the string time by setting the DBHandle attribute, and you can replace or delete the string if necessary with the UpdateInDB and DeleteFromDB methods respectively. These methods are described in How You Can Display a String from a Database (see page 289) and How You Can Update Strings in a Database (see page 290).

How You Can Display a String from a Database

To display a string stored in a database, use the DBHandle attribute of the StringObject object that is associated with the entry field. Set the value of the DBHandle attribute to the value of the DBHandle for the particular string.
To check whether the DBHandle was set successfully, use the ErrorStatus attribute. The ErrorStatus attribute is set to ER_OK if the DBHandle setting is successful or to a nonzero value if an error occurs.

Each time you set the DBHandle attribute, OpenROAD actually loads a copy of the string into the StringObject object associated with the field. If the user edits the text in the entry field, these changes are made to the text string in the StringObject object. To update the corresponding string in the database, use the UpdateInDB method. This method is described in How You Can Update Strings in a Database (see page 290).

### How You Can Update Strings in a Database

There are two ways to update a string in the database:

- Replace the string in the database with the current value of the StringObject object.
- Replace the old string in the database with the contents of an external file.

The following sections describe each of these ways of updating strings.

### How You Can Replace the Current StringObject Value

Replacing the string in the database with the current value of the StringObject object is useful when the user has entered a value into an entry field or changed the value in the entry field, and you want the update to be reflected in the database.

After the user completes the changes, use the UpdateInDB method to remove the old string from the database and add the new string (the current value of the StringObject object) in its place.

To make this change, you would perform the following basic steps:

1. Set the value of the DBHandle attribute to the DBHandle for the string you want to replace.
2. Allow updates by the user.
3. Use the UpdateInDB method to remove the old string from the database and add the new string (the current value of the StringObject object) in its place.
4. Commit the changes to the database.

The UpdateInDB method returns ER_OK if successful or a nonzero value if there is an error.

The new string has the same DBHandle as the old one. For more information about error handling for the UpdateInDB method, see Working with a Database (see page 147).
How You Can Replace the Contents of an External File

You can replace an old string in the database with the contents of an external file. To replace an old string with the contents of an external file, you would perform the following basic steps:

1. Set the value of the DBHandle attribute to the DBHandle for the string you want to replace.
2. Set the FileHandle attribute to the file name that contains the new string. OpenROAD copies the new string into the StringObject object.
3. Use the UpdateInDB method to remove the old string from the database and add the new string (the current value of the StringObject object) in its place.
4. Commit the changes to the database.

How You Can Delete Strings from a Database

To remove a string from the database, use the DeleteFromDB method, which deletes the string that has the current value of the DBHandle attribute.

To remove a string, you would perform the following basic steps:

1. Set the value of the DBHandle attribute to the DBHandle for the string you want to delete.
2. Use the DeleteFromDB method to remove the string.
3. Commit changes to the database.

The DeleteFromDB method returns ER_OK if successful or a nonzero value if there is an error.

For more information about error handling for the DeleteFromDB method, see Working with a Database (see page 147).
How You Can Create a String Storage Table

Generally, you store strings in the database that contains the other data for the application. Someone with authority to grant user permissions must own this storage table, which must have exactly the following columns and data types:

**string_id**

*Data Type: i4 not null*

Specifies the ID of the string in the table. This value is a sequential number, starting from 1, that was established when the string was inserted into the database. In the DBHandle that refers to this string, this ID appears in textual form after the table name.

**row_sequence**

*Data Type: i4 not null*

Specifies the sequence number for text in the string, beginning with 1

**text_total**

*Data Type: i4 not null*

Specifies the total number of bytes in the full text string

**text_value**

If the database is *not* a Unicode-enabled database, then the data type for this column must be:

*Data Type: varchar(1786) not null*

If the database is a Unicode-enabled database, then the data type for this column must be:

*Data Type: nvarchar(893) not null*

Create this table with a B-tree structure (compressed or not) with the unique key values on string_id and row_sequence. Be sure to grant the correct permissions on the table.
How You Can Determine If a Database is Unicode-enabled

For Ingres, a database is Unicode-enabled when you use either the -i or -n option when you create the database. You can determine whether a database is Unicode-enabled by issuing the following query:

```sql
SELECT cap_value
FROM iidbcapabilities
WHERE cap_capability = 'NATIONAL_CHARACTER_SET'
```

If the value of cap_value is 'N', then the database is not Unicode-enabled. If the value of cap_value is 'Y', then the database is Unicode-enabled.

If the database is Unicode-enabled, the system default catalog used to store StringObjects, ii_stored_strings, cannot be used to save and retrieve data. The user always must supply a predefined table as a parameter on the InsertIntoDB method of the StringObject.

**Important!** For Unicode-enabled databases, we strongly recommend using Normalization Form C (NFC) normalization format databases (-i option with the createdb command) because OpenROAD uses NFC internally when coercing between Unicode and non-Unicode data.

You can determine the normalization form in the database by issuing a call to dbmsinfo():

```sql
SELECT dbmsinfo('unicode_normalization')
```

The select statement will return blank if the database does not support Unicode or if the database does not perform normalization. It will return the value 'NFC' if the database supports the NFC normalization form, and returns the value 'NFD' if the database supports the NFD normalization form.

For more information on dbmsinfo(), see the Ingres SQL Reference Guide.
Chapter 10: Using 3GL in Your Application

This section contains the following topics:

- How You Can Call 3GL Procedures (see page 295)
- How You Can Use Exec 4GL Statements in 3GL Procedures (see page 298)
- How You Can Use Exec SQL Statements in 3GL Procedures (see page 305)
- How You Can Preprocess Exec 4GL and Exec SQL Statements (see page 305)
- How You Can Link 3GL Procedures (see page 306)
- Technical Tips (see page 306)
- Sample 3GL Procedures (see page 307)

Most of your application development needs can be met by coding in OpenROAD 4GL. However, there are occasions when you want to make procedure calls to 3GL or embedded SQL. The most common such occasions include:

- Improving performance of numerical calculations
- Using existing 3GL code

Note: All the code examples described in this chapter assume that you are using an Ingres DBMS.

How You Can Call 3GL Procedures

This section describes how to call 3GL procedures from an OpenROAD application.

Callproc Statement

You call a 3GL procedure from an OpenROAD script or 4GL procedure using the callproc statement similarly to the way you call a 4GL procedure. The basic syntax for calling a 3GL procedure is:

\[ \text{return_variable =} \] callproc procedurename;

procedurename

Specifies the name that you specified when you created the 3GL procedure and also used when you registered the 3GL procedure. Use this procedure name and create a component for the procedure in OpenROAD Workbench. This name does not refer to the name of the file in which the procedure is stored but to the procedure itself.

For more information about creating components for 3GL procedures in OpenROAD Workbench, see the User Guide.
In the following example, the callproc statement calls the 3GL procedure named my3glprocedure:

callproc my3glprocedure;

When the procedure returns a value, you can specify a variable in the calling frame or procedure to receive the return value. The return variable must be a simple variable of the same data type as the return value.

In the following example, returnvalue is an integer variable:

returnvalue = callproc my3glprocedure;

**How You Can Pass Parameters to 3GL Procedures**

To pass parameters to a 3GL procedure, you specify a list of expressions. Because you do not include parameter names, you must specify the parameters by position. The syntax for calling a 3GL procedure is:

```
[return_variable =] callproc procedurename
    ([expression | byref(variable)]
    {, expression | byref(variable))}
```

Each expression in the list must correspond to a parameter in the 3GL procedure parameter list. You must verify that each expression matches the data type and position of the corresponding parameter. The expression you pass to the 3GL procedure can be a constant or any legal OpenROAD expression, so long as the resulting data type is compatible with the data type of the corresponding 3GL procedure parameter.

Because 3GL procedures do not allow null values, you cannot pass a simple variable with a value of null. If your variable is nullable in 4GL, you can pass it using the ifnull function. This function helps ensure that a fixed value, instead of a null, is passed when a null is encountered.

For more information about the ifnull function, see the Language Reference Guide online help.

**Examples—Passing Parameters to a 3GL Procedure:**

The following example calls a C procedure, my3glprocedure. This procedure's three parameters are simple data types: two integers and a varchar. The second parameter references intvar, a 4GL integer variable, and the third references a 4GL varchar variable called video.title:

```
callproc my3glprocedure(256, 3 + intvar, 'Free movie this week is ' + video.title);
```
The `my3glprocedure` procedure declared corresponding parameters as follows:

```c
my3glprocedure (var1, var2, var3)
    int var1, var2;
    char *var3;
{
    ...
}
```

For information about matching data types between 3GL procedures and OpenROAD variables, see the *Language Reference Guide* online help.

**Note:** Because the parameters must match exactly in position and type, you must specify all parameters to a 3GL procedure. The OpenROAD runtime system does not check errors in parameter passing to 3GL procedures; such errors can result in abnormally terminated programs. If you are using OpenROAD Workbench, it could abort without saving your changes.

As with 4GL procedures, OpenROAD lets you pass parameters to 3GL procedures by reference. The following example shows two parameters passed by reference and one passed by value:

```c
callproc my3glprocedure(byref(floatvar), 36,
    byref(textvar));
```

To help ensure full portability across all systems, pass all floating point parameters to C procedures with the byref qualifier, even if you do not want to change the value of the data in the C procedure. For example, the following statement calls the `scale_y_array` procedure, passing it two floating point parameters (vmin and vmax) by reference:

```c
callproc scale_y_array(byref(vmin), byref(vmax),
    ymax - (labelheight*2), 0, sales_array);
```

**Important!** There are limits to the number of parameters you can pass to a 3GL procedure. The specific limit depends on the platform, the language in which the procedure is written, and the data types of the parameters. If you want your application to be portable across all platforms, do not pass more than 39 parameters to a 3GL procedure.

If you pass floating point parameters to C procedures without the byref qualifier, the limit may be smaller. Floats passed by value require an 8-byte parameter for a copy of the float, whereas floats passed by reference require only a 4-byte parameter for the address of the float (on 32-bit platforms).
How You Can Use Exec 4GL Statements in 3GL Procedures

A limited number of scalar data items can be passed easily between 4GL and 3GL procedures. However, if the amount of scalar data to be passed exceeds the number of parameters allowed on your system, or if you want to pass structured data such as an array or an OpenROAD object, use the C language to pass the data to an embedded SQL procedure.

To access OpenROAD structured data from 3GL, you use a set of embedded statements that have the following syntax:

`exec 4gl 4gl_statement [terminator]`

**Note:** This syntax is similar to that used for embedded SQL statements. All 3GL routines using these `exec 4gl` statements can be used only in OpenROAD applications.

For more information about using `exec 4gl` statements, see the *Language Reference Guide* online help.

You can use an embedded procedure with OpenROAD to perform the following functions:

- Getting and setting object and array attributes
- Dynamically accessing object and array attributes by a describe operation using the SQLDA
- Indexing into arrays to get or set objects
- Inserting rows into, deleting rows from, and removing rows from arrays
- Clearing arrays
- Accessing global variables and constants
- Sending user events to 4GL

**How You Can Pass Structured Data**

Structured data is passed to 3GL as a handle. You reference the handle in the `exec 4gl` statements whenever you want to reference the object or array. The handle is passed as a 4-byte integer. To store a handle passed in from 4GL, use a 32-bit integer data type.

After control is returned to 4GL, a later call from 4GL to 3GL cannot use the handle stored earlier in a static variable because the earlier handle is no longer valid. Therefore, the 3GL routine should not store a handle in a static variable.
How You Can Check for Errors

Use the inquire_4gl statement to check for errors resulting from exec 4GL statements. By default, any such errors are directed to the trace window and the log file. To turn off the display of these errors, use the set_4gl statement with the messages keyword.

Neither the global error handler that set_sql defines nor the sql whenever statement apply to exec 4GL errors. An error handler specified with the IIseterr function is not called when an error occurs.

Summary of Exec 4GL Statements

The following table displays the operations with their associated statements that are available in 3GL when you pass objects and arrays from 4GL. You must always place the exec 4GL keywords before each statement.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access global variables and</td>
<td>get</td>
</tr>
<tr>
<td>constants</td>
<td>get global constant</td>
</tr>
<tr>
<td>Clear an array</td>
<td>clear array</td>
</tr>
<tr>
<td>Get attribute(s) of an object</td>
<td>get attribute</td>
</tr>
<tr>
<td>Get array information</td>
<td>inquire_4GL(var = allrows(array))</td>
</tr>
<tr>
<td></td>
<td>inquire_4GL(var = firstrow(array))</td>
</tr>
<tr>
<td></td>
<td>inquire_4GL(var = lastrow(array))</td>
</tr>
<tr>
<td>Get error status or text from</td>
<td>inquire_4GL(var = errno)</td>
</tr>
<tr>
<td>previous operation</td>
<td>inquire_4GL(var = errortext)</td>
</tr>
<tr>
<td>Get list of attributes of a class</td>
<td>describe object into descriptor</td>
</tr>
<tr>
<td>Get member of an array</td>
<td>getrow</td>
</tr>
<tr>
<td>Get object information</td>
<td>inquire_4GL(var = IsArray(object))</td>
</tr>
<tr>
<td></td>
<td>inquire_4GL(var = Classname(object))</td>
</tr>
<tr>
<td>Insert row into an array</td>
<td>insertrow</td>
</tr>
<tr>
<td>Remove row from an array</td>
<td>removerow</td>
</tr>
<tr>
<td>Send a UserEvent to an OpenROAD</td>
<td>send userevent</td>
</tr>
<tr>
<td>frame</td>
<td></td>
</tr>
<tr>
<td>Set attribute(s) of an object</td>
<td>set attribute</td>
</tr>
<tr>
<td>Set error display status</td>
<td>set_4gl</td>
</tr>
</tbody>
</table>
**Operation Statement**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set member of an array</td>
<td>setrow</td>
</tr>
<tr>
<td>Set row in an array as deleted</td>
<td>setrow deleted</td>
</tr>
</tbody>
</table>

For the complete syntax of exec 4GL statements, see the *Language Reference Guide*.

**Example—Passing an Array to a 3GL Procedure**

Assume, for example, a frame that displays a line chart constructed dynamically from values in a database table. This frame calls a 4GL procedure named `scale_y_array` to calculate an array of Y-axis values. The `scale_y_array` procedure can be written in embedded SQL to enhance performance when calculating many graph points.

The following sections describe a 3GL version of the `scale_y_array` procedure.

**How You Can Call the 3GL Procedure**

When the frame calls the `scale_y_array` procedure, the frame passes an array (sales_array) whose data type is the saleschart_data user class. Two of the attributes in this class are significant to the 3GL procedure:

**sales_value (float not null)**

Contains the dollar sales for a given quarter of the fiscal year quarter (quarter)

**y (integer not null)**

Contains the Y-axis coordinate for each quarter to be displayed, after completion of the procedure.

The frame also passes simple variables to the 3GL procedure. These variables are used to calculate the scale factor used to plot each point on the graph.

The line in the script that calls the `scale_y_array` procedure would be replaced by the following code if `scale_y_array` were an embedded SQL procedure:

```c
/* Pass highest and lowest dollar amounts by */
** reference to ensure portability when passing **
** floating point parameters to C procedures. */
vmax = 1000.0;
vmin = 0.0;
callproc scale_y_array(byref(vmin), byref(vmax),
                      ymax - (labelheight*2), 0, sales_array);
```
The frame passes several parameters:

**vmax**

Specifies the highest dollar amount to be displayed on the chart (represents the top of the chart)

**vmin**

Specifies the lowest dollar amount to be displayed on the chart (represents the bottom of the chart)

**labelheight**

Specifies the amount of space used by labels on the dynamically created chart; used to calculate the window position of the lowest Y point

**sales_array**

Specifies an array that contains the sales dollar amounts and contains the Y axis values

The vmax and vmin values are passed as variables, rather than directly as values, to ensure portability. For a discussion of passing floating point parameters to C procedures, see How You Can Pass Parameters to 3GL Procedures (see page 296).

---

### Scale_y_array Procedure

This procedure declares the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Source in Calling Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_handle</td>
<td>Handle for the 4GL array containing values of data to scale</td>
<td><em>sales_array</em> object</td>
</tr>
<tr>
<td>low_value</td>
<td>Lowest value of data</td>
<td>vmin</td>
</tr>
<tr>
<td>high_value</td>
<td>Highest value of data</td>
<td>vmax</td>
</tr>
<tr>
<td>low_y_point</td>
<td>Lowest y point</td>
<td>0</td>
</tr>
<tr>
<td>high_y_point</td>
<td>Highest y point</td>
<td>ymax - (labelheight*2)</td>
</tr>
</tbody>
</table>

On an OpenROAD window, the point of origin is the upper left corner, which has the X,Y coordinates of 0,0. The X and Y coordinates increase as you move to the right and down the window. Consequently, the value for low_y_point is always greater than the value of high_y_point.
The 3GL procedure also declares the following local variables:

**scale_factor**
- Used to calculate each Y-axis position

**nrows**
- Specifies the number of rows in the array; used in the while loop and returned to the calling frame

**i**
- Specifies the index for the while loop

**row_handle**
- Specifies a handle for a row in the array

The following code is the declaration section of the 3GL procedure:

```c
int scale_y_array (low_value, high_value,
low_y_point, high_y_point, array_handle)
double *low_value, *high_value;
int low_y_point, high_y_point;
exec sql begin declare section;
long array_handle;
exec sql end declare section;
{
exec sql begin declare section;
double scale_factor, value;
int nrows; /* number of rows in array */
int i; /* index for while loop */
long row_handle;
exec sql end declare section;
}
```

**Note:** Variables used in exec 4GL or exec SQL statements must be:
- Declared in the following statements:
  - exec SQL begin declare section
  - exec SQL end declare section
- Preceded by a colon when used in a statement

To determine the scale factor, this procedure performs the following calculation:

```c
/* Calculate scale factor one time only */
scale_factor = (low_y_point-high_y_point)
/(*high_value-*low_value);
```
The procedure uses the inquire_4gl statement to determine the number of rows in the array. It then uses this number as the maximum iteration of a while loop, for example:

/* Find out how many rows in the array */
exec 4gl inquire_4gl (:nrows = lastrow(:array_handle));

The while loop gets a handle for each row in the array. This loop uses the handle to get the value in that row's sales_value attribute. The value obtained is subtracted from the highest dollar value and multiplied by the scale value to determine the Y value for that row. Finally, the loop sets the value of the y attribute in the appropriate row of sales_array.

The following code shows how the while loop processes each row in the array to determine and set its Y value:

/* Loop through sales values in the array and ** put scaled values into y. */
i = 0;
while (i++ < nrows)
{
    exec 4gl getrow :array_handle :i
    (:row_handle = ROW);
    exec 4gl get attribute :row_handle
    (:value = sales_value);
    value = scale_factor * (*high_value - value);
    exec 4gl set attribute :row_handle (y = :value);
}

The following statement, the last in the 3GL procedure, returns control to the calling frame:

return;

Example—Sending a User Event from 3GL to 4GL

A 3GL procedure can send user events to 4GL frames and procedures by using exec 4GL statements. However, because the user event is not received by the 4GL until control returns from the 3GL, user events are not as common a form of communication between 3GL and 4GL as they are between concurrently open 4GL frames.

Sending user events from 3GL to 4GL is useful when the 4GL calls the 3GL regularly to communicate with the outside world. For example, an application might use a ghost frame to call a 3GL procedure every minute by sending itself delayed user events. The 3GL procedure monitors outside conditions and sends the ghost frame a user event when it encounters significant conditions. Because the 3GL can monitor several conditions, it can send multiple user events to the ghost frame.
Assume, for example, that a ghost frame defines a user event called Monitor_conditions and that it sends this event to itself every sixty seconds. In addition to sending the event to itself, this user event calls a 3GL procedure that communicates with external devices. In calling the 3GL procedure, the ghost frame passes its FrameExec object, allowing the 3GL procedure to send a user event to the ghost frame in the event of a triggering condition.

The following code in the ghost frame performs these operations:

```plaintext
on userevent 'Monitor_conditions' =
begin
    callproc monitor_proc(CurFrame);
    CurFrame.SendUserEvent
        (eventname = 'Monitor_conditions',
         delay = 60);
end;
```

The monitor_proc procedure that the ghost frame calls must declare a parameter to store the handle to the FrameExec object passed to it. The following code contains an example 3GL procedure:

```plaintext
int monitor_proc (frame_handle)
exec sql begin declare section;
    long frame_handle;
exec sql end declare section;
{
    exec sql begin declare section;
        char msg[100];
    exec sql end declare section;
...
    strcpy (msg, "Appropriate condition text here.");
    exec 4gl send userevent :frame_
        (eventname = 'Monitor_conditions',
         messagevarchar = :msg);
...
return;
}
```

**Note:** The use of single quotes within an exec 4GL statement follows the syntactic rules of OpenROAD rather than those of the 3GL language.

This procedure uses the frame_handle variable, which received the ghost frame's FrameExec object, to serve as the address when sending the user event to the ghost frame.

For a discussion of communicating between frames with user events, see Inter-Frame Communication Techniques (see page 333). For more information about ghost frames, see Programming Frames (see page 95).
How You Can Use Exec SQL Statements in 3GL Procedures

You can also access the database from your 3GL procedure by using embedded SQL statements.

The current database connection of your 4GL application becomes the current embedded SQL connection when your 3GL procedure is called. You can also use embedded SQL statements to establish new database sessions within a 3GL procedure. But when the procedure completes and control returns to OpenROAD, the application automatically returns to the session in which it was running when the procedure was called.

Take care to ensure that your embedded SQL 3GL procedure cooperates with your 4GL application on their shared session. Your 3GL can impact the shared session in ways that could adversely affect your 4GL application (such as disconnecting the session, or changing the transaction state).

How You Can Preprocess Exec 4GL and Exec SQL Statements

If your 3GL procedure contains exec 4GL or exec SQL statements, that source must be preprocessed into regular C code before it can be compiled. This is done through the esqlc preprocessor.

By default, the embedded source should have a file extension of .sc, and is preprocessed into a .c file.

The preprocessor command has the form:

```
esqlc [flags] yourfile.sc
```

Note: Delete the -prototypes flag from esqlc commands.

On all platforms, your source file should include the eqdefc.h header file, which contains prototypes of the function calls generated by the esqlc preprocessor.

Windows

The eqdefc.h file is located in the %II_SYSTEM%\inges\files directory.

UNIX

The eqdefc.h file is located in the $II_SYSTEM/inges/files directory.
How You Can Link 3GL Procedures

After coding your 3GL procedures, you must compile and link them into an execution module appropriate for your platform. On Windows platforms, the execution module takes the form of dynamic link libraries (DLL). On UNIX platforms, it takes the form of shared libraries. OpenROAD makes a call into the execution module when a 3GL procedure is called in 4GL code.

Note: Use esqlc.lib instead of ingres.lib when linking.

For instructions about how to build these libraries, consult the documentation supplied for your compiler and linker for the specific operating system. Also consult the readme_OR_3GLsample.txt file in the supplied 3GL example for detailed information:

Windows
The readme file is located in the %II_SYSTEM%\ingres\w4glsamp\3gl directory.

UNIX
The readme file is located in the $II_SYSTEM/ingres/w4glsamp/3gl directory.

Technical Tips

The following sections discuss several important issues associated with building DLLs in general and 3GL DLLs for Windows.

4GL/3GL Data Types

The following table shows the types for 4GL and the corresponding 3GL types:

<table>
<thead>
<tr>
<th>4GL</th>
<th>3GL on All Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>char*</td>
</tr>
<tr>
<td>char</td>
<td>char*</td>
</tr>
<tr>
<td>date</td>
<td>char*</td>
</tr>
<tr>
<td>money</td>
<td>double</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>smallint</td>
<td>int (4 bytes)</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>
**Note:** Addresses returned to OpenROAD from a 3GL procedure should be global or static so that they are not out of scope when used in the 4GL.

## Sample 3GL Procedures

Your OpenROAD development installation contains the following two sample 3GL procedures:

- A 3GL sample that only contains 3GL statements
- An exec 4GL and exec SQL sample that includes 4GL and exec SQL statements

### 3GL Sample

**Windows**

The 3GL sample is located in the `%II_SYSTEM%\ingres\w4glsamp\3gl` directory.

**UNIX**

The 3GL sample is located in the `$II_SYSTEM/ingres/w4glsamp/3gl` directory.

Consult the readme_OR_3GLsample.txt file for instructions about building the sample.

When the sample is running, you can invoke all functions by clicking the Call All 3GL Procedures button. Or, you can invoke individual functions by selecting the associated radio fields. A message box displays to the user whether the functions were called successfully. This determination is based on checking both the return and argument values after the call to ensure that they are correct. This capability is built into the sample and does not require you to do any coding.
Exec 4GL and Exec SQL Sample

Windows

The exec 4GL and exec SQL sample is located in the %II_SYSTEM%\ingres\w4glsamp\esqlc directory.

UNIX

The exec 4GL and exec SQL sample is located in the $II_SYSTEM/ingres/w4glsamp/esqlc directory.

Consult the readme_OR_4GLsample.txt file for instructions about building the sample.
Chapter 11: Data Entry Error Handling

This section contains the following topics:

How You Can Use Data Entry Error Handlers (see page 309)

This chapter describes data entry error handlers and how to use them to customize error messages. It also provides examples of different types of data entry error handlers.

How You Can Use Data Entry Error Handlers

If a user types invalid data into an entry field (for example, typing text into a field that has a numeric data type), an error message appears in a pop-up by default. Because these messages often do not communicate meaningfully to the user, you may want to create a data entry error handler.

A data entry error handler is a global or local procedure written in 4GL that gains control as a result of a user typing invalid data into an entry field. The error condition is detected when the entry field loses input focus, or the GetFieldValue method is executed on the entry field.

In the case of the entry field losing input focus, the following control situations can occur:

- If the data entry error handler was defined for a frame, it gains control when invalid data is detected in any field in either:
  - The frame for which the handler was defined
  - Any frame called or opened (directly or indirectly) by that frame
- If the data entry error handler was defined for an executable object other than a frame (a method or procedure), it gains control when invalid data is detected in any field in any frame called or opened (directly or indirectly) by that executable object.

In the case of the GetFieldValue method being executed on the entry field, the data entry error handler gains control when the GetFieldValue method was invoked either of the following ways:

- The executable object for which the error handler was defined
- Any executable called or opened (directly or indirectly) by that executable object
Defining a 4GL procedure to be the data entry error handler for an executable object requires setting the DataEntryErrorHandler attribute of the executable object to the handle of the error-processing 4GL procedure. Set this attribute for the current frame using the following syntax:

```
CurFrame.DataEntryErrorHandler = 
   CurFrame.Scope.GetProcHandle (name = 'string');
```

This syntax invokes the GetProcHandle method of the Scope system class to perform the following operations:

- Get the handle for the named procedure in the named scope
- Load the procedure's handle into the DataEntryErrorHandler attribute of the currently active executable object

To set the DataEntryErrorHandler attribute for the current procedure or method, use the CurProcedure and CurMethod variables rather than CurFrame.

To enable you to modify the error message, correct the value entered by the user, and locate the frame containing the error-triggering field, data entry error handlers are passed the following parameters:

**ErrorMessage**

**Data Type:** Varchar(2000) not null

Specifies either a default error message provided by OpenROAD or a user-defined error message provided by a lower handler. It is passed by reference.

**ErrorField**

**Data Type:** EntryField

Specifies the error-triggering field. The attributes ErrorField.TextValue and ErrorField.StringValue contain the erroneous text that the user typed into the triggering entry field.

**ErrorFrame**

**Data Type:** FrameExec

Specifies the reference variable pointing to the frame containing the error-triggering field.

The first eight characters of the ErrorMessage parameter contain the error number of the triggering data entry error. Currently, the following are the data entry errors:

- **E_PW0007**
  - Indicates a mandatory field error
- **E_PW0008**
  - Indicates a format error
How You Can Use Data Entry Error Handlers

**E_PW0013**
Indicates a data type error

**E_PW0027**
Indicates an input-masking mandatory position error (mandatory positions of the field are not filled in)

The following sections explain how to create data entry error handlers.

**How You Can Change Values with a Data Entry Error Handler**

A data entry error handler can change almost any data it can access. For example, it can:

- Change the error message to be displayed in a pop-up
  
  Setting the error message to an empty string prevents display of the pop-up.

- Reset the field's value
  
  Resetting the value removes the error condition (unless you attempt to set a mandatory field's value to an empty value), but does not remove the error message. The handler generally removes the error message by changing it to an empty string. However, the handler may replace the message with a user-defined warning message.

  **Note:** Resetting the error field's value in a handler does not clear any pending SetValue events for the field.

**How You Can Access a Hierarchy of Data Entry Error Handlers**

OpenROAD lets a called or opened executable object override the handler specified by its parent by setting its own data entry error handler. The locally invoked error handler can:

- Bypass error handlers defined by its parent

- Invoke the parent's error handler after executing its own code

When a data entry error handler returns control, the error handler can cause OpenROAD to search parent executable objects for other data entry error handlers defined at higher levels. OpenROAD calls each handler it finds, passing to it all modifications made by previous handlers. If no handler is found, execution resumes as if no search for other handlers was performed.
The integer value returned by a data entry error handler determines whether OpenROAD searches for other error handlers defined by parents. There are two integer values that error handlers return:

**EH_RESUME**

Causes normal execution to be resumed, displaying non-blank messages in a pop-up; the message can be revised by the code.

**EH_NEXT_HANDLER**

Causes the search for handlers to be resumed, starting with the parent of the executable object in which the current handler was found.

If none of the handlers removes the error condition by resetting the field's value, the field triggering the error condition retains input focus and the cursor remains on the error field.

If one of the handlers removes an error condition that was detected when the field attempted to give up input focus, the input focus is changed. Events (such as SetValue) are generated as if the user had entered valid data.

Error handlers that remove an error condition should not return EH_NEXT_HANDLER.

The search for other handlers defined by parent executable objects stops when either no handler is found, or the current handler returns EH_RESUME.

**How You Can Customize Error Messages**

The top frame (main_control) of the Videos application defines a local error-handling procedure that it establishes as a data entry error handler. Because all frames in the application are called from the main_control frame, every frame in the application calls this handlerproc procedure implicitly whenever users type invalid values into entry fields.
The following code is the entire error-handling procedure defined in the main_control frame:

```sql
procedure handlerproc(     
    errorfield = entryfield;  
    errorframe = frameexec;  
    errormessage = varchar(2000)  
    not null) /* b yref() */=
begin
    CurProcedure.beep();
    if errorfield.DataType = 'date' then
        errormessage = 'You must enter a valid date in this field';
    else
        errormessage = '';  
    endif;
    return EH_RESUME;
end;
```

Typing incorrect data in any entry field on any frame called by the main_control frame causes the monitor to beep.

When invalid data (such as an integer or Feb 31) is typed into a field with a date data type, the handlerproc procedure displays a pop-up message specific to date fields. Similar messages can be customized for other fields, but this procedure only beeps when invalid data is entered into a field other than a date field. No error message is displayed for other fields because the error message is changed to an empty string.

Because the handlerproc procedure returns EH_RESUME, control returns to the calling frame without searching for other data entry error handlers.

The following code from the main_control frame declares handlerproc as a procedure and establishes it to handle data entry errors:

```sql
initialize =
    declare
        handlerproc = procedure returning integer  
        not null: /* local procedure */
    enddeclare
begin
    CurFrame.DataEntryErrorHandler =  
        CurFrame.Scope.GetProcHandle  
        (name = 'handlerproc');
...  
end;
```

**How You Can Pass Control to Another Error Handler**

The another_handlerproc procedure returns EH_NEXT_HANDLER, causing OpenROAD to search the parents of this frame for other data entry error handlers. Because the frame that defines the handlerproc procedure is the parent frame, control passes to the handlerproc procedure.
The following is the complete code for the another_handlerproc procedure:

```sql
procedure another_handlerproc(
    errorfield = entryfield;
    errorframe = frameexec;
    errormessage = varchar(2000) not null) =

declare
    str = varchar(100) not null;
enddeclare

begin
    message 'Error in field' + errorfield.Name + HC_NEWLINE +
    errormessage;
    return EH_NEXT_HANDLER;
end;
```

Although this procedure uses the message statement to compose the error information, the text of the message is written to the w4gl.log file rather than displayed in a pop-up frame. Statements that normally cause display of a pop-up frame (such as the message statement and the ConfirmPopup method) perform differently when executed from within a data entry error handler. For a discussion of other restrictions, see Restrictions on Data Entry Error Handlers (see page 315).

The following statement from the frame's initialize block establishes the another_handlerproc procedure as its error handler:

```sql
CurFrame.DataEntryErrorHandler =
    CurFrame.Scope.GetProcHandle(name = 'another_handlerproc');
```

**Example—How Data Entry Error Handler Hierarchy Works**

Because this frame is called by the parent frame that defined the handlerproc procedure, the following process occurs if a user enters an erroneous value in an entry field in the child frame:

1. The another_handlerproc procedure is called and it writes the name of the error-triggering field, as well as the standard error message, into the log file or trace window.
2. The handlerproc procedure is called (because the another_handlerproc procedure returns EH_NEXT_HANDLER), and it does the following:
   - Beeps
   - Displays a pop-up message only if the data type of the error field is date
   - Returns control to the child frame (because it returns EH_RESUME)
Restrictions on Data Entry Error Handlers

A data entry error handler is subject to the following restrictions that are imposed on the handler and on any procedures or methods it calls:

- Handlers cannot invoke a frame; the callframe, gotoframe, and openframe statements are all illegal.
- Handlers cannot wait for user events or database events, but they can issue database queries and do file I/O.
- The OpenROAD Debugger does not stop at breakpoints or for any other condition during the execution of the handler.

  Statements that usually cause display of a pop-up frame (when executed outside a data entry error handler), such as the message statement and the ConfirmPopup method, are written to the trace window and the w4gl.log file.

  Except for message statements, such statements cause the entire OpenROAD process to wait for a response, even if there are other threads that could run.

A handler can be in a thread different from that of the frame that encountered the error, even if the handler is a local procedure.

**Note:** The error handlers described in this section do not apply to database errors. For more information about handling database errors, see Working with a Database (see page 147).
Chapter 12: Managing Event Queues

This section contains the following topics:

Conceptual Background (see page 317)
Event-based Programming (see page 318)

OpenROAD uses event queues to manage the order in which events are processed. Event queues can apply to the entire application or to individual frames. This chapter discusses:

- Event queues in the context of event-based programming and the three basic event types:
  - Frame
  - Field
  - Menu
- Interrupting event queues that are programmed in event blocks
- Obtaining information about a current event

For a discussion of writing event blocks, see Writing Scripts and Procedures (see page 67).

Conceptual Background

Although the user perceives the application as an environment in which several tasks can be performed concurrently, events in an OpenROAD application are executed one at a time. The sequential ordering of tasks is handled by the event queue that is maintained for an entire application, as well as event queues for each frame.
### How Events Are Executed

Event code is not necessarily executed immediately after the event is triggered. The process for executing events is:

1. When the event is triggered, OpenROAD sends the event code to the bottom of the application event queue.

2. When the event reaches the top of this event queue, OpenROAD dispatches the event to the bottom of the appropriate frame's event queue. If the event is a user event from an external application, it may go to more than one frame.

3. The event code block is executed when the event reaches the top of the frame's event queue.

Although actions may appear to the user as concurrent, you must consider the exact order in which various events are triggered. When you understand how the events work, you can use OpenROAD to communicate between frames for synchronization and to ensure that the user can perform tasks concurrently.

### Event-based Programming

An important concept to keep in mind when planning your OpenROAD scripts is that users' actions, rather than the order of your code, control the flow of the application.

In most cases, the code you provide for a particular event is executed only when the user triggers that event from the user interface. Depending on how the user interacts with the application, the order in which the events are triggered may differ and some events may not be triggered at all.
The following diagram illustrates the order in which the events are placed into the application event queue if the user clicks Button A, Button B, Button C, and Button D, in that order, from three different frames:

In this example, the events execute in the same order in which they are placed on the application event queue. However, events can execute in a different order when external or user events are being sent. When a frame is blocked (for example, by a WaitFor), external and user events are still delivered. They are not executed until the event that the frame is waiting for arrives. Events that occur later in time for unblocked frames execute before the earlier events of the blocked frame.

You should not rely on the sequence of events as a means of simulating the effect of a local procedure. To avoid duplication of code, use a local procedure instead of sending a user event to the current frame.

**Order of Events: Event Queues**

OpenROAD processes a separate queue for each active frame. If there are two or more concurrently active frames, there is no guarantee as to the order in which the frame queues are processed. Therefore, if you send a user event to more than one active frame, you cannot assume that the order of sending is the order of processing.

Within a queue, events are processed on a First In First Out (FIFO) basis. When OpenROAD processes the frame queue, it finishes executing the current event block before executing the next event in the queue.
By default, when the event is being executed, all open frames are blocked so that the user cannot select ahead and trigger more events. However, you can change this behavior by setting the BlocksFrames attribute to FALSE. For more information about the BlocksFrames attribute, see How You Can Block an Open Frame (see page 106).

Example:

The following sample event block from the video_detail frame includes the SendUserEvent method, which triggers the specified user event. When OpenROAD executes this method, the following happens:

1. It sends the UpdateTitle user event to the application event queue and finishes executing the SetValue event block.
2. When the UpdateTitle event reaches the top of the application event queue, it places this event on the frame queue for the graphic_frame frame.
3. It executes the code associated with the UpdateTitle user event only after that event reaches the top of the frame queue.

```plaintext
on setValue video.title =
begin
  CurFrame.WindowTitle = video.title;
  if graphic_frame is not null then
    graphic_frame.SendUserEvent (eventname = 'UpdateTitle', messagevarchar = video.title);
  endif;
end
```

To force the current event block to wait until a specified user event has arrived, you can use the WaitFor method (for more information about user events and the WaitFor method, see Inter-Frame Communication Techniques (see page 333) and WaitFor Method (see page 345)).

Note: Using the WaitFor method to wait for events sent to the current frame causes the wait to last forever.

How You Can Examine and Manipulate Event Queues

When you are debugging an application, you can manipulate an event before it occurs. The OpenROAD Debugger lets you set break conditions on events, causing the application to pass control to the Debugger immediately before executing the specified event. For more information, see Debugging Your Application (see page 441).

You can also examine the sequence of events in the Debugger's Event Queue window. This window displays the contents of a given frame's event queue. For more information about debugging events and the event queue, see Debugging Your Application (see page 441).
Event Types

The Language Reference Guide online help describes the events you can include in your application. These events can be grouped into the following basic types: frame events, field events, and menu events. Each event type is described in the following sections.

Frame Events

Frame events include events that are triggered by user interactions with the frame's window, such as the WindowResized event that is triggered when the user resizes the window. Frame events also include interactions with the background of the frame, such as the Details event, which is triggered for the frame when the user clicks the Details button on the area surrounding the fields.

The following sample script shows the event block for the WindowIcon frame event. This event is triggered whenever the user iconifies the frame. The code in Step 2 prevents the child, graphic_frame, from remaining displayed if the parent frame is iconified. If a user iconifies the parent frame, this event does the following:

1. Verifies that graphic_frame is displayed.
2. Sets the WindowVisibility of that frame to iconify.

```plaintext
on windowicon =
begin
    if graphic_frame is not null then
        graphic_frame.WindowVisibility = WV_ICON;
    endif;
end
```

A frame also receives child events when the user interacts with the fields on the form. Because the form associated with a frame is considered a single subform containing all non-menu fields on the frame, events triggered in any field also trigger a child event for the frame. This enables the frame to provide generic code that is triggered when any of the fields on its form receive a particular event.
For example, a frame receives a ChildEntry event when any of its child fields receives an Entry event. The following ChildEntry event block changes the color of the field that the user has entered to pale cyan. This code uses the TriggerField attribute of the FrameExec system class to determine which entry field triggered the current event:

```plaintext
on childentry =
begin
  if CurFrame.TriggerField.ISA(class = EntryField) = TRUE then
    EntryField(CurFrame.TriggerField).BgColor=CC_PALE_CYAN;
  endif;
end
```

WindowClose and Terminate are two other important frame events:

**WindowClose**

Is triggered when the user selects the window manager close operation for the running frame.

**Terminate**

Is triggered in any of the following circumstances:

- When the frame is closed by the Terminate method
- When the frame's parent frame closes itself
- When the frame's parent is itself closed by another frame or procedure
- When the user selects the window manager Close operation for the running frame

You can use an event block for either the WindowClose or Terminate events to clean up a frame (for example, to close open transactions) prior to closing it. To keep the window open under specified circumstances, use the WindowClose event. For more information about keeping a window open after a WindowClose event has been triggered, see How You Can Interrupt an Event Block (see page 328).

The UserEvent event, which is a user-defined event used primarily for communicating between frames and with external programs, is also considered a frame event. For more information about this event, see Inter-Frame Communication Techniques (see page 333).

For more information about OpenROAD events in general, see the *Language Reference Guide* online help.
Field Events

Because users mostly interact with individual fields on a form, the events for fields are the most diverse and complex. For example, the following table summarizes the events that a user can trigger for a button field:

<table>
<thead>
<tr>
<th>Event</th>
<th>How Triggered by User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>User clicks the field with the Select mouse button when the field has a bias of FB_LANDABLE or FB_CHANGEABLE.</td>
</tr>
<tr>
<td>DoubleClick</td>
<td>User double-clicks the field with the Select mouse button. This event is not available for fields with biases of FB_VISIBLE, FB_DIMMED, and FB_INVISIBLE.</td>
</tr>
<tr>
<td>ClickPoint</td>
<td>User clicks a field that has a bias of FB_CLICKPOINT.</td>
</tr>
<tr>
<td>Details</td>
<td>User clicks the field with the Details button. This event is not available for fields with biases of FB_VISIBLE, FB_DIMMED, and FB_INVISIBLE.</td>
</tr>
<tr>
<td>DragBox</td>
<td>User draws a drag box in a field that has a bias of FB_DRAGBOX.</td>
</tr>
<tr>
<td>DragSegment</td>
<td>User draws a drag segment in a field that has a bias of FB_DRAGSEGMENT.</td>
</tr>
<tr>
<td>Moved</td>
<td>User moves a field with a bias of FB_MOVEABLE or FB_FLEXIBLE.</td>
</tr>
<tr>
<td>Properties</td>
<td>User clicks a field with the Properties button. This event is not available for fields with biases of FB_VISIBLE, FB_DIMMED, and FB_INVISIBLE.</td>
</tr>
<tr>
<td>Resized</td>
<td>User resizes a field that has a bias of FB_RESIZEABLE or FB_FLEXIBLE.</td>
</tr>
<tr>
<td>Select</td>
<td>User selects a field that has a bias of FB_FLEXIBLE, FB_RESIZABLE, FB_MOVEABLE, or FB_MARKABLE.</td>
</tr>
<tr>
<td>UnSelect</td>
<td>User deselects a field that has a bias of FB_FLEXIBLE, FB_RESIZABLE, FB_MOVEABLE, or FB_MARKABLE.</td>
</tr>
</tbody>
</table>

For a complete list of the events for each of the OpenROAD fields, see the Language Reference Guide online help.

Many events are triggered only when a field is set to a certain bias, allowing you to provide code that is executed only when the user is actually performing a certain task. For example, the Click event can be triggered only when the user is actually allowed to click the button, as determined by the allowable biases. For more information about field biases, see the User Guide.
Event Chains

A simple user action often triggers a series of events called an event chain. An event chain is a linked series of events triggered by a starting event. For example, each time the user changes some data in an entry field and then moves to another field, three events are triggered, as shown in the following illustration:

![Event Queue Diagram]

Another important event chain occurs when the user selects a field. Each time the user selects a field, the following three events are triggered:

- UnSelect event for previous field
- Select event for current field
- SelectionChanged event for the frame

Event chains do not include user events, except when you issue the SendUserEvent method with a focusbehavior parameter setting of FT_SETVALUE or FT_TAKEFOCUS. In these instances, if the value of the current field has changed, the event chain begins with a SetValue event and includes the user event.

**Important!** If you provide more than one event block for an individual field, you must consider the order in which the events are triggered.

Child Events

When a field belongs to a composite field, a single user action on the child field triggers a corresponding event for the parent field. For example, if field A belongs to subform X, and the user selects field A, a Click event is triggered for field A and a ChildClick event is triggered for subform X, its parent field.
Because all fields are considered to be children of the frame's top form, each field event that is triggered in turn triggers a corresponding child event for the frame. The rules for queuing these events are:

1. The original event for the child field is always queued first.
2. The immediate parent for the field is queued second.
3. The parent field of the parent field is queued next, and so on, up to the frame.

The following diagram illustrates this concept:

In the following example frame script, the ChildEntry event is specified for the frame, so that whenever the user enters any field on the form, the color of that field is changed to white. The ChildExit event is also specified for the frame, so that whenever the user leaves the field, the color is changed back to pale gray.

```plaintext
on childentry =
begin
  CurFrame.TriggerField.BgColor=CC_WHITE;
end

on childexit =
begin
  CurFrame.TriggerField.BgColor=CC_PALE_GRAY;
end
```
If there is a chain of events, OpenROAD queues all the child events for the first event in the chain before moving on to the second event in the chain. For example, field A and field B belong to subform Y. If the user changes the data in field A and tabs out of field A to field B, the events are triggered in the order shown in the following illustration:

---

**Menu Events**

*Menu events* are events triggered when a user selects a menu item. Menu events are simpler than field events because they have fewer available events. These menu events are:

**Click event**

The three menu fields (menu button, menu toggle, and menu list) all have a Click event, which is triggered when the user selects the menu item.

**SetValue event**

The menu toggle and menu list fields also have a SetValue event, which is triggered when the user changes the value of the menu item—that is, switches the toggle setting or selects a new item from the list.

**Validate event**

The menu toggle and menu list fields also have a Validate event, which is triggered at exactly the same places as the SetValue event, but is processed before the SetValue chain. Therefore, if a Validate event block contains a resume statement, control returns to the frame without triggering the SetValue event.
The following sample frame script is the Click event block for the Graphic operation on the View menu. This menu operation functions as a toggle field that lets the user turn on or turn off the display of the photograph associated with a given video.

```plaintext
on click menu.view_menu.show_graphic =
begin
  if menu.view_menu.show_graphic = FALSE then
    graphic_frame.Terminate();
    graphic_frame = null;
  else
    graphic_frame = openframe
    video_graphic(vid_graphic_bitmap =
       video.vid_graphic_bitmapobject)
    with windowtitle = video.title,
        windowxleft = CurFrame.WindowWidth,
        windowytop = 0;
  endif;
end

Show_video_graphic = menu.view_menu.show_graphic;
```

Based on user interaction, this block of code either terminates the graphic frame and reinitializes its FrameExec pointer to null, or opens it with the appropriate photograph and window location.

The difference between the Click event and the SetValue event is meaningful for the menu list, as follows:

- The Click event is triggered when the user clicks any list item, even the currently highlighted list item.
- The SetValue event is triggered only when the user actually changes the value of the field.

For example, assume there is a frame with a Get menu that contains a menu list called initial_acct, which offers the following choices:

- New Account
- Existing Account
- All Accounts
- Closed Accounts

Because the user might need to repeat the command, New Account, the event block for this menu list uses the Click event. If SetValue were used, nothing would happen the second time the user clicked New Account.

The Click, Validate, and SetValue events create an event chain in this order for menu toggles and menu lists.
**Note:** The form associated with each frame (CurFrame.TopForm) contains only non-menu fields. Menu fields are separate from the form because they are toolkit-specific. Therefore, menu events do not trigger corresponding child events for the frame. There are no child events for a menu stack or for the menu as a whole.

### How You Can Interrupt an Event Block

Typically, OpenROAD finishes executing the current event block before moving on to the next event in the queue. However, you can use the resume statement to interrupt the current event block. Statements following the resume statement are not executed.

In the following sample frame script, the resume statement is used to interrupt the event block if the user decides to not delete the videos. Otherwise, the processing continues.

```plaintext
on clearTable checkout_form.checkout =
  begin
    status = CurFrame.ConfirmPopup(messagetext =
      'You are going to delete all checked out videos' +
      ' for this customer.');
    if status != PU_OK then
      resume;
    endif;

    /* We will delete the rows in reverse order to play it safe */
    i = checkout_form.checkout.LastRow;
    while i > 0 do
      /* This does not actually delete the row, but tags it with a rowstate of RS_DELETED. */
      checkout_form.checkout.SetRowDeleted(rownumber = i);
      i = i - 1;
    endwhile;

    field(checkout_form.checkout).HasDataChanged = TRUE;
  end

Because the resume statement terminates the current operation, it is usually the last statement either in the event block or in the statement list of the if statement (as shown in the previous example).```
The resume statement is useful for ensuring data validation. When you use the SetValue event to perform data validation for a field, the resume statement can prevent the user from leaving the field when the data is invalid. The user remains on the field that triggered the SetValue event, unless you use the resume next statement to continue with the next operation or use the CurFrame.InputFocusField attribute to move the user to a specified field.

For more information about resume next and about the InputFocusField attribute, see the Language Reference Guide online help.

When the resume statement is executed in an event block for an event that is part of an event chain, OpenROAD removes the remaining events in the chain from the event queue. For example, after a user triggers a SetValue event for a field and leaves the field, an Exit event is triggered for that field, and an Entry event is triggered for the next field getting the input focus.

However, you can use the resume statement in the SetValue event block to prevent the user from entering invalid data. When the SetValue event block executes the resume statement, the Exit and Entry events are removed from the queue and the user remains in the field with invalid data.

Because the resume statement removes remaining events from an event chain, you can use it to prevent a window from closing when the user selects the window manager close operation for the running frame. When the user closes a window, OpenROAD triggers both the WindowClose and Terminate events for the frame.

To keep the window open under specified circumstances, you can put a resume statement in a WindowClose event block. This statement removes the Terminate event from the event queue and prevents the window from closing.

You cannot use a resume statement in a Terminate event to prevent the window from closing. Regardless of how a Terminate event block completes, it always terminates the frame and the window. For this reason, OpenROAD always processes a WindowClose event, if specified, before a Terminate event block.
How You Can Obtain Information about the Current Event

The FrameExec object (described in Programming Frames (see page 95)) contains information about the current event. This information can be very useful in your event code. The Language Reference Guide online help lists the FrameExec attributes you can use for each of the OpenROAD events.

The most useful of the FrameExec attributes are the following:

**TriggerField**

Data Type: FieldObject

Contains the field that triggered the event (the user originally interacted with this field)

**OriginatorField**

Data Type: FieldObject

Contains the field specified in the original on statement of the current event block

**InputFocusField**

Data Type: ActiveField

Contains the field on the form that currently has the input focus

**PreviousField**

Data Type: ActiveField

Contains the field that most recently had the input focus (the field that was just exited)

**TargetField**

Data Type: ActiveField

Contains the field that is to be the next field with the input focus (the field to be entered next)

In the following example, the TriggerField attribute is used to change the background color of the field that the user has just entered:

```plaintext
on childentry = 
begin 
  ActiveField(CurFrame.TriggerField).BgColor = CC_RED;
end
```
**Note:** The data type of the TriggerField attribute is FieldObject, allowing it to refer to any field on a form. However, to manipulate a specific field's attributes, such as changing an entry field's background color, you must inform OpenROAD about the correct system class of the field being affected. The process for informing the system is called *casting*. For more information about casting, see How You Can Work with Attributes (see page 140).
Chapter 13: Inter-Frame Communication Techniques

This section contains the following topics:

- Communicating Between OpenROAD Frames (see page 333)
- How You Can Communicate with an External Program Using External User Events (see page 347)
- How You Can Communicate Between OpenROAD Applications Using Database Events (see page 357)

This chapter discusses communication between frames, which involves:

- Communication between OpenROAD frames using the Terminate method, global variables, and user events
- Communicating with an external program using external user events, such as:
  - Registering the external user event
  - The user event block
  - Sending the external user event to OpenROAD
- Communicating between OpenROAD applications using database events, such as:
  - The DBEventObject
  - Database events and multiple sessions
  - Creating, registering, and raising database events
  - Purging database events
- The external user event sample

Communicating Between OpenROAD Frames

OpenROAD provides the following three techniques for communicating between frames:

- The Terminate method
- Global variables
- The user event
The Terminate method lets you close one frame from another frame. Use the Terminate method when you want to close an inactive frame. The Terminate method closes an inactive frame immediately.

When you want to close an active frame, it is best to use a user event (described in the following section). A user event lets you close an active frame conditionally, which you cannot do with the Terminate event. However, because inactive frames do not receive user events, the Terminate method is the best way to close an inactive frame.

Global variables provide a simple means of sharing data across frames. Because the value of a global variable is updated immediately, you can change the value from one frame and access the new value from another frame.

**User Events**

The user event is the most important communication technique. A user event is an event that you trigger with 4GL code (rather than an event triggered directly by a user action).

By using the SendUserEvent method in one frame script to trigger a user event in another frame, you can make one frame force the execution of some code in another frame. For example, you can provide code so that when the user updates a field in one frame, that frame sends a user event to another active frame. When the other frame receives this user event, it triggers corresponding updates to data being displayed in the frame’s window.

Another common use for a user event is to allow a child frame to communicate with its parent frame. For example, when the parent frame requests its child frame to close, the child frame may need to send a message back to the parent indicating that it cannot close because changes are outstanding.

Each of these techniques is discussed in the following sections in more detail.

**Terminate Method**

You can use the Terminate method from one frame to stop the execution of another frame. If the frame you are terminating has any child frames, these are closed also. This method is useful when you want to close a frame that is currently inactive and you do not want to wait until the frame is active again. The syntax is:

```
FrameExec_var.Terminate()
```

You can use this method for any running version of a frame. To specify the FrameExec variable for the running version of the frame, use any of the methods described in SendUserEvent Method (see page 339).
In the following example, the Terminate method is used to close the video_graphic window when the user switches the Graphics toggle on the View menu to off:

```on click menu.view_menu.show_graphic =
  begin
    if menu.view_menu.show_graphic = FALSE then
      graphic_frame.Terminate();
      graphic_frame = null;
  end;
```

If you want to catch any terminate requests and do some processing before the frame is actually closed, you can provide an event block for the Terminate event. The Terminate method triggers the Terminate event for the frame. For more information on the Terminate event, see the Language Reference Guide online help.

**Note:** The Terminate event block cannot prevent the frame from being terminated. If you include a resume statement in the event block, OpenROAD ends the event block and terminates the frame.

### Global Variables

OpenROAD provides global variables that you can use to store information that is available to all the frames in the application. You can use global variables to store global state information, such as the user’s name and security level. You can also use them to share information between frames. Because updates to global variables take effect immediately, when one frame changes the value of a global variable, the new value is immediately available to any other frame in the application.

Global variables are not displayed directly on OpenROAD forms. To display the information contained in a global variable, you can assign the global variable to a local variable that is associated with a field on the form. In this case, to keep the value of the local variable up to date, you must reassign the value of the local variable when updates are made to the global variable.

For more information about how to declare global variables, see the User Guide.
How You Can Communicate Between Frames Using User Events

You can utilize user events to communicate between two frames by performing the following basic steps:

1. Write the event block for the user event in one frame script.
   The event block for the user event contains the code that is executed when the user event is triggered.
2. Include in the other frame script the SendUserEvent method to trigger the user event in the first frame.

When the SendUserEvent method is executed, OpenROAD places the specified user event in the native event queue. From there, OpenROAD dispatches the user event to the appropriate frame. The code for the user event is executed when the event reaches the top of the frame's event queue. The following illustration demonstrates this:

You can also have one frame send a user event to itself. For example, a frame that displays a clock can send an event to itself at regular intervals to update the clock. For an example of this, see Delay Parameter (see page 342).

Because the events in event queues are processed one at a time, the event block that places the user event in the native event queue (that is, the event block that contains the SendUserEvent method) completes before the user event is processed. If you want to force the event block that sends the user event to complete after a specified user event, use the WaitFor method. The WaitFor method prevents the current event block from being executed until the current frame receives the specified user event. For more information, see WaitFor Method (see page 345).

If necessary, you can remove a user event from an event queue with the PurgeUserEvent method. For more information, see PurgeUserEvent Method (see page 347).
UserEvent Event Block

In the UserEvent event block, you provide the code that is executed when the event is triggered. The syntax is:

```plaintext
on userevent ['eventname'] =
  declare
    declarations
  end declare

  { 
    statement list
  }
```

**eventname**

Specifies any string of up to 32 characters enclosed in quotes. If you specify the event name, OpenROAD executes the event block whenever the frame receives that particular user event. If you do not specify an event name, OpenROAD executes the block whenever the frame receives a user event for which the frame does not have an event block specified by name.

For example, assume that a frame has event blocks for two user events, Event1 and Event2, and a general user event block that does not specify a user event name. When the frame receives Event1, OpenROAD executes the event block specified for Event1. Similarly, if the frame receives Event2, OpenROAD executes the event block specified for Event2. OpenROAD does not execute the generic event block in either case. However, if the frame receives Event3, for which the frame has no specific block, it executes the generic block.

You can specify any number of user event blocks in the frame script, although each must have a unique event name, or no event name at all.

Because the SendUserEvent method and external user events both trigger user event blocks, it is best to use appropriate naming conventions for your user events. (There is no way to tell within the event block whether the event was triggered by OpenROAD or by an external program.)

The event block for the user event can contain any OpenROAD statements. The following event block is for the DeleteEntry user event:

```plaintext
on userevent 'DeleteEntry' =
  begin
    delete_details_frame =
      FrameExec(CurFrame.MessageObject);
```
Communicating Between OpenROAD Frames

/* Determine which row in table contains ** video. */
if find_video_row
  {video_list = vlist, details_frame = delete_details_frame, row = byref(i)}
  = TRUE
  then
/* Now delete it */
  vlist.RemoveRow(rownumber = i);
  endif;

/* Now add it to new details frames list, ** since it is now in insert mode */
i = new_details_frames.LastRow() + 1;
new_details_frames[i] = delete_details_frame;
end;

How You Can Access the SendUserEvent Parameters

The SendUserEvent method has four message parameters, messageobject, messageinteger, messagefloat, and messagevarchar that you can use to pass values to the receiving frame. For information about these parameters, see SendUserEvent Method (see page 339).

If the SendUserEvent call activates a userevent event block in the receiving frame, OpenROAD stores these values in four corresponding FrameExec attributes, MessageObject, MessageInteger, MessageFloat, and MessageVarchar, that are accessible only in the event block. To access this data, use the CurFrame system variable to reference the current FrameExec.

In the following sample from a frame script, the UpdateTitle event block sets the window title to the value of the MessageVarchar attribute that is passed to the frame by the messagevarchar parameter of the SendUserEvent method:

on userevent 'UpdateTitle' =
begin
  CurFrame.WindowTitle =
  CurFrame.MessageVarchar;
end:

Because the MessageObject attribute stores an object of the generic Object class, you must cast the object to the appropriate system or user class in order to work with it in your user event block. (Casting is described in How You Can Work with Attributes (see page 140).)
In the following example, the MessageObject attribute contains a bitmap, so it is cast to the BitmapObject system class:

```plaintext
on userevent 'UpdateGraphic' =
begin
    vid_graphic_bitmap = BitmapObject(CurFrame.MessageObject);
    CurFrame.Flush();
    CurFrame.IsAutoSized = TRUE;
    CurFrame.WindowVisibility = WV_VISIBLE;
    CurFrame.BringToFront();
end;
```

In the next example, the MessageObject attribute contains an object of type video_row. The UpdateEntry event block casts the MessageObject to the video_row user class.

```plaintext
on userevent 'UpdateEntry' =
begin
    video = VIDEO_ROW(CurFrame.MessageObject);
```

**Note:** If the SendUserEvent call completes a WaitFor method call, the message parameters are stored in the Event object returned by the WaitFor call. The Event object has five attributes. One attribute holds the name of the event and each of the other four holds one message parameter. You can access the parameter values in the Event object attributes in the event block that contains the WaitFor call.

### SendUserEvent Method

Use the SendUserEvent method to trigger the user event in a frame. When the user event is triggered, OpenROAD adds it to the application's native event queue. If the frame is active, OpenROAD queues the user event immediately. If the frame is inactive because it has called another frame or procedure, OpenROAD waits until the frame becomes active again before it queues the user event. The message parameters of the SendUserEvent method let you send data to the receiving frame.

The syntax of the SendUserEvent method is:

```plaintext
FrameExec_var.SendUserEvent(eventname = varchar, messageobject = object, messageinteger = integer, messagefloat = float, messagevarchar = varchar, delay = float, focusbehavior = integer, errorevent = varchar)
```
Communicating Between OpenROAD Frames

The following example uses the SendUserEvent method to send the Flushvalues event to a child frame.

```plaintext
... begin ...
    childframe.SendMessage
        (eventName = 'Flushvalues',
         messageObject = CurFrame,
         errorEvent = 'WaitForEvent');
end;
```

To specify the frame that contains the user event you want to trigger, you must reference the FrameExec variable for the running instance of the frame. (You cannot use the frame name because OpenROAD allows you to open the same frame more than once.) There are four ways to reference the FrameExec variable:

- Use the CurFrame system variable name to reference the current frame.
  
  Use this variable when the frame needs to send a user event to it. The following frame script example illustrates this concept:
  ```plaintext
  CurFrame.SendMessage(eventName = 'LoadBalanceTable');
  ```

- Use the ParentFrame attribute to reference the parent of the current frame.

  Because the ParentFrame attribute represents the ProcExec object for the parent frame, you must cast it as a FrameExec object in order to use it as a FrameExec reference. The following frame script example illustrates this concept:
  ```plaintext
  on windowclose =
      begin
          FrameExec(CurFrame.ParentFrame).SendMessage
              (eventName = 'GraphicClosed');
          return;
      end;
  ```

- Use the reference variable returned by the openframe statement that you used to open the frame.

  In the following example, graphic_frame is a variable of type FrameExec:
  ```plaintext
  graphic_frame.SendMessage
      (eventName = 'UpdateTitle',
       messagevarchar = video.title);
  ```

- Store a FrameExec reference obtained through any of the previously mentioned methods in a global variable so you can access it from other frames.
**Message Parameters**

The four optional message parameters of the SendUserEvent method, messageinteger, messagefloat, messagevarchar, and messageobject, let you pass values to the receiving frame. If the SendUserEvent method triggers a user event block, OpenROAD stores the values in the corresponding message attributes of the receiving frame's FrameExec object, MessageInteger, MessageFloat, MessageVarchar, and MessageObject. If the SendUserEvent method completes a WaitFor call in the receiving frame, OpenROAD stores the values in the Event object returned by the WaitFor call when it completes.

Use the three simple-variable parameters, messageinteger, messagefloat, and messagevarchar, to pass integer, float, and varchar values, respectively. The following example uses the messagevarchar parameter to send the title of the video to a frame:

```
on setvalue video.title =
begin
  CurFrame.WindowTitle = video.title;
  if graphic_frame is not null then
    graphic_frame.SendUserEvent
    (eventname = 'UpdateTitle',
     messagevarchar = video.title);
  endif;
end;
```

Use the messageobject parameter to pass any object. Because the object can be any system or user class, this parameter enables you to send any set of information that you want. This is especially useful with user classes. Simply create a user class that specifies the set of data you want to pass and then use the messageobject parameter to pass an object of that class. The following example uses this technique to pass information about a new video to the video_list frame. New_video is a reference variable of the video_row user class.

```
parent_frame.SendUserEvent(eventname =
  'InsertEntry', messageobject = new_video);
```

In the receiving frame, you can access the object with the MessageObject attribute for CurFrame (described in UserEvent Event Block (see page 337)).

**Focusbehavior Parameter**

Every frame has its own input focus. When the user leaves a window, the frame does not lose its input focus. The input focus stays on the same field, even though the user has selected another field in another window.
For example, if the user is in the process of entering data into a field on Window 1 and then selects Window 2, the input focus for Window 1 stays on the field where the user was entering the data. Because the field on the first frame still has the input focus, the data validation code for the field is not triggered, even though the user has left the window. (Typically, the data validation code for a field is contained in the SetValue or Exit event blocks for the field, so data validation takes place when the user changes or leaves the field.) The SendUserEvent method provides a focusbehavior parameter to enable you to deal with this situation.

The focusbehavior parameter of the SendUserEvent method lets you trigger the SetValue event or both SetValue and Exit events for the field with the input focus before the user event is triggered. The values for the focusbehavior parameter are:

**FT_SETVALUE**
- Triggers the SetValue event for current field

**FT_TAKEFOCUS**
- Triggers the SetValue and Exit events for current field

**FT_NOSETVALUE**
- Specifies no forced processing

**Default:** FT_NOSETVALUE

**Note:** Setting the focusbehavior parameter to FT_SETVALUE or FT_TAKEFOCUS can cause the receiving frame to discard the user event that you sent. This happens when the SetValue or Exit event block for the current field in the receiving frame executes a resume statement. (For more information about the resume statement, see the Language Reference Guide online help.) The user event is also discarded if a data type conversion error occurs when OpenROAD attempts to execute the SetValue event.

The following example uses the focusbehavior parameter to trigger the SetValue event in its child frames before they close:

```c
vlist[i].details_frame.SendUserEvent(eventname='Cleanup',focusbehavior = FT_SETVALUE);
```

**Delay Parameter**

When you want to repeat a task at regular intervals, such as polling the database for changes or updating the time on a clock, you can trigger a user event after a specified amount of time. The delay parameter of the SendUserEvent lets you specify a number of seconds to wait before triggering the user event.
OpenROAD triggers the user event after the specified time has elapsed. In the meantime, OpenROAD continues executing the current event block, and events can continue to be triggered in any of the currently active frames. The default delay is 0.0 seconds (in other words, the event is triggered immediately).

In the following example, we use the delay parameter to update a block every 10 seconds:

```plaintext
class initialize
begin
CurFrame.SendUserEvent (eventname = 'clock');
end;

don event 'clock'
begin
    current_time = date('now');
    CurFrame.SendUserEvent (eventname = 'clock',
        delay = 10.0);
end
```

### ErrorEvent Parameter

The errorevent parameter specifies a user event that is returned to the sending frame by the system if the current user event cannot be delivered. The following examples demonstrate how to use this parameter.

The following code would be in the sending frame:

```plaintext
begin
    declare
        response = Event;
    end declare
begin
    targetframe.SendUserEvent (eventname = 'originalevent',
        errorevent = 'replyevent');
    response = CurFrame.WaitFor (eventname = 'replyevent');
    if response.MessageErrorCode = 0 then
        /*
        ** The target frame received and replied.
        */
    ...
else
    /*
    ** The target frame never got the event.
    */
    ...
end if;
end;
```
In the target frame:

```plaintext
on userevent 'originalevent' =
{
    /* Acknowledge that the event is received */
    sendingframe.SendUserEvent
        (eventname = 'replyevent');

    /* Now process the event */
    ...
}
```

**Important!** The target frame must reply with the replyevent event. Otherwise, the WaitFor method will not return, providing the event was delivered.

The following examples show another way of using the errorevent parameter without using the WaitFor method.

The following code would be in the sending frame:

```plaintext
begin
    targetframe.SendUserEvent
        (eventname = 'originalevent',
         errorevent = 'errorevent');
end;
```

The following code would be in the target frame:

```plaintext
on userevent 'originalevent' =
{
    /* Process the event */
    ...
}
```

```plaintext
on userevent 'errorevent' =
{
    /* Error handling */
    ...
}
```

For more information about MessageErrorCode, see the *Language Reference Guide* online help.
**WaitFor Method**

Usually, when you use the SendUserEvent method, OpenROAD queues the user event and then continues executing the current event block. The WaitFor method lets you prevent the rest of the current event block from being executed until you tell it to stop waiting.

When you invoke the WaitFor method, you specify the name of a user event that the event block must wait for. When the current frame receives that user event, control returns to the statement immediately following the WaitFor call.

The syntax of the WaitFor method is:

```python
event = CurFrame.WaitFor(eventname = varchar)
```

You can use the WaitFor method for the current frame only with the CurFrame reference variable. The method returns an Event object that contains the eventname, messageobject, messageinteger, messagefloat, and message varchar parameters sent with the SendUserEvent call that completes the method. (When the SendUserEvent call completes a WaitFor call, the SendUserEvent focusbehavior parameter is ignored in the receiving frame.)

WaitFor makes the executing frame ignore any other events, both from the user and other SendUserEvent methods. Window-level events, such as button clicks, are disabled. User events are delivered to the frame after it receives the user event that it is waiting for.

**Important!** Do not use the WaitFor method when you are sending the user event from the current frame to itself. The user event is never delivered to the frame and the event block is never continued. This results in the frame remaining in an unending wait state.

The most common use of the WaitFor method is immediately following a SendUserEvent method to wait for acknowledgment that the user event was processed. This is especially useful when coordination of frames is important.
The following sample frame script uses the WaitFor method to wait until all child frames have closed themselves before executing the return statement to close itself:

```plaintext
on click close_button,
on windowclose =
begin
   /* If there are any detail frames open, ** tell them to clean up themselves. */
i = 1;
while i <= vlist.LastRow() do
   if vlist[i].details_frame is not null then
      vlist[i].details_frame.SendUserEvent
         (eventname = 'Cleanup',
          focusbehavior = FT_SETVALUE);
      CurFrame.WaitFor(eventname = 'Done');
   endif;
i = i + 1;
endwhile;

/* If there are any detail frames for ** new videos, tell them to clean up. */
i = 1;
while i <= new_details_frames.LastRow() do
   new_details_frames[i].SendUserEvent
      (eventname = 'Cleanup',
       focusbehavior = FT_SETVALUE);
   CurFrame.WaitFor(eventname = 'Done');
i = i + 1;
endwhile;

return;
end;

Be sure to use the SendUserEvent method to send the user event back to the frame that contains the WaitFor call. The following example sends the Done user event back to the parent frame:

```plaintext
if cleaning_up = TRUE then
   parent_frame.SendUserEvent(eventname = 'Done');
```
PurgeUserEvent Method

The PurgeUserEvent method lets you remove user events from the event queue. If you trigger a series of related user events and one of the events fails for some reason, you can use this method to remove the remaining user events from the event queue.

In addition, when you use the resume statement to prevent the user from leaving a field that contains invalid data, OpenROAD does not remove the user events from the event queue (although it does remove the other frame events for the current frame). Use the PurgeUserEvent method to remove them if necessary. The PurgeUserEvent method removes one or all the user events from the event queue.

The syntax of this method is:

\[ \text{integer} \_\text{var} = \text{FrameExec} \_\text{var}.\text{PurgeUserEvent} \]
\[ (\text{eventname} = \text{varchar}(256)) \]

The PurgeUserEvent method removes the specified user event from the event queue. If you do not specify the eventname parameter, it removes all user events for the specified frame. The method returns the number of events removed from the queue.

You can use this method for any running version of a frame. To specify the FrameExec variable for the running version of the frame, use any of the methods described in SendUserEvent Method (see page 339).

How You Can Communicate with an External Program Using External User Events

In many computing environments, there are multiple applications running concurrently. Often, it is critical that these applications communicate with each other. For example, in a manufacturing plant, several applications might control the production process, each sending status information such as temperatures and inventory use to a central location. Another application might gather this information continuously, providing a central display that lets workers monitor the machines and tells them when there is a problem. This application communicates with the process applications to update its displays.

OpenROAD applications can communicate with external programs through user events. Just as you can trigger a user event for frames with 4GL code, you can trigger a user event from an external program.
First, you register a frame (or frames) with OpenROAD to receive a particular event. Then, when the event occurs in the external program, the program notifies the OpenROAD application, which adds the user event to the application's native event queue.

When the event reaches the top of the queue, OpenROAD dispatches the event to those frames registered to receive the event. When the event reaches the top of the frame's event queue, OpenROAD executes the event code for the user event.

This feature lets you send a message from the external program to the application notifying it, for example, to update its display or check the database for changes. You can also pass one integer value along with the user event, such as a record or file ID. Since both the OpenROAD application and the external program can have open sessions to the same database, this provides a good way for the two programs to interact.

Unless the communicating programs are both OpenROAD applications, using user events to communicate is a one-way process, from the external program to the OpenROAD application. If both programs are OpenROAD applications, each can call a 3GL procedure to communicate with the other.

The basic steps for using external user events are:

1. Use the 4GL RegisterUserEvent method in a frame script to register the external event.
2. In your frame script, write the event block for the external user event. The event block for the user event contains the code that is executed when the external event is triggered.
3. Using the routines described in How You Can Register the External Event (see page 348), write a 3GL program that sends the event from your external program.
4. Compile your 3GL program and link it as an external program.

**How You Can Register the External Event**

When you want a frame to receive a particular event from an external program, you must register the frame with OpenROAD for that particular event. Registering a frame tells OpenROAD to notify the frame whenever an external program sends the specified event. For example, if frame A is registered to receive the external _update event, whenever OpenROAD receives the external_update event from the external program, it notifies frame A that the event has occurred. OpenROAD then adds the external_update event to the event queue. (If the specified frame is not running at the time, OpenROAD simply ignores the event. This does not cause an error.)
You can register one frame for any number of external events. You can also register more than one frame for the same external event.

To register the frame to receive the external event, use the FrameExec RegisterUserEvent Method (see page 349). To remove an event from the list of registered events for the frame, use the FrameExec UnRegisterUserEvent Method (see page 349).

**RegisterUserEvent Method**

The RegisterUserEvent method registers a frame to receive the specified external event. The syntax follows:

\[
\text{FrameExec}_\text{var}.\text{RegisterUserEvent} (\text{eventName} = \text{varchar}(32))
\]

To specify the frame to be registered, you must reference the FrameExec variable for the running instance of the frame. (You cannot use the frame name because OpenROAD lets you open the same frame more than once.) For more information about how to reference the FrameExec variable, see SendUserEvent Method (see page 339).

To register the event for the frame that currently has the input focus, use the following syntax:

\[
\text{CurFrame}.\text{RegisterUserEvent} (\text{eventName} = \text{varchar}(32))
\]

**eventName**

Specifies the name of the user event. This must be the same as the name you use in the corresponding user event block (see UserEvent Event Block (see page 350)). This parameter is case sensitive.

The following example illustrates registering the external_update user event:

\[
\text{CurFrame}.\text{RegisterUserEvent} (\text{eventName} = \text{'external\_update'}}); 
\]

**UnRegisterUserEvent Method**

The UnRegisterUserEvent method removes one or all external events from the list of registered events for a frame. The syntax follows:

\[
\text{FrameExec}_\text{var}.\text{UnRegisterUserEvent} (\text{eventName} = \text{varchar}(32))
\]

To specify the frame, you must reference the FrameExec variable for the running instance of the frame. For more information about how to reference the FrameExec variable, see SendUserEvent Method (see page 339).
**eventname**

Specifies the name of the user event. If you do not specify the event name, all registered events for the frame are removed from the list. This parameter is case sensitive.

The following example illustrates this:

```curframe.UnRegisterUserEvent(eventname = 'external_update');```

**UserEvent Event Block**

The UserEvent event block for an external user event is the same as the event block for an internal user event. In this event block, you provide the code that is executed when the event is triggered. The syntax follows:

```on userevent ['eventname']
begin
  statement list
end;```

**eventname**

Specifies any string of up to 32 characters enclosed in quotes. If you specify the event name, OpenROAD executes the event block whenever the frame receives that particular user event. This name must be the same as the one you use when you register the frame to receive the event with the RegisterUserEvent method as described in RegisterUserEvent Method (see page 349). The following example illustrates this:

```on userevent 'external_update'
begin
  program statement
  program statement
  program statement
end;```

If you do not specify an event name, OpenROAD executes the block whenever the frame receives a user event for which there is no specific event block in the frame script. You can specify any number of user event blocks in the frame script, although each must have a unique event name, or no event name at all.

You can discover the name of the event that caused the execution of an unnamed user event block by referencing the EventName attribute of the frame's FrameExec object.
Because external events and the OpenROAD SendUserEvent method both trigger user event blocks, it is preferable to use appropriate naming conventions for your user events. (It cannot be determined within the event block whether the event was triggered by OpenROAD or by an external program.)

The event block for the external event can contain any OpenROAD statements.

**How You Can Pass an Integer Value to an Event Block**

The IIW4GL_SendUserEvent procedure, which you use in your 3GL program to send the user event, has a messageinteger parameter that you can use to pass one integer value to the user event block. In the receiving frame, OpenROAD stores this value in the MessageInteger attribute of the FrameExec object. To access this data in your user event block, use the CurFrame system variable to reference the current FrameExec.

For more information about this procedure, see IIW4GL_SendUserEvent Procedure (see page 351).

**How You Can Send the External User Event to OpenROAD**

To send the event from the external program to OpenROAD, you must write a 3GL program that sends the named event to OpenROAD.

This program sends the event to any running OpenROAD frames. If an OpenROAD frame has registered for that event, the associated event block is triggered. If none of the events are registered or if none of the registered frames are running, the request is ignored.

OpenROAD provides the IIW4GL_SendUserEvent procedure to send the event name and optional integer value. This procedure is described in the IIW4GL_SendUserEvent Procedure (see page 351).

**IIW4GL_SendUserEvent Procedure**

The IIW4GL_SendUserEvent procedure sends the event name and optional integer value directly to the OpenROAD application. It returns an integer status value. The syntax follows:

```
status = IIW4GL_SendUserEvent(errormsg, &sent, token,
                              eventname, fb, msgi4)
```
This procedure takes the following arguments:

**errormsg**
Specifies an output character string variable that holds the text of any error that occurs while the procedure executes. Declare this as "char errmsg[256]."

**&sent**
Specifies an output parameter that indicates how many frames are registered for the specified event and, consequently, how many events were sent. You must use the & operator. Declare this as “int sent.”

**token**
Specifies an obsolete parameter. No values are sent or returned to this variable. Declare this as "char *token.”

**eventname**
Specifies an input parameter that identifies the user event. This event must have been registered by a previous 4GL RegisterUserEvent method. This parameter is case sensitive and must be null terminated. Declare this as "char *eventname.”

**fb**
Specifies an input parameter that specifies the focus behavior for the current field in the frame. Declare this as "long fb.”

This parameter corresponds to the focusbehavior parameter of the SendUserEvent method for internal events. It lets you trigger the SetValue event or both SetValue and Exit events for the field with the input focus before the external event is triggered. For more information, see Focusbehavior Parameter (see page 341).

Valid values for **fb** are:

1.
   FT_NOSETVALUE

2.
   FT_SETVALUE

3.
   FT_TAKEFOCUS

These FT constants are defined in the w4glext.h include file.

**msgi4**
Specifies an input parameter that specifies a single integer value to be sent to the receiving frame. In the receiving frame, you can access this value with the MessageInteger attribute of the FrameExec object. Declare this as “long msgi4.”
How You Can Communicate with an External Program Using External User Events

status

Specifies a return variable that is an output integer variable that indicates the success or failure of the procedure. If the return value is IIW4GL_OK, the send was successful. If the return value is a nonzero value, the send failed (the text of the error message is in errormsg). Declare this as "int status."

External User Event Demo

If you have installed the OpenROAD Development package, a sample is provided to demonstrate how external user events are used to facilitate the communication from a 3GL application to an OpenROAD application. The demo files are included in the following directory:

Windows

%II_SYSTEM%\inges\w4glamp\extevent

UNIX or Linux

$II_SYSTEM$/inges/w4glamp/extevent

For instructions to use the demo, see the appropriate section, following, for your platform.

How the External User Event Demo Works on Windows

The directory %II_SYSTEM%\inges\w4glamp\extevent contains the following files:

extevent.exp

Is an export file of the OpenROAD application used to run the sample application

extevent.c, extev.h, extevent.rc

Are source files for a non-OpenROAD executable

makefile

Is a makefile for building the external application

To use the demo application, you must perform the following basic steps:

1. Build the 3GL application.
2. Import the sample application into your database.
3. Run the sample application.

See the following sections for details on each of these steps.
Build the 3GL Application

You can build the 3GL application by setting environment variables and using the makefile to build the sample application.

To build the 3GL application

1. Set your PATH, LIB, and INCLUDE environment variables using the Microsoft compiler.
   The II_SYSTEM environment variable should point to your OpenROAD installation.
2. Use the makefile to build extevent.exe:
   
nmake all

Import the Sample Application into Your Database

You can import the sample OpenROAD application by entering the following command at the command prompt:

w4gldev backupapp in dbname ExtEvent
"%II_SYSTEM%\ingres\w4glsamp\extevent\extevent.exp"

Run the Sample Application

You can run the sample application by entering the following command at the command prompt:

w4gldev rundbapp dbname ExtEvent

The sample application starts.

To use the application

1. Click the Register Event button.
2. Click the Send menu button in the external application frame.
   The ExtEvent demo frame displays a message that confirms that the event was received.
3. Click the UnRegister Event button in the ExtEvent demo.
4. Click the Send menu button again.
   The OpenROAD application does not receive the external event.
How the External User Event Demo Works on UNIX or Linux

The directory $II_SYSTEM/ingres/w4glsamp/extevent contains the following files:

**extevent.exp**
Is an export file of the OpenROAD application used to run the sample application

**exteventu.c**
Is a source file for a non-OpenROAD executable

**makefile_hp**
Is a makefile for compiling and linking liborevent.sl on the HP platform

**makefile_sol**
Is a makefile for compiling and linking liborevent.so on Sun Solaris

**makefile_aix**
Is a makefile for compiling and linking liborevent.so on IBM AIX

**makefile_lnx**
Is a makefile for compiling and linking liborevent.so on Linux

**makefile_axp**
Is a makefile for compiling and linking liborevent.so on the Compaq Tru64 platform

To use the demo application, you must perform the following basic steps:

1. Build the 3GL application.
2. Import the sample application into your database.
3. Run the sample application.

See the following sections for details on each of these steps.
How You Can Communicate with an External Program Using External User Events

Build the 3GL Application

You can build the 3GL application by setting environment variables and using the makefile to build the sample application.

**To build the 3GL application**

1. Set your PATH and SHARED LIBRARY PATH environment variables to run OpenROAD.
   The Mainwin runtime environment should also be initialized.
2. Use the makefile provided for your platform to compile exteventu.c:
   ```shell
cd $II_SYSTEM/ingres/w4glsamp/extevent
make -f makefile_yourplatform
```

Import the Sample Application into Your Database

You can import the sample OpenROAD application by entering the following command on the command line:

```shell
w4gldev backupapp in dbname ExtEvent
%II_SYSTEM%/ingres/w4glsamp/extevent/extevent.exp
```

Run the Sample Application

You can run the sample application by entering the following command at the command prompt:

```shell
w4gldev rundbapp dbname ExtEvent
```

The sample application starts.

**To use the application**

1. Click the Register Event button.
2. Click the Send menu button in the external application frame.
   The ExtEvent demo frame displays a message that confirms that the event was received.
3. Click the UnRegister Event button in the ExtEvent demo.
4. Click the Send menu button again.
   The OpenROAD application does not receive the external event.
How You Can Communicate Between OpenROAD Applications Using Database Events

Database events are user-defined events. They are created and manipulated using SQL statements and stored in a database. Database events are a way for two programs that are connected to the same database to communicate. Using a database event, one program can inform another that some predefined event has occurred. The receiving program then can take whatever action is necessary. Database events allow a program to send up to 256 bytes of information to another program.

How Communication Using a Database Event Works

Communication using database events is two-way: any application can either send or receive a database event.

To use a database event, you would perform the following basic steps:

1. Use the SQL create dbevent statement to create the database event.
2. In the receiving application, use the SQL register event statement to register frames to receive the event.
3. In the frames' scripts, write the database event blocks.
4. In the sending application, use the SQL raise dbevent statement to send the event.

When an application executes a raise dbevent statement, the application sends the specified event, and any specified text, to the DBMS. In turn, the DBMS sends it to any application that is registered to receive the event and is connected to the same database as the sending application.

In OpenROAD, applications poll regularly for database events from the DBMS. The DBEventPollrate attribute defined for the SessionObject determines the time interval between each polling.

When an OpenROAD application polls and finds a waiting database event, it places the event on the native event queue and initializes a separate DBEventObject for each frame registered to receive the event. When the event reaches the top of the native event queue, OpenROAD dispatches it to the appropriate frame queues.
**DBEventObject**

When an application receives a database event from the DBMS and places it on the native event queue, it also initializes a DBEventObject for each frame that is registered to receive the database event. The attributes of the DBEventObject contain the following information:

- **CurFrame.DBEvent.DBEventName**
  Contains the event's name

- **CurFrame.DBEvent.DBEventOwner**
  Contains the name of the event's owner

- **CurFrame.DBEvent.DBEventDatabase**
  Contains the database in which the event is stored

- **CurFrame.DBEvent.DBEventTime**
  Contains the date and time when the event was raised

- **CurFrame.DBEvent.DBEventText**
  Contains the text specified when the event was raised, if any

In the frame that receives the event, the values in the DBEventObject are only accessible in the event block that the frame executes for the event. In this block, you use the CurFrame variable to reference the event's DBEventObject.

**How Database Events Work with Multiple Sessions**

When the DBMS receives the event from a sending application, it checks to see which applications are registered for that event and sends the event to those applications. Because the DBMS perceives each database session in an application as a separate application, a multiple-session application receives a database event from the DBMS in the same session in which the register dbevent statement for that event was issued. The application receives all database events for which it registered in any currently open session even if the application is not currently working in the session.

When the application receives a database event, OpenROAD ultimately dispatches it to the frame or frames that are registered to receive that event. When the event reaches the top of the frame's event queue, OpenROAD processes it. To execute the event block for the event, the frame must be working at this time in the session in which the event was registered. If the frame is working in a different session when the event reaches the top of the frame's event queue, OpenROAD discards the event.
How You Can Create, Register, and Raise Events

The tasks of creating, registering, and raising events are performed with the SQL statements, create dbevent, register dbevent, and raise dbevent, respectively. This section provides a brief overview of each statement as it is used in the OpenROAD programming environment.

How You Can Create the Event

The SQL create dbevent statement creates a database event. When you create a database event, you are the owner of the event, and it is part of the database with which you were connected when you created the event, for example:

```
initialize()=
{
    create dbevent myevent;
    commit;
}
```

Generally, you can create database events outside of the applications that send or receive them. When you are designing your applications, you can decide what database events are needed and create them all at once. However, it is possible to create and send a database event in the same application.

How You Can Register the Event

Each frame must be registered to receive those database events that you want it to receive. To register a frame for a database event, you use the register dbevent statement. Generally, you place the register dbevent statement in the frame's initialize statement block. However, you can put it in an event block if you want, for example:

```
initialize()=
{
    register dbevent myevent;
    commit;
}
```

An event registration is associated with a specific session. The frame receives all events for which it is registered in any currently open session. However, to process the event, the frame must be working in the session where the event was registered when the event reaches the top of the frame's event queue.
For example, assume that Frame A switches back and forth between two Sessions, 1 and 2. If the frame issues a register dbevent in Session 1, then the frame processes the specified event only when it is working in Session 1. If the frame happens to be working in Session 2 when the event reaches the top of the frame’s event queue, the event is discarded. (For more information about the interaction of database events and multiple sessions, see How Database Events Work with Multiple Sessions (see page 358).)

The application’s effective user must have permission to register events. (The effective user is the user name under which the application is running.) In a multi-session application, each session can be running under a different user name. If you want to execute the register dbevent statement in a particular session, you must ensure that its effective user has the register privilege.

Registrations are valid as long as the session in which they were executed is open or until you use the remove dbevent statement to unregister the event. Using the remove dbevent statement removes the registration only for the frame where you execute the remove dbevent statement. You must issue this statement in the same session in which you registered the event. Disconnecting a session automatically removes all of the event registrations associated with that session for all frames.

**How You Can Raise an Event**

To send a database event, you use the raise dbevent statement. This statement sends the specified database event to the DBMS for distribution to those applications registered to receive the event, for example:

```plaintext
on click =
{
    buff = 'Sending DBEvent';
    raise dbevent myevent :buff:
    commit;
}
```

To issue the raise dbevent statement, the application’s effective user must own the specified database event or must have the raise privilege. In a multi-session application, this means that the effective user of the session that issues the statement must either own the specified event or have the raise privilege.

**How You Can Purge Database Events**

The PurgeDBEvent method lets you remove database events from a frame’s event queue. The syntax follows:

```plaintext
integer = GhostExec.PurgeDBEvent(dbeventname = varchar(256))
```

---

360  Programming Guide
This method removes the database event specified in the dbeventname parameter from the frame queue. If you do not include this parameter, the method removes all of the database events on the frame's queue. When the method completes, it returns the number of events that it removed.

**How You Can Communicate Between Frames with Database Events**

You can use database events to communicate between frames in the same way that you use them to communicate between applications. However, using user events and the SendUserEvent method provides a more efficient way to do this and lets you pass values as well.
Chapter 14: Creating Dynamic Frames

This section contains the following topics:

- **Conceptual Background** (see page 364)
- **How You Can Change Field and Frame Appearance and Behavior** (see page 365)
- **How You Can Create and Modify Fields Dynamically** (see page 373)
- **How You Can Remove Fields Dynamically** (see page 378)
- **How You Can Declare a Composite Field's Array Variable Dynamically** (see page 380)
- **How You Can Manipulate Data in Dynamic Fields** (see page 382)
- **How You Can Manage Operations with a Control Button** (see page 389)

In OpenROAD you can develop a single frame that serves multiple purposes by changing the bias and modes of fields and frames created in OpenROAD Workbench. This information is presented in the beginning of this chapter. For example, you can code the frame to change in the following ways:

- Display data initially as read only, allowing changes to data only in specified circumstances
- Display different data to different users (for example, salary information available only to managers)
- Allow the user to specify at runtime the database tables and columns to display and manipulate

The first two operations can be accomplished by changing the bias of form fields and menu items. When several fields are affected, changing the frame's mode simplifies these operations. The third operation requires dynamically creating fields based on user specifications at runtime.

This chapter discusses how to create fields dynamically and how to access the data values of these fields. This chapter also describes the process of creating and modifying fields dynamically and the means for accessing the values of such fields. You will also learn how to create frames whose fields are added or deleted either at frame start-up or while the frame is running.
Conceptual Background

By changing a field or menu option's bias, you can change its appearance and behavior. Even objects created in OpenROAD Workbench can be changed at runtime. These changes include:

- Appearance of menu options and buttons, which includes the following available states:
  - Enabled
  - Disabled
  - Invisible

- Behavior of fields and frames, which includes the following available states:
  - Visible
  - Changeable
  - Resizable
  - Moveable
  - Landable

You can also create frames whose fields are added or deleted either at frame start-up or while the frame is running. Adding fields to a frame by creating them at runtime is a straightforward process. However, fields created dynamically do not have the data variables of most fields created in OpenROAD Workbench. Therefore, accessing the data values of these fields involves using methods and objects.

When you create a field in Workbench, the Variable Declared toggle field on the Property Inspector for that field defaults to on. The variable declared by default lets you access the field's data value directly. This data variable cannot be used to refer to the attributes of the object.

When you create a field in Workbench, you generally change the data variable name provided by default on its Property Inspector. You reference your data variable name in your 4GL code whenever you need to access the data in the field.

For example, assume that you create an entry field and overwrite the default variable name (Field0) as title. You can set the value in that field with the following statement:

```java
title = 'Destry Rides Again';
```
It is not necessary to declare field and menu item variables in your scripts; OpenROAD declares them automatically when the Variable Declared toggle field is set to on. If you do declare title or another field name as a variable in your 4GL script, you get a compile error.

To reference the field object represented by a field or menu item instead of its value, use the field function with the data variable. For example, the following statement changes the background color of the entry field described in the previous code example:

```latex
field(title).bgcolor = CC_ORANGE;
```

When you create a field dynamically, you do so by declaring and instantiating a reference variable that points to the field object. You use that variable to set the field's attributes and attach it to the form. Because the field function operates on the data variable to return its reference variable, it is irrelevant to dynamically created fields. Fields created dynamically do not have a data variable.

Moreover, the field function is unnecessary with fields created dynamically because the variable used in creating the dynamic field references its field object. However, because the data value is not an attribute of an object, accessing the data value of such an object requires the GetFieldValue and SetFieldValue methods or the use of dynamic expressions (DynExpr).

How You Can Change Field and Frame Appearance and Behavior

Every field and menu item in OpenROAD has a bias. Biases let you change the way the user can interact with a field under certain conditions. The field’s bias determines whether the field is visible and how the user can interact with it. For example, when a field is set to the Changeable bias, the user can change the data in the field. When the same field is set to the Visible bias, the user can see the field but is unable to select it.

The bias of a field also determines which events it can receive. For example, when a field is set to the DragSegment bias, it can receive a DragSegment event. When set to any other bias, it cannot receive this event.

In addition to providing bias settings for every field and menu item, OpenROAD provides six different modes for every frame. Modes are sets of biases that let you change the appearance and behavior of several fields on a form with a single line of code.
The following items are ways to change the bias for a field. Using these following techniques depends on how many fields you want to change and when you want to change them:

- Setting the CurBias attribute immediately changes a specific field’s bias regardless of the current frame mode.
- Changing the frame mode simultaneously changes the bias of several or all fields to predefined biases.
- Setting the mode-bias attribute changes the field’s bias when the frame changes to a specific mode.

These techniques are discussed in the following sections.

**Biases for Fields**

The OpenROAD field biases are divided into the following categories:

**Interactive**

- Lets the user interact with the fields; for example, by tabbing into or selecting the field or by changing data in the field

**Passive**

- Prevents the user from entering data in the field, either by making the field invisible or by displaying the field but disallowing data entry

**Select**

- Is provided for applications that let the user manipulate the fields themselves, such as visual editors

**Draw**

- Is provided for graphic applications that let the user manipulate the fields themselves, such as diagrams and network maps

The interactive and passive biases are used most frequently. These biases are useful for applications that let the user display and enter data.

The following table lists the field biases and their corresponding categories:

<table>
<thead>
<tr>
<th>Bias</th>
<th>Description</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changeable</td>
<td>Lets the user select or tab to the field and to edit displayed data</td>
<td>Interactive</td>
</tr>
<tr>
<td>Landable</td>
<td>Lets the user select or tab to the field, but the user cannot edit displayed data</td>
<td>Interactive</td>
</tr>
<tr>
<td>Visible</td>
<td>Displays the field</td>
<td>Passive</td>
</tr>
</tbody>
</table>
### Bias Description Category

<table>
<thead>
<tr>
<th>Bias</th>
<th>Description</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewable</td>
<td>Lets the user view the contents of field with a mouse click</td>
<td>Interactive</td>
</tr>
<tr>
<td>Dimmed</td>
<td>Causes the field to be grayed out and the user cannot interact with it. (Only supported for some fields; for all other fields, Dimmed is equivalent to Visible.)</td>
<td>Passive</td>
</tr>
<tr>
<td>Invisible</td>
<td>Does not display the field on the form, so the user cannot interact with it. However, the application can continue to set and access data associated with the field.</td>
<td>Passive</td>
</tr>
<tr>
<td>Flexible</td>
<td>Lets the user select the field with a mouse click to move or resize the field</td>
<td>Select</td>
</tr>
<tr>
<td>Resizeable</td>
<td>Lets the user select the field with a mouse click to resize the field (but not move it)</td>
<td>Select</td>
</tr>
<tr>
<td>Moveable</td>
<td>Lets the user select the field with a mouse click to move the field (but not resize it)</td>
<td>Select</td>
</tr>
<tr>
<td>Markable</td>
<td>Lets the user select the field with a mouse click</td>
<td>Select</td>
</tr>
<tr>
<td>ClickPoint</td>
<td>Enables the field to take a ClickPoint event (allowing the code to determine the exact cursor coordinates)</td>
<td>Draw</td>
</tr>
<tr>
<td>DragBox</td>
<td>Lets the user draw a dragbox starting in the field</td>
<td>Draw</td>
</tr>
<tr>
<td>Drag Segment</td>
<td>Lets the user drag a line segment starting in the field</td>
<td>Draw</td>
</tr>
</tbody>
</table>
Biases for Menu Items

OpenROAD provides the following biases for menu items:

**Enabled**
- Specifies that the user can select the menu item (not available for menu separator)

**Disabled**
- Specifies that the menu item is displayed (grayed out) but the user cannot select it or change it

**Invisible**
- Specifies that the menu item is not displayed

The menu item biases let you make certain menu items available to the user under different conditions.

How You Can Set Field and Menu Biases

There are two ways to set the bias of an individual field or menu item from 4GL. The simplest way is to use the CurBias attribute to set the current bias. The alternative way is to set the bias for a particular mode with the ReadBias, QueryBias, UpdateBias, User1Bias, User2Bias, or User3Bias attributes. Use the mode-bias settings only when you are working with frame modes.

The following section describes the CurBias attribute. For a discussion of using the mode-bias attributes, see How You Can Use the Mode-Bias Attributes (see page 372).

How You Can Use the CurBias Attribute

Every field and menu item has a CurBias attribute that lets you specify the current bias for the field. This attribute changes the field bias immediately, regardless of the current frame mode.

The CurBias attribute for fields has the following valid values:

- FB_CHANGEABLE
- FB_LANDABLE
- FB_VISIBLE
- FB_VIEWABLE
- FB_DIMMED
- FB_INVISIBLE
- FB_FLEXIBLE
- FB_RESIZEABLE
- FB_MARKABLE
- FB_MOVEABLE
- FB_CLICKPOINT
- FB_DRAGBOX
- FB_DRAGSEGMENT

If the field does not support the Dimmed bias, OpenROAD uses FB_VISIBLE instead. If the field does not support the Flexible bias (cannot be resized), OpenROAD uses FB_MOVEABLE instead.

The CurBias attribute for menu items has the following valid values:
- MB_INVISIBLE
- MB_DISABLED
- MB_ENABLED

Changing the CurBias attribute also changes the field’s bias for the current mode. For a discussion of using the mode-bias attributes, see How You Can Use the Mode-Bias Attributes (see page 372).

**How You Can Use the CurBias Attribute for Composite Fields**

When you set the CurBias attribute of a composite field, it indirectly affects the biases of its child fields. Based on the new bias of the composite field, OpenROAD sets a maximum bias for the children.

For example, if you change the CurBias attribute for a table field from FB_CHANGEABLE to FB_VISIBLE, the user cannot access any of the columns in the table, even though their individual biases are still FB_CHANGEABLE.

This maximum bias is changed when you reset the CurBias for the composite field. For example, if you reset the CurBias for the table field to FB_CHANGEABLE, the user can then access any of the columns in the table (so long as their individual bias setting allows it).

Changing the CurBias attribute also changes the field’s bias for the current value of CurMode. For example, if the frame’s CurMode setting is FM_READ and you set CurBias for a field to FB_INVISIBLE, the ReadBias attribute for that field is also changed to FB_INVISIBLE.

For more information about changing a field or menu item’s bias, see the *Language Reference Guide* online help.
How You Can Change Frame Modes

When creating a multipurpose application that uses a single frame for more than one task (for example, some users update information while others only examine it), you can use frame modes to change the bias for several or all fields simultaneously.

A frame mode is a name given to a particular set of field and menu biases. For example, the Read mode for a frame might set all fields on the frame to have a bias of Visible, allowing the user to see the fields but not select them.

Frame modes let you switch the biases for many fields and menu items with a single statement. When you set the frame to a particular mode, OpenROAD sets the biases for all the fields and menu items in the frame to the correct bias for that mode.

Every OpenROAD frame has six frame modes, which are described in Frame Modes (see page 370). When you create the frame in OpenROAD Workbench, you specify the initial mode for the frame on the frame's Property Inspector. Then, in the Frame Editor, you can use the Bias menu to set the initial bias of each field on the form for all six frame modes.

OpenROAD provides default biases for all six modes. All but Read mode default to a bias of Changeable. Read mode defaults to a bias of Landable. There is no inherent link between a mode's name and its operations. You can make fields Changeable in Read mode or Landable in Update mode.

The frame mode changes only when explicitly changed in the 4GL code. It is not affected directly by user actions.

You can also change the bias of an individual field for a specific mode. For example, you can use the UpdateBias attribute (a mode-bias attribute) to set the bias of a field to Visible when the frame is in Update mode. After setting the UpdateBias attribute, the next time you set the frame to Update mode, OpenROAD sets the bias of that field to Visible.

Frame Modes

OpenROAD provides the following frame modes:

**FM_UPDATE**

Specifies that default biases for entry fields allow reading and updating. All other fields can be selected (clicked). All menu items are enabled.

**FM_QUERY**

Specifies that default biases for entry fields allow reading and updating. All other fields can be selected (clicked). All menu items are enabled.
How You Can Change Field and Frame Appearance and Behavior

**FM_READ**

Specifies that default biases for entry fields allow reading but not updating. All other fields can be selected (clicked). All menu items are enabled.

**FM_USER1**

Specifies a name that lets you create your own mode (default biases allow reading and updating)

**FM_USER2**

Specifies a name that lets you create your own mode (default biases allow reading and updating)

**FM_USER3**

Specifies a name that lets you create your own mode (default biases allow reading and updating)

Although OpenROAD provides a set of default biases for each of the modes, the most useful modes are the default Update and Read modes. Switching between Update and Read modes is effective when you want to change a frame from an interactive to a passive state. For example, the video_detail frame script sets the frame mode to Read to provide a view-only version of the video details.

Because the Query, User1, User2, and User3 modes have the same defaults as Update mode, these modes are useful only when you want to create your own customized mode by explicitly setting the biases of the individual fields and menu items in that mode. You can do this from OpenROAD Workbench using the Frame Editor’s Bias menu or from your 4GL code with the mode-bias attributes.

When you create a frame with Workbench, OpenROAD uses Update mode as the default mode. You can change this initial frame mode on the frame's Property Inspector. To change the frame mode at runtime, use the CurMode attribute of the frame's FrameExec object.

**How You Can Use the CurMode Attribute**

The CurMode attribute specifies the current mode for a frame. When you set this attribute, OpenROAD changes all fields and menu items on the frame to the appropriate bias for the new mode. The CurMode attribute can be set to any valid mode.

You can force the bias of an individual field or menu item to change when the frame changes modes by setting the mode-bias attribute. However, you need not change frame modes to change the biases of individual fields. The CurBias attribute for fields lets you set the field bias regardless of the frame mode.
How You Can Use the Mode-Bias Attributes

To enable you to change the bias of a field when the frame is in a specific mode, OpenROAD provides the following set of attributes for every field and menu item:

**UpdateBias**
- Sets the bias for the field when the frame is in Update mode

**QueryBias**
- Sets the bias for the field when the frame is in Query mode

**ReadBias**
- Sets the bias for the field when the frame is in Read mode

**User1Bias**
- Sets the bias for the field when the frame is in User1 mode

**User2Bias**
- Sets the bias for the field when the frame is in User2 mode

**User3Bias**
- Sets the bias for the field when the frame is in User3 mode

**AllBias**
- Sets the bias for the field item for all six modes

These attributes let you override the field's initial bias settings specified by using the Frame Editor's Bias menu or using the system defaults. The new setting takes effect the next time you change the frame to the specified mode (immediately if the frame is already in that mode).

How You Can Use the Frame Status Bar

You can enhance an application's ease of use by using the frame status bar. The frame status bar is a read-only text field that contains user-specified text associated either with the frame, or with a specific field.

When the mouse cursor is placed over a field or a menu item, OpenROAD automatically displays the status bar text associated with the field or menu item in the status bar. If the field or menu item has no status text defined for it, OpenROAD places the status text of the first parent of the field or menu item that has status text defined for it. If no fields have status text, or if the mouse cursor is over the frame's form, OpenROAD automatically places the status text associated with the frame in the status bar.
The presence of the frame status bar and its associated informational text can be set in OpenROAD Workbench, or in the 4GL script of the frame. The following code is an example of how the status bar can be used when the mouse is over a particular field:

```
Begin
    curframe.hasstatusbar = true;
    curframe.statustext = 'Your mouse cursor is over the form.';
    field(field0).mousemovetext = 'Your mouse cursor is over field0';
end
```

For more information about how to use the frame status bar, see the attributes of the FrameSource, FrameExec, FormField, and MenuField classes in the Language Reference Guide online help.

---

**How You Can Create and Modify Fields Dynamically**

This section describes how to create and modify fields dynamically.

**How You Can Create Dynamic Simple Fields**

Dynamically creating simple fields (ShapeFields or ScalarFields) is a simple four-step process:

1. Declare a reference pointer to a field of the appropriate type.

   Unless you declare it with default null, the declaration creates an instance of the object. The following code declares and instantiates a free trim object called nextquarter:

   ```
   initialize ()=
   declare
       nextquarter = FreeTrim;
       labelstart = integer not null;
       linechart_subform = Subform;
       i = integer not null;
       name_array = array of StringObject;
       bfield = ButtonField;
       stack_field = StackField;
       efields = EntryField;
       colname = varchar(11) not null;
       dexp = DynExpr default null;
   enddeclare
   ```

2. Create a new instance of the field if necessary.

   The following code creates a new free trim object called nextquarter:

   ```
   nextquarter = FreeTrim.Create();
   ```
3. Set field attributes.

The following code sets attributes for position and point size:

```pascal
nextquarter.AbsXleft = labelstart;
nextquarter.TypeSize = 10;
```

4. Attach the field to the form by giving it a parent. Fields are attached to a form in the following examples:

- The following code adds a dynamically created field directly to the form:

```pascal
nextquarter.ParentField = CurFrame.TopForm;
```

- The following code adds a dynamically created field to a composite field called linechart_subform:

```pascal
linechart_subform = Subform.Create();
nextquarter.ParentField = field(linechart_subform);
```

If the parent field is a MatrixField or Stackfield, the field is placed into a specific cell of the parent with the InsertChild method.

The following code creates multiple buttons, getting their label text from an array of button names, and inserts the buttons into a stack field:

```pascal
for i = 1 to name_array.LastRow do
    bfield = ButtonField.Create();
    field(bfield).TextLabel = name_array[i];
    stack_field.InsertChild(fieldtoinsert = bfield, position = i);
endfor;
```

After the field has been attached to the form, you can reuse the same reference variable to create another field. If you reuse the reference variable, however, you no longer have a direct way to point to your original field and, therefore, have no simple way to manipulate the field’s attributes or data or remove it from the form.

If you want to keep reference variables to all fields you create, use an array. For example, the following code declares an array of EntryField objects and creates each new entry field as a member of this array:

```pascal
efields = Array of EntryField;
...
j = j + 1;
efields[j] = EntryField.Create();
```

The ChildFields attribute of the CompositeField system class stores a reference variable pointing to each field contained in the composite field. The ChildFields attribute often provides the best way to look at all the fields in a composite, although stepping through all the child fields of CurFrame.TopForm looking for a particular field can be awkward.
Because the ChildFields attribute makes it easy to reference the fields in a StackField object, the example for Step 4 uses a single ButtonField reference variable instead of an array.

If you are creating several fields of the same type that share most of their attributes, you can duplicate the first and then change attribute values as appropriate, for example:

```csharp
efields[j+1] = efields[j].Duplicate();
```

If you create an entry field dynamically, its style defaults to multi-line. To allow the user to tab from entry field to entry field, change the dynamically created entry field to single-line by setting the IsMultiLine attribute to FALSE. The following code makes this change:

```csharp
efields[].IsMultiLine = FALSE;
```

**How You Can Create Dynamic Composite Fields**

Composite fields are created dynamically using the same steps used to create a simple field. For a discussion of these steps, see How You Can Create Dynamic Simple Fields (see page 373).

However, creating a MatrixField object dynamically requires an additional step. When first instantiated, matrix fields contain no columns and no rows. Because you cannot insert a field into a matrix field cell that does not exist, you must specify the number of rows and columns before you insert any child fields.

Begin dynamic matrix field creation by specifying a number of rows and columns large enough to contain all the fields to be inserted. After all child fields have been added, remove the empty cells by setting the CollapsePolicy attribute.
The following code creates a matrix field, sets the number of rows and columns, populates the matrix field, and removes empty cells:

```csharp
mfield = MatrixField.Create();
mfield.Rows = 25;
mfield.Columns = 4;

Loop through some array to get information about each field to be added. Create fields and set their attributes.

...  
mfield.InsertChild(fieldtoinsert = trim,  
    row = i, column = 1);
mfield.InsertChild(fieldtoinsert = efields[j],  
    row = i, column = 2);
endfor;
/* get rid of extraneous rows */
mfield.CollapsePolicy = CP_ROWS;
```

For more information about the CollapsePolicy attribute of the MatrixField system class, see the Language Reference Guide online help.

**How You Can Create Dynamic Table Fields and Table Field Columns**

You can create table fields dynamically in the same way as other composite fields, or you can add new columns to an existing table field (so long as its underlying array has not been declared in OpenROAD Workbench).

To add or remove columns dynamically from a table field created in Workbench, the variable for the table field must not be declared until runtime. When you create the table field, set the Variable Declared property to off. At runtime, add or remove the requisite columns. After the table field contains only the desired columns, declare an array variable for the table field with the DeclareData method of the ActiveField system class.

The following section describes the process of dynamically adding columns to a table field. For a discussion of dynamically removing table field columns, see How You Can Remove Columns from a Table Field (see page 379). For a discussion of dynamically declaring the array variable for the table field, see How You Can Declare a Composite Field's Array Variable Dynamically (see page 380).
How You Can Add a Column to a Table Field

You can add a column to an existing table field using the following steps.

1. Create a form field to serve as the prototype field for the column.
   
   Table fields are typically composed of entry fields, but the prototype can be any form field.
   
   The following code instantiates an entry field object as a member of an array named efields:
   
   ```
   efields[j] = EntryField.Create();
   ```
   
2. Assign appropriate attributes to the field.
   
   You must give the field a name. If you do not supply a data type for an entry field, OpenROAD provides a default of varchar(100).
   
   The following example sets attributes for an entry field to contain dates:
   
   ```
   efields = EntryField.Create();
   colname = 'some_string';
   efields.Name = colname;
   efields.DataType = 'date';
   efields.FormatString = 'd"2/3/01"';
   ```
   
3. Insert the field into the table field, for example:
   
   ```
   field(tfield).InsertColumn(fieldtoinsert = efields[j], position = j, title = colname);
   ```
   
   If position is not specified, the column is inserted as the first column of the table field. The mandatory title parameter specifies the text for that column's title.
   
   Inserting a field into a table field creates a ColumnField. The reference variable pointing to the column field is stored in the tblfld.TableBody.ChildFields array.
   
4. To change other attributes of the column's title, such as type size, reference the TitleTrim attribute of the ColumnField.
   
   The following code changes the column's font style and size:
   
   ```
   ColumnField(tfield.TableBody.ChildFields[j]).TitleTrim.IsBold = TRUE;
   ColumnField(tfield.TableBody.ChildFields[j]).TitleTrim.TypeSize = 10;
   ```
   
   You must cast a TableBody's child field to a ColumnField. For more information about casting, see How You Can Work with Attributes (see page 140).
After all columns have been added to the table field, create an array variable for the table field using the DeclareData method.

For a discussion of dynamically declaring the array variable for the table field, see How You Can Declare a Composite Field's Array Variable Dynamically (see page 380).

How You Can Remove Fields Dynamically

To remove a field from the form while its frame is running, set the field's parent to null. The technique used to set the parent depends on whether the field was created originally with a variable declared or was created without declaring its variable.

How You Can Remove a Field from Display

To remove a field that was created originally in OpenROAD Workbench with a declared variable (the default behavior), you can use the field function to reference the field when setting its parent to null. The following statement removes a field created in Workbench:

```java
field(myfield).ParentField = null;
```

Fields that are created dynamically, however, cannot be referenced using the field function. You must refer to these fields with the reference variable used to create them. Because the field function operates on a data variable to return its reference variable, it is irrelevant to dynamically created fields. Fields created dynamically do not have a data variable.

The following statement removes a dynamically created table field named tfield from display:

```java
tfield.ParentField = null;
```

Because every field that is part of a composite field has a reference variable pointing to it in the composite's ChildFields array, you can remove all the fields in a composite by clearing that array. The following statement removes all fields in the mfield matrix field:

```java
mfield.ChildFields.Clear();
```
How You Can Restore a Field to Display

After you set an object’s parent to null, the object still exists. The object’s attributes (such as size and location) are unchanged. You can, therefore, restore the object to the form simply by giving it a parent. Both of the following statements put the myfield field back onto the current form:

```plaintext
field(myfield).ParentField = CurFrame.TopForm;
frmfield.ParentField = CurFrame.TopForm;
```

If you want to place the field in a new location or change any other attributes, you can do this either before or after giving it a parent.

When you set attributes of a dynamically created field, OpenROAD might set other attributes implicitly based on the values you set. For example, OpenROAD uses the point size specified for a field’s TypeSize as well as the length of the string provided as its TextValue to set the Width attribute of the field. To perform calculations based on implicitly set attributes, first attach the field to a parent.

For example, assume you want to determine the width of a dynamic field to center it horizontally in the window. You must parent the field before you attempt to center it. The `trimfield.ParentField = CurFrame.TopForm` statement in the following example attaches the field to the form before the statement below it, which centers the field on the form:

```plaintext
trimfield = FreeTrim.Create();
trimfield.TypeSize = 14;
trimfield.TextValue = ‘Frame Title’;
trimfield.ParentField = CurFrame.TopForm;
trimfield.AbsXLeft = (CurFrame.WindowWidth - trimfield.Width) / 2;
```

If the parenting statement had followed the statement that centers the new field based on its width, the field would not be positioned correctly.

How You Can Remove Columns from a Table Field

To remove a column from a table field, use the DeleteColumn method. For example, the following statement deletes the second column from a table field named `tblfld`:

```plaintext
tblfld.DeleteColumn(position = 2);
```
Because table fields must always contain at least one column, you can never delete all columns from a table field. Therefore, replacing all of a table field's columns with another set of columns requires the following steps:

1. Delete all but one column.
2. Add the new columns.
3. Remove the last of the original columns.

The following code performs these operations:

```pascal
for i = tfield.TableBody.ChildFields.LastRow downto 2 do tfield.DeleteColumn(position = i);
endfor;
insert new columns, beginning with position 1
insert new columns, beginning with position 1
tfield.DeleteColumn(position =
tfield.DeleteColumn(position =
    tfield.TableBody.ChildFields.LastRow);
```

## How You Can Declare a Composite Field's Array Variable Dynamically

To change the fields in a composite field or the columns in a table field dynamically at runtime, the composite field (including table fields) must have been created without a variable declared. Only after the composite field has been modified to contain the required fields or columns should you use the DeclareData method to declare a user class or array variable for it.

Moreover, to change the fields of a composite field or the columns of a table field whose user class or array variable has been dynamically declared, use the UnDeclareData method to remove the current user class or array before changing the fields or columns and declaring a new user class or array.

The following sections describe dynamically declaring and undeclaring an array variable for a table field.

## How You Can Declare and Undeclare Data

Before you can load data into a dynamically created or modified table field, perform the following operations:

1. Attach the table field to the form by parenting it (if it is not already parented).
2. Specify the table field's underlying array with the DeclareData method.
The following code declares an array variable (tarray), attaches the tfield table field to the current form, and declares tarray as the array associated with the tfield table field:

```plaintext
Declare
tarray = ArrayObject default null;
enddeclare
```

```plaintext
tfield.ParentField = CurFrame.Topform;
tfield.DeclareData(result = byref(tarray));
```

In the process of declaring the table field's array variable, the DeclareData method creates a user class whose attributes match the columns in the associated table field. Unless you explicitly set the data type of the table field, after invoking the DeclareData method, the named array points to an initially empty array whose rows are objects of the user class created by the method.

In addition to creating a user class for the table field and creating a new array whose rows' class is the new user class, the DeclareData method adds the name of the table field to the scope of the frame to which it belongs. In this example, the scope is the current frame.

Because the table field's name has been added to the current frame's scope, that name can be used subsequently in dynamic expressions to refer to the array associated with the table field. Moreover, dynamic expressions can also refer to columns of the rows in the table field's array.

**Note:** Because the user class is constructed to match the table field's columns, invoke the DeclareData method only after you have added or removed all the desired columns and given the table field a parent.

### How You Can Change a Dynamic Table Field after Declaring Data

After declaring data, you can change a dynamic table field, adding or removing columns, using the following steps:

1. Set its array to null.
2. Undeclare its data.

The following code disassociates a target array from the qo query object, disassociates any particular class from the tarray array, and disassociates any array from the tfield table field:

```plaintext
qo.TargetArray = null;
tarray = null;
status = tfield.UnDeclareData();
if status != ER_OK then
    ...
endif;
```
If data has not been declared, the UnDeclareData method returns an error (nonzero value). Similarly, invoking the DeclareData method on an object for which data has already been declared (either a variable was declared for the table field in OpenROAD Workbench or with the DeclareData method) returns an error. Moreover, if a variable was declared for the table field in Workbench, you cannot later undeclare and redefine the data.

How You Can Manipulate Data in Dynamic Fields

If a field was created with a declared variable in OpenROAD Workbench, you access the field's data value by using the name of the field as a variable.

For example, assume that salary is the name of a field with data type money and dollars is a variable declared with the same data type. The following statements are valid:

```
salary = salary * 1.15;
dollars = salary;
salary = dollars + 5.25;
```

You cannot perform these calculations as easily when working with a dynamically created field. To access the value of a dynamically created field, you must invoke a method on the field or use a dynamic expression (DynExpr).

For a discussion of accessing values in dynamically created fields, see How You Can Set and Get Simple Field Values (see page 382) and How You Can Access Individual Cells of a Dynamic Table Field (see page 385).

How You Can Set and Get Simple Field Values

OpenROAD provides the following methods for manipulating data in simple dynamic fields:

- SetFieldValue
- GetFieldValue

For example, assume that the field is a dynamic entry field whose Name attribute has been set to salary. Moreover, assume you want to perform a simple calculation, equivalent to the following statement's multiplication of the value in a statically created field named salary:

```
salary = salary * 1.15;
```
The following lines of code perform the same operation as the previous one line of code:

```csharp
status = efield.GetFieldValue(value =
    byref(dollars));
dollars = dollars * 1.15;
status = efield.SetFieldValue(value = dollars);
```

The first statement gets the value in the efield field and loads it into the dollars variable. The second line increases the value in the dollars variable by 15 percent. The last line sets the value of the efield field with the current value of the dollars variable, thus increasing the value of a field (efield) named salary by 15 percent.

The integer variable status in this example is set to ER_OK if the method is successful. If not, status contains a value indicating the error condition. A typical source of error is data type incompatibility between the field and the variable to which its value is being assigned.

You can access other attributes of the dynamic field by using the reference variable pointing to the field (in the same manner as accessing attributes of statically created fields). For example, you can access the character string displayed in an entry field by using the TextValue attribute, which contains the string representing the characters that the user or program has entered into the field.

In the case of a varchar field, the value of the TextValue attribute is equivalent to the field's value. For any other kind of field, such as an integer or money field, you must use a conversion function to obtain the field's value from the TextValue attribute.

For example, the following code converts the value of a TextValue attribute to money, multiplies it by 1.5, and puts it into the dollars variable:

```csharp
dollars = money(efield.TextValue) * 1.5;
```

The following code converts the money value stored in the dollars variable to varchar and puts it into the TextValue attribute of the efield field:

```csharp
efield.TextValue = varchar(dollars);
```

**Note:** If the user erroneously enters characters that are incompatible with a field's data type, using type conversion functions in this way causes errors that may be difficult to trace.
How You Can Populate a Dynamic Table Field

To populate a dynamic table field from a database table, use an SQLSelect object or a QueryObject object and point the target array of the query object to the array you declared for the table field.

For example, the DynamicQuery frame uses a query object to load a dynamically created table field at runtime. The code in this frame that retrieves the data from the database and displays it in a table field performs the following operations:

- Declares an array variable called tarray
- Establishes the values in the tarray array as:
  - The array underlying the tbl table field
  - The query object's TargetArray
- Opens a query object named qo in array mode, retrieving the data from the database
- Closes the query object
- Updates the form display, forcing display of data in tbl

The following code performs these operations:

```
declare
tarray = ArrayObject;
enddeclare
... qo.TargetArray = tarray;
... tfield.DeclareData(result = byref(tarray));
... status = qo.Open(QueryMode = QY_ARRAY);
qo.Close();
field(tbl).UpdField();
```

Opening a query object in QY_ARRAY mode retrieves all the data directly from the database into the named array during the Open method invocation. (The Load and NextRow methods are not required to display the data on the form.) Because array mode eliminates intermediate steps in retrieving and displaying data, performance is better using array mode rather than cache, cursor, or direct modes.

For array mode, the class of the array being loaded must have attribute names matching the names of the columns in the query (the array attributes must match all columns whose IsSelectTarget attribute has been set to TRUE).

For more information about using SQLSelect and QueryObject objects, see Working with a Database (see page 147).
How You Can Access Individual Cells of a Dynamic Table Field

You cannot use the GetFieldValue and SetFieldValue methods to obtain or change data values in cells in a dynamic table field. Use the following procedure to access the values in an array underlying a dynamic table field.

To access the values in an array underlying a dynamic table field, you would perform the following basic steps:

1. Create a dynamic expression—an object of the DynExpr system class that represents an expression that can be reevaluated dynamically or a variable that can be reevaluated or assigned dynamically.

2. Use one of the following methods, defined for the DynExpr system class, to get or set values:

   - **Assign**
     
     Assigns the value of one dynamic expression to another

   - **GetValue**
     
     Puts the value of the field referenced by the dynamic expression into a variable

   - **SetValue**
     
     Puts the value of an expression into the field referenced by the dynamic expression

How You Can Create a Dynamic Expression

You create DynExpr objects by invoking the CreateDynExpr method on an object of the Scope class. The Scope object represents the context for evaluation of the DynExpr object. The dynamic expression is no longer valid when its scope terminates, that is, when its associated application, frame, or other component, closes.

The CreateDynExpr method takes the following parameters:

- **string**
  
  Contains any syntactically valid OpenROAD expression except procedure or method invocations. Use the string parameter to specify the field or table field cell whose value you want to access. This parameter (which takes varchar values) can reference the field directly or can contain a valid OpenROAD expression that references the field.

  The value of the string parameter is loaded into the dynamic expression created by the CreateDynExpr method. You can then use the dynamic expression to access the value of the field or cell originally specified.
For example, the following statement creates a dynamic expression (dexp) that contains the value of a cell in the Salary column of a table field named tbl:

```plaintext
declare
dexp = DynExpr default null;
enddeclare

dexp = CurFrame.Scope.CreateDynExpr(string = 'tbl[i].Salary');
```

When the dynamic expression is used in a GetValue, SetValue, or Assign method invocation, the value of i must be a valid row number for the array.

The value of the string parameter is evaluated within the context of the named scope. In the previous statement, the dexp dynamic expression is valid while the current frame is running. The scope in the following statement limits the dynamic expression to the current event block:

```plaintext
declare
dexp = DynExpr default null;
enddeclare

dexp = CurEventScope.CreateDynExpr(string = 'tbl[i].Salary');
```

Any attempt to use the dynamic expression after the event block terminates causes a runtime error.

**errors**

(Optional.) Provides the text of the error message. If the scope object on which the CreateDynExpr method is invoked is invalid, or the text specified in the string parameter cannot be compiled, the CreateDynExpr method returns a null.

If compilation errors occur, the CreateDynExpr method creates a string object containing the errors and sets the variable specified by the errors parameter (of StringObject data type) to point to that string object. If no compilation errors occur, this parameter is set to null.

The following statement creates a dynamic expression and sets the errors parameter:

```plaintext
declare
dexp = DynExpr default null;
enddeclare

dexp = CurFrame.Scope.CreateDynExpr(string = 'tbl[i].Salary', errors = byref(stringvar);
```

If an error occurs creating the dexp dynamic expression, the stringvar variable contains the error message in its Value attribute.
How You Can Set and Get Values with Dynamic Expressions

You can use the following methods to get and set values in the field referenced by a dynamic expression:

- `GetValue` to obtain the value from the field
- `SetValue` and `Assign` to change values in the field

The `GetValue` method reevaluates the expression that was specified when the DynExpr object was created and places the value of the dynamic expression in the specified variable.

For example, to get the value of a dynamic expression that was created for a field whose data type is money, create a money variable and invoke DynExpr's `GetValue` method to get the value from the dynamic expression and load it into the money variable.

The following statement performs this operation:

```
status = dexp.GetValue(value = byref(money_var));
```

If the `GetValue` method succeeds, `status` is set to `ER_OK`; if it fails, it is set to some other integer value.

Setting a value in a dynamic expression works similarly. Use DynExpr's `SetValue` method to load a value from a valid OpenROAD expression into a dynamic expression.

The following code sets the value of the dynamic expression from a variable parameter to the `SetValue` method:

```
status = dexp.SetValue(value = money_var);
```

Like the `GetValue` method, the `SetValue` method sets `status` to `ER_OK` if successful; if it fails, it sets `status` to some other integer value.

The `SetValue` method is valid only if the expression specified when the dynamic expression was created was not a computed expression. The `GetValue` method is valid only if the expression specified when the dynamic expression was created represents a field or table field cell.

You can get a value from a dynamic expression and assign a new one in a single step using the `Assign` method. For an example of using the `Assign` method, see How You Can Get Values from Unknown Fields (see page 388).
How You Can Get Values from Unknown Fields

The previous examples of getting and setting values with dynamic expressions assume you know the name and data type of the current table field column. Assume instead that you want to get a value from the current cell of a dynamic table field named tbl but you do not know which cell currently has the input focus. Assume also that all columns in the table have a data type of integer or varchar.

When you invoke the GetValue method, you must put the value obtained into a variable of the appropriate data type. Therefore, getting the value from the current cell requires using a conditional statement as well as creating a dynamic expression and invoking the GetValue method.

The following field script performs these operations when the user enters a cell in a table field:

```plaintext
declare
txt = varchar(50) not null;
dexp = DynExpr default null;
errstr = StringObject default null;
intvar = integer not null;
textvar = varchar(50) not null;
status = integer not null;
enddeclare
on childentry =
begin
  txt = 'tbl[].' + CurFrame.TriggerField.Name;
dexp = CurFrame.Scope.CreateDynExpr
    (string = txt, errors = byref(errstr));
  if CurFrame.TriggerField.DataType = 'integer'
    then status = dexp.GetValue(value =
                               byref(intvar));
  else
    status = dexp.GetValue
      (value = byref(textvar));
  endif;
  ...
end
```

After declaring the appropriate variables, this code uses the Name attribute defined for the FieldObject system class to create a text string that contains the full name of the current table field cell.

If the expression in the txt variable is valid (that is, points to an existing field and performs valid operations), the errstr variable is null after the statement creating this dynamic expression; otherwise, it contains the text of the error message.
The following code assigns a new value to the same cell in tbl, using the same dynamic expression:

```plaintext
intvar = intvar * 1.15;
status = dexp.SetValue(value = intvar);
```

You can get a value from the cell and assign a new one in a single step using the Assign method. Because the Assign method assigns the value of one dynamic expression to another, you can use this method to copy values between dynamic expressions without knowing the data type of the value in each expression.

For example, the following code creates two dynamic expressions, one containing the current value of the specified cell and the other containing that value increased by 15 percent, and then assigns the value of the second dynamic expression to the first:

```plaintext
txt = 'tbl[i].' + CurFrame.TriggerField.Name;
dexp_a = CurFrame.Scope.CreateDynExpr(string = txt,
   errors = byref(errstr));
txt = txt + '* 1.15';
dexp_b = CurFrame.Scope.CreateDynExpr(string = txt,
   errors = byref(errstr));
status = dexp_a.Assign(fromdynexpr = dexp_b);
```

Although using the Assign method in this manner requires creating two dynamic expressions, it eliminates the need for an intermediate variable.

---

**How You Can Manage Operations with a Control Button**

If you want a Control Button to manage operations on the dynamic table field, create the control button in OpenROAD Workbench as an independent object. Dynamically created table fields, as well as table fields created in Workbench but changed dynamically, cannot use the script supplied with a control button.

For the script of a table field’s control button to be available to a running frame, the script must be compiled before the frame runs. If the table field is created with its variable declared (Variable Declared property set to on), its control button’s script is compiled. However, table fields created without declaring the variable (a requirement of dynamically created or modified table fields) are created with empty control button scripts.

Even moving the control button’s script into the frame script does not make the script available to the running frame if the table field has been created without declaring its variable.
Chapter 15: Creating a Frame at Runtime

This section contains the following topics:

- **Conceptual Background** (see page 391)
- **How You Can Build a Frame Dynamically** (see page 392)
- **Dynamic Statements Considerations** (see page 418)

This chapter describes how to build a frame at runtime, illustrating the following concepts and procedures:

- Creating a frame and setting its attributes
- Creating the frame's form and fields
- Creating the frame's menu
- Creating the frame's script
- Creating the FrameSource object
- Attaching the frame to the application
- Compiling and running the frame
- Dynamic statement considerations

The example frame created in this chapter lets the user select a database table and displays the table columns as fields on a form. The DynamicFrame frame creates every aspect of the frame from scratch. The DynamicFrame frame also differs from the DynamicQuery frame in that it displays selected data in simple fields.

For a discussion of changing the behavior of a frame or field at runtime by changing the setting of frame modes and field biases, creating a composite field dynamically, and modifying a table field dynamically, see Creating Dynamic Frames (see page 363).

For a discussion of creating the lines in a line chart dynamically based on data in the database, see the User Guide.

**Conceptual Background**

*Dynamic frames* are frames that perform operations based on decisions made by the user at runtime. In a dynamic database frame, you can let users decide which table to access or to execute an SQL statement that they compose. Dynamic frames can also create objects at runtime. With a dynamic frame, you can build any object that you can create with OpenROAD Workbench.
Dynamic database frames are often a useful way to provide a generic frame, that is, a frame that a user can run against any table in the database. For example, a library could give patrons the ability to search online tables of authors, titles, or subjects. Writing a separate frame for each possible table would be inefficient. Much more efficient is writing one general frame to run against any table in the library’s database.

The *DynamicFrame frame* at runtime displays the starting frame that provides the user with a choice of database tables. The starting frame then builds a second frame that presents the columns of the chosen table as simple fields on a form. The second frame contains menu items that let the user update values in the current row, delete the row, or request the next row. A sample dynamic application illustrates this process.

How You Can Build a Frame Dynamically

When you build a frame dynamically, you must do in 4GL the tasks that you (or OpenROAD) would do if you were using OpenROAD Workbench to create a frame, which include:

- Create a frame and set its attributes
- Create the form associated with the frame
- Create the fields on the form and set their attributes
- Create the frame's menu
- Create the frame's script

The preceding tasks require you to create several objects, for example, a frame, a form, and a menu. To create an object dynamically, use the Create method. Executing the Create method returns an object of the specified class.

The syntax of the Create method is:

```
refvariable = Class.Create()
```

- **refvariable**
  Specifies a return variable that must have been declared as the same class as `Class` or one of its superclasses. For example, if you want to create an entry field, you must first declare the reference variable that points to the object:

  ```
  field_ptr = EntryField;
  ```

  Then you can use the following statement to create the actual object:

  ```
  field_ptr = EntryField.Create();
  ```

- **Class**
  Specifies the class of the object that you want to create
How You Can Create the Frame and Set Attributes

To create a frame dynamically, you must first create a FrameSource object for the frame. The FrameSource object contains a frame's source definition, its initial settings when it is called or opened. You must first declare the reference variable that points to the FrameSource object. For example:

test_frame = FrameSource;

A FrameSource has the following attributes that you must set to run the frame:

- **TopForm**
  - Contains the form associated with the frame
- **Script**
  - Contains the frame's script

In a dynamic context, setting these attributes means that you must construct the form and the script in your 4GL code. For a discussion of these procedures, see How You Can Create the Form (see page 393) and How You Can Create the Script (see page 395).

If the frame you are constructing has a menu, you must construct the menu (another attribute of the frame source) in your 4GL code. For a discussion of this procedure, see How You Can Create the Menu (see page 394).

Other frame source attributes that you can set are:

- **DataType**
  - Specifies the data type of the frame's return value
- **IsNullable**
  - Specifies whether the return value data type is nullable

How You Can Create the Form

The form that a frame displays is an object of the FrameForm class. A FrameForm object is a special case of a subform that encloses the full set of FormField objects in a frame.

After you have created the frame source, create the frame's form and assign it to the TopForm attribute of the frame source. For example:

test_frame.TopForm = FrameForm.Create();

At this point, your form is a blank form, without fields. The next step is to create the fields.
How You Can Build a Frame Dynamically

How You Can Create the Fields

To create each field, use the Create method to create the specific field object and then set its required attributes. The attributes that you set depend on the type of field you are constructing. In addition, if you want the field to have a title on the form, you must construct the field trim as well.

If your fields correspond to columns in a database table, as do the fields in the example application, retrieve the column name, length, and data type information from the database to create the fields. You can retrieve this information in any of the following ways:

- A select loop
- An SQLSelect object
- A QueryObject object

The DynamicFrame frame uses a select loop to retrieve information for building the columns. For an example of using a select loop to create fields dynamically, see How You Can Construct the Fields (see page 405). For an example of using a query object to retrieve this information, see Creating Dynamic Frames (see page 363).

How You Can Attach Fields to the Form

After a field is constructed, attach the field to the form by setting the field's ParentField attribute. The data type of ParentField is CompositeField. Setting a field's ParentField attribute to the name of a reference variable, pointing to a form, attaches the field to that form.

For example, the following statement attaches the field represented by field_ptr to the form represented by test_frame.TopForm:

```c
field_ptr.ParentField = test_frame.TopForm;
```

The preceding statement says that the parent field of the field identified by field_ptr is the form identified by test_frame.TopForm.

How You Can Create the Menu

A frame's menu is a MenuBar object displayed across the top of the window, containing a single line of menu selections. The MenuBar object is contained in the StartMenu attribute of the FrameSource object.

MenuBar objects contain menu groups, which can contain other menu groups, individual menu items, and menu separators. Menu groups allow a variety of menu structures beneath the top level of the menu bar.
To create a menu for your frame, you would perform the following basic steps:

1. Create the MenuBar object, assigning it to the StartMenu attribute of the FrameSource.

2. Create the menu groups that make up the menu bar. For each:
   - Create the MenuGroup object.
   - Define its TextLabel attribute. This becomes the initial menu item that appears on the menu bar.
     For example, assume that you create two menu groups and assign the value File to the TextValue attribute of one and Edit to the TextValue attribute of the other. After you attach these menu groups to a menu bar, the menu bar presents the selections File and Edit to the user.
   - Create the menu objects that make up the menu group, set their attributes, and attach them to the menu group.

3. Attach the completed menu group to the menu bar.
   To attach a menu object to a menu group or a menu group to the menu bar, set the object's ParentMenuGroup attribute. Setting this attribute attaches the specified MenuGroup, MenuItem, or MenuSeparator to a specified menu.
   For example, the DynamicFrame frame uses the reference variable test_menu to hold the individual MenuButton objects as they are created. After they have been created and named, they are attached to the MenuGroup object represented by top_menu:
   
   ```
   test_menu.ParentMenuGroup = top_menu;
   ```
   In effect, the preceding statement says that the menu button represented by test_menu is part of the menu group represented by top_menu. For an example of how to create menu items dynamically, see How You Can Create the Frame Menu Items (see page 411).

**How You Can Create the Script**

A frame's script is contained in the Script attribute of its FrameSource object. The data type of this attribute is StringObject. The first step to building a script for a frame is creating a string object and assigning it to the Script attribute of the frame's FrameSource object.

After you have created the string object, use the ConcatVarchar method to put the text into your script. This method appends text to a StringObject. The syntax of the ConcatVarchar method is:

```
StringObject = StringObject.ConcatVarchar(text = varchar)
```

The ConcatVarchar method returns a reference to the original string.
If your script includes statements that access the database, such as select or update, you can use a select statement to retrieve the necessary column names for the target lists of the statements and a select loop to build at least part of the database statements.

Alternatively, you can use a query object to create the target fields and provide the user with the ability to change the data in the database. For an example of using a query object to perform these operations, see Creating Dynamic Frames (see page 363).

The DynamicFrame frame uses a select loop to obtain information and build the target fields and a cursor object to allow the user to make changes to data in the database. For details about the implementation in the DynamicFrame frame, see How You Can Construct the Fields (see page 405) and How You Can Complete the SQL Statements (see page 411).

How You Can Attach the Frame to the Application

After you have completed all of the frame elements and attached them to the frame, you must attach the frame itself to the application by setting the ParentApplication attribute of the new frame's FrameSource object. The value to which you set this attribute must be the same value that is in the ParentApplication attribute of the currently executing frame. (The currently executing frame is a frame of your dynamic application.)

To access the ParentApplication attribute of a currently active frame, use the ObjectSource attribute of the frame's FrameExec object. Whenever you call or open a frame, OpenROAD creates a FrameExec object for that instance of opening or calling. The FrameExec object gives you information about the current state of a running frame. The ObjectSource attribute points to the frame's FrameSource object. The ParentApplication attribute is defined for the FrameSource object.

For example, the DynamicFrame frame attaches the generated test_frame frame to the currently executing frame (the starting frame) by assigning the value of the ParentApplication attribute of the current frame's FrameSource object to the corresponding attribute of the generated frame's FrameSource object:

```javascript
    test_frame.ParentApplication =
    CurFrame.ObjectSource.ParentApplication;
```

For an example of attaching a frame to an application, see How You Can Add the Frame to the Application (see page 416).
How You Can Compile and Run the New Frame

This section presents a sample dynamic application that contains two frames, the starting frame and the dynamic frame:

- The starting frame

The starting frame is created in OpenROAD Workbench. This frame contains one option field (providing a list of tables) and one menu item, File (with Go and Close submenu options). The following figure illustrates the dynamic application's starting frame:

![Starting Frame Diagram]

- The dynamic frame

The frame called by the starting frame is built dynamically and contains one field for every column in the selected table. This frame contains a File menu option (with two submenu options: Commit and Close, and Rollback and Close) and an Edit menu option (with Delete, Update, and Next submenu options). The generated frame lets the user edit the data in the fields, save or cancel changes, or request the next row of retrieved data. The following illustrates the dynamic application's generated frame:

![Dynamic Frame Diagram]

All of the application's dynamic work is done in the frame script associated with the starting frame. Each of the following subsections describe a portion of the starting frame's script and the role that it plays in the dynamic application.
How You Can Build a Frame Dynamically

Initialize Statement

The example application uses the DynamicFrame frame's initialize statement to perform the following start-up tasks:

- Declare necessary variables
- Select and list the tables from which the user can choose
- Initialize some variables

How You Can Declare Variables

The DynamicFrame frame declares the following local variables in its frame script's initialize block:

```sql
initialize () =
declare
/* Needed for loading the option list */
i = integer not null; /* temp index */
/* Needed in generation of frame */
numcolumns = integer not null;
/* number of columns in tab */
widthchar = integer not null;
/* width of one character */
heightchar = integer not null;
/* height of one character */
fieldleft = integer not null;
/* left edge of field */
trimleft = integer not null;
/* left edge of trim */
fieldtop = integer not null;
/* top of next field */
fieldheight = integer not null;
/* height of a field */
maxformheight = integer not null;
/* maximum height of form */
maxformwidth = integer not null;
/* fixed width of form */
vertspace = integer not null;
/* space between lines */
trimspace = integer not null;
/* space between trim/field */
dbaname = varchar(32) not null;
/* name of dba */
```
How You Can Build a Frame Dynamically

/* variables used for generating the source code */
/* select statement string */
selectstring = varchar(2000) not null;
/* FETCH statement string */
fetchstring = varchar(2000) not null;
/* UPDATE statement string */
updatestring = varchar(2000) not null;
/* update clause string */
selectupdatestring = varchar(2000) not null;
/* Variables for dictionary information of the table columns used in select choices */
tmp_table_name = varchar(32) not null;
/* used in getting column names */
columnname = varchar(32) not null;
/* used in getting columns */
columndatatype = varchar(32) not null;
/* used in getting columns */
columnlength = integer not null;

/* Variables used as holders of form objects, etc. */
test_frame = FrameSource default null;
/* The frame being generated */
top_menu = MenuGroup default null;
/* The menu being generated */
test_menu = MenuButton default null;
/* The menu item */
field_ptr = EntryField default null;
/* Ptr to entry field */
trim_ptr = FreeTrim default null;
/* The trim ptr */

/* Name of generated frame */
test_frame_name = varchar(32) not null
enddeclare

How You Can Select Tables for the Option Field

The starting frame lets the user pick a table from a list of tables presented in an option field. The initialize block contains the code that fills in the list of table choices. The DynamicFrame frame lets the user select from any table owned by the user or the database's DBA (which are available to any user with the necessary permissions).

The following statements from the sample application retrieve the name of the database's DBA from the iidbconstants catalog and release the lock taken by the select statement:

/* Get the dba name for finding the tables */
select dba_name as :dbaname
from iidbconstants;
commit;

*
**Note:** The table iidbconstants is a view in the Ingres Standard Catalog Interface that has two columns. One column holds the name of the current user and the other holds the name of the DBA of the current database.

Before loading the table names selected at runtime into the option field, the starting frame clears any previous values from the option field. This step is necessary because you must give a value to the option field when you create it in OpenROAD Workbench. The following statement clears the option field in the example frame:

```java
```

ValueList is the attribute of the OptionField class that contains the values that appear in the list. Its data type is ChoiceList. One attribute of ChoiceList is ChoiceItems, which is an array of class ChoiceItem. It is the ChoiceItems array that actually contains the values displayed in the option field. For a more detailed explanation of loading values into the option field, see How You Can Load Values into a ChoiceField Object (see page 401).

After clearing the option field of values, the starting frame uses a select loop to retrieve the new values to be loaded into the option field. The retrieved values are the names of all the tables that are owned by the DBA. The table names are retrieved from iitables (an Ingres Standard Catalog Interface view). The selected tables are returned one at a time and processed by the select loop. The select loop places each table name into the option field’s ValueList.

The following code is an example of the select loop discussed in the previous paragraph:

```java
i = 1;
select distinct table_name as :tmp_table_name
  from iitables
  where table_owner = :dbaname
begin
  field(table_choices).ValueList.ChoiceItems[i]
  EnumText = tmp_table_name;
  field(table_choices).ValueList.ChoiceItems[i]
  EnumValue = i;
  i = i + 1;
end;
```

The following section, How You Can Load Values into a ChoiceField Object (see page 401), explains the syntax used to load values into the table_choices option field.
How You Can Load Values into a ChoiceField Object

Because the statement that loads values into the table_choices option field uses the reference variable associated with the option field, the field function is required. For more information about the field function and its use, see Working with Classes (see page 113) and Creating Dynamic Frames (see page 363).

As a ChoiceField object, one of the attributes of the table_choices option field is ValueList. ValueList contains the values in an enumerated list. The data type of ValueList is ChoiceList, which means that ValueList contains an object of the class ChoiceList.

A ChoiceList object has an attribute called ChoiceItems. The ChoiceItems attribute contains an array of class ChoiceItem with two attributes, EnumText and EnumValue. The EnumText attribute contains the text associated with the list; in the example frame, the value of EnumText is the table name. The EnumValue attribute contains an enumerated value associated with the text; the example frame uses the row number of the ChoiceItems array.

To access the EnumText and EnumValue attributes, you can move through the ChoiceItems array by incrementing the array index counter.

The following statement assigns the contents of the variable tmp_table_name to the EnumText attribute in the ith row of the array of ChoiceItem objects contained in the ChoiceItems attribute:

```javascript
field(table_choices).ValueList.ChoiceItems[i].EnumText = tmp_table_name;
```

The ChoiceItems attribute is an attribute of the ChoiceList object that is contained in the ValueList attribute of the OptionField object represented by field(table_choices).

An alternative method for loading the ValueList attribute of a ChoiceField object is using the ChoiceList's AddTextItem or the AddBitmapItem methods. For an example of how to use the AddBitmapItem method, see Working with Images and Text Strings (see page 277).

How You Can Display the List

When the user opens the starting frame, the frame's initialize statement executes, and the retrieved table names are displayed automatically in the option field. To change the list after the application has started, you must explicitly refresh the option field.
When a fixed-choice list (for example, table_choices) is currently displayed, OpenROAD does not automatically refresh the display after you make changes to the option field's ValueList. You must explicitly refresh the display with ChoiceField's UpdChoiceList method. The following statement from the example application refreshes the display of the table_choices option field:

```plaintext
field(table_choices).UpdChoiceList();
```

### How You Can Initialize Variables

After loading appropriate table names into the option field, the example starting frame initializes all variables whose values remain the same throughout the program. The following code from the starting frame's initialize block initializes variables:

```plaintext
/* Set up initial values. */
widthchar = 100;
heightchar = 200;
trimleft = 100;
trimspace = 100;
maxformheight = 3000;
maxformwidth = 9000;
fieldleft = trimleft + (33*widthchar) + trimspace;
vertspace = 250;
```

All values refer to one-thousandths of an inch.

### The Go Menu Item (Constructing the Frame)

After selecting a table, the user selects the Go menu item. Because the example application lets the user return to the initial frame to select another table, the code for the Go operation checks whether a frame already exists for the chosen table before generating a frame for that table. If such a frame exists, it executes the frame.

The following code from the example starting frame first checks whether the required frame exists. Then it executes the frame, if it does exist, or generates the frame and then executes it if it does not exist:

```plaintext
if Generated_Frame_List.LastRow >0 then
    /* There is at least one generated frame. */
    for i = 1 to Generated_Frame_List.LastRow do
        if Generated_Frame_List[i].table_name =
            table_choices then
            /* Frame already there. Execute it. */
            callframe
              :Generated_Frame_List[i].frame_name;
            resume;
            endif;
        i = i + 1;
    endfor;
endif;
```
Generated_Frame_List is a global array variable used by the example application to keep track of frames that have already been generated, avoiding the need to regenerate them. The LastRow attribute, defined for array objects, returns the sequence number of the last row in the array. If the LastRow attribute returns any positive number in the preceding code, the code checks the Generated_Frame_List array for a table name that matches the user's chosen table. If a match is found, that frame is executed.

If no rows exist in the Generated_Frame_List variable or if no match is found, the starting frame code constructs a frame. The construction proceeds in the following order:

1. Create the FrameSource object and define its attributes.
2. Create the FrameForm object.
3. Construct the fields on the form.
4. Set the height and width of the form.
5. Complete the statement strings.
6. Create the menu items.
7. Generate the frame script for the frame.
8. Add the frame to the application.

After the frame is generated, it can be executed.

The following subsections describe each of these steps in detail.

How You Can Create the FrameSource Object

The first step in the construction of a frame is the creation of its FrameSource object, which contains the source definition of a frame. The attributes of the FrameSource object represent the initial settings of the frame when it is first called or opened. The example application creates the FrameSource with the following code:

```c
/* No frame found. Generate a new one. */
fieldtop = 200;        /* Starting field on frame */
test_frame = FrameSource.Create();
```

The Create method creates a new instance of an object of the specified class. In this case, it returns a new instance of a FrameSource object to the test_frame reference variable.
How You Can Build a Frame Dynamically

The fieldtop variable designates the starting position of the top of each field on the form. It is incremented by the size of each field and some predetermined amount (identifying the amount of space between fields) in each iteration of the select loop. The example frame sets this variable at this point so that if the user selects the Go menu item more than once, to create more than one frame, the top field on each new frame starts in the correct position.

How You Can Set the FrameSource Attributes

After the starting frame's Go operation creates the FrameSource object, it sets the following frame attributes:

**Name**
- Specifies the name of the frame

**Datatype**
- Specifies the data type of the frame's return value

**IsNullable**
- Specifies whether the return value is nullable

**WindowTitle**
- Defines the title that appears on the frame's window

**CurMode**
- Defines the frame's mode

**WindowPlacement**
- Defines the location of the frame's window when the frame is called or opened

The sample application assigns the following values to these attributes:

```plaintext
/* Fill in the Frame Information */
test_frame_name = table_choices + '_frame';
test_frame.Name = test_frame_name;
test_frame.DataType = 'integer4';
/* This is default */
test_frame.IsNullable = TRUE;
/* This is default */
test_frame.WindowTitle = 'Edit Data For Table: ' +
        table_choices;
test_frame.CurMode = FM_UPDATE;
test_frame.WindowPlacement = WP_FLOATING;
```

Some frame attributes, such as StartMenu, are set later in the program (after the code creates the objects contained by the attributes). Other frame attributes are left at their default values. For an example of creating the StartMenu object, see How You Can Create the Frame Menu Items (see page 411).
How You Can Create the FrameForm Object

The first step in creating the frame's form is creating a FrameForm object, which is a subform that contains the fields on a form. The only step involved in creating a FrameForm object is invoking the Create method of the FrameForm system class and assigning the result to the generated frame's TopForm attribute. There are no FrameForm attributes to set.

The following statement from the example application creates a FrameForm object and assigns it to the TopForm attribute of the FrameSource object represented by test_frame:

```plaintext```
test_frame.TopForm = FrameForm.Create();```
```

How You Can Construct the Fields

In the DynamicFrame frame, all of the steps involved in constructing the form's fields occur inside a select loop. A select statement retrieves the column names of the chosen table. As each is returned, the statements in the select loop generate its corresponding field, including the field's title and data type, and attach it to the form.

Moreover, because the script of the generated frame in the DynamicFrame application contains SQL statements (the select statement and cursor statements) that include the column names, the select loop also contains code that builds these SQL statements.

The following code includes the select statement that retrieves the column names:

```plaintext```
/* Get the list of columns in the form */
numcolumns = 0;
repeated select column_name as :columnname,
    column_datatype as :columndatatype,
    column_length as :columnlength
from iicolumns
where table_name = :table_choices
```
The DynamicFrame frame uses a select loop to process the values returned by the select statement. The following code adds the column names to the SQL statement strings that are part of the generated frame's script:

begin
/* Use the SQL trim function to remove trailing blanks. */
columnname = trim(columnname);
columndatatype = trim(columndatatype);
/* Add column name to SQL statement strings being generated */
if numcolumns <= 0 then
/* This is the first column found. Insert statement headers */
selectstring = 'open table_cursor for select ' +
HC_NEWLINE + HC_TAB + HC_TAB + HC_TAB +
columnname;
fetchstring = 'fetch table_cursor into ' +
HC_NEWLINE + HC_TAB + HC_TAB + HC_TAB + ':' +
columnname + ' = :' + columnname;
selectupdatestring = 'for update of ' +
HC_NEWLINE + HC_TAB + HC_TAB + HC_TAB +
columnname;
updatestring = 'update ' + table_choices +
HC_NEWLINE + HC_TAB + HC_TAB + 'set ' + HC_TAB +
columnname + '=' + columnname;
else
/* Not the first column. Concatenated to strings */
selectstring = selectstring + ',' + HC_NEWLINE +
HC_TAB + HC_TAB + HC_TAB + columnname;
fetchstring = fetchstring + ',' + HC_NEWLINE +
HC_TAB + HC_TAB + HC_TAB + ':' + columnname + ' = :' + columnname;
updatestring = updatestring + ',' + HC_NEWLINE +
HC_TAB + HC_TAB + HC_TAB + columnname + ' = :' +
columnname;
selectupdatestring = selectupdatestring + ',' +
HC_NEWLINE + HC_TAB + HC_TAB + HC_TAB +
columnname;
endif;
numcolumns = numcolumns + 1;

This code uses the SQL trim string function to remove trailing blanks from the text of the column's name and data type.
The variable numcolumns, initialized to zero before the select, controls the execution of this code. If numcolumns is non-positive, the columnname variable represents the first row returned by the select statement and the Go operation executes the code that builds the initial portions of the SQL statements. The numcolumns variable is incremented with each pass through the loop so that subsequent values of the columnname variable are added to an existing string.

The variables that hold the character strings representing the SQL statements are of the data type varchar(2000). The system constants HC_NEWLINE and HC_TAB represent newline and tab characters, respectively. If your statement strings are longer than 2000 characters, it is preferable to use StringObject variables instead of varchar variables.

After adding the column name to the statement strings, the Go operation constructs the field title and the field itself (all of which are entry fields in the example). The following code creates the field title and attaches it to the form:

```c
trim_ptr = FreeTrim.Create();
trim_ptr.XLeft = trimleft;
trim_ptr.YTop = fieldtop;
trim_ptr.TextValue = trim((
    trim(uppercase(left(columnname,1))) +
    trim(lowercase(shift(columnname,-1))) + ':');
trim_ptr.IsBold = TRUE;
trim_ptr.TypeSize = 10;
trim_ptr.Width = 33 * widthchar;
trim_ptr.Height = heightchar;
trim_ptr.ParentField = test_frame.TopForm;
```

Each field's title is an object of the class FreeTrim, which is used only for textual trim on forms. The previous code defines the trim's position on the form (XLeft and YTop), its actual text value (TextValue), and the appearance of the text (IsBold, TypeSize, Width, and Height).

The code also uses SQL string functions (uppercase and lowercase) to ensure that the title appears with initial capitalization.

Specifying a ParentField attribute for an object identifies which composite field contains that object. Therefore, assigning the form object in test_frame.TopForm to the ParentField attribute of the FreeTrim object attaches the trim to the form by identifying the form as the composite field that contains the trim.
After the title is constructed and attached to the form, the following code constructs its corresponding entry field:

```c
field_ptr = EntryField.Create();
field_ptr.Name = columnname;
field_ptr.IsNullable = TRUE;
field_ptr.XLeft = fieldleft;
field_ptr.YTop = fieldtop;
```

The Create method returns an object of type EntryField to the reference variable, field_ptr. The previous code sample assigns a name to the field_ptr variable and specifies its nullability and its top left-hand starting position.

The Go operation then constructs the data type for the field:

```c
fieldheight = 1;
/* Check first for non-character data types. */
if columndatatype = 'integer' then
    columnlength = 13;
    field_ptr.DataType = 'integer';
    field_ptr.FormatString = 'f10';
    field_ptr.IsMultiLine = FALSE;
elseif columndatatype = 'float' then
    columnlength = 15;
    field_ptr.DataType = 'f8';
    field_ptr.FormatString = 'f12.2';
    field_ptr.IsMultiLine = FALSE;
elseif columndatatype = 'date' then
    columnlength = 25;
    field_ptr.DataType = 'date';
    field_ptr.FormatString = 'd"Feb 3, 1901"';
    field_ptr.IsMultiLine = FALSE;
elseif columndatatype = 'money' then
    columnlength = 15;
    field_ptr.DataType = 'money';
    field_ptr.FormatString = '"$$$.$$$.$$$.nn"';
    field_ptr.IsMultiLine = FALSE;
else /* All character datatypes follow */
    field_ptr.DataType = 'varchar(' + ascii(columnlength) + ')';
    /* See if we need a multiline entry field */
    if columnlength > 50 then
        field_ptr.IsMultiLine = TRUE;
        field_ptr.FocusBehavior = FT_TABTO;  /* Default */
        fieldheight = columnlength / 50 + 1;
        columnlength = 50;
        if fieldheight > 4 then
            field_ptr.HasScrollBar = TRUE;
            fieldheight = 4;
        endif;
    endif;
```
else
    field_ptr.IsMultiLine = FALSE;
    field_ptr.FocusBehavior = FT_TABTO;
    if columnlength < 5 then
        columnlength = 5;
    endif;
endif;
endif;

If the data type is integer, float, date, or money, the preceding code ignores
the column length returned by the select statement and assigns a length
explicitly. Column length is set explicitly for non-character columns because
the column length returned by the select statement identifies the number of
bytes used by Ingres to store the data internally. Because the DynamicFrame
frame requires the column length to define the width of the field on the form,
it explicitly assigns column length values to these data types. (Alternatively,
the CharsPerLine attribute of EntryField objects can be used to define the width
of the field.)

In addition to assigning the column length for each non-character data type,
the Go operation assigns the appropriate values for the DataType,
FormatString, and IsMultiLine attributes of the EntryField objects.

If the value in the columndatatype variable indicates that the column has a
character data type (char, varchar, c, or text), the preceding code sets the
field’s data type to varchar. The SQL ascii function converts the integer value
in the columnlength variable to a character string.

In addition, character columns that are longer than 50 characters are
formatted as multiline fields, and a maximum field height and length is
enforced. Scroll bars are enabled for columns that require more than four 50-
character lines to display.

After the Go operation constructs each field, it sets relevant attributes and
attaches the field to the form. The example code sets each entry field’s width
equal to the value of the columnlength variable multiplied by the width of one
character. The height of each entry field is set to the height of one character
times the fieldheight variable. The outline width of each field is set using the
LW_VERYTHIN constant.
The following code from the example frame sets each entry field’s attributes and attaches it to the form:

```
/* Now place it on the form. */
field_ptr.Width = columnlength * widthchar;
field_ptr.Height = heightchar * fieldheight;
field_ptr.OutlineWidth = LW_VERYTHIN;
/* attach to form */
field_ptr.ParentField = test_frame.TopForm;
fieldtop = fieldtop + field_ptr.Height + vertspace;
end; /* End of select loop */
commit;
```

After attaching the field to the form, the previous code assigns a new value to the fieldtop variable, which determines the position of the next field. The value in the fieldtop variable was assigned to the YTop attribute when the entry field was constructed. For discussions on setting the fieldtop variable, see How You Can Create the FrameSource Object (see page 403).

The statement in the previous code that sets the fieldtop variable takes the current value of fieldtop and adds to it the height of the field just constructed and a predetermined amount of vertical space. The resulting figure is the value of fieldtop for the next iteration of the loop and places the next field on the form below the field just constructed. The amount of space between them is equal to the amount specified by the vertspace variable.

The select loop is repeated once for each column name returned by the select statement. When it is finished, all fields have been constructed and placed on the form. In addition, portions of the SQL statements used in the generated frame’s script are also constructed.

**How You Can Set the Form Height and Width**

After all the fields have been constructed, you can set the dimensions of the window displaying the frame by setting the WindowHeight and WindowWidth attributes of the FrameSource object. The following code from the Go operation of the DynamicFrame frame sets these dimensions:

```
/* Now set the form height and width */
test_frame.WindowHeight = fieldtop + vertspace;
test_frame.WindowWidth = maxformwidth;
if test_frame.WindowHeight > maxformheight then
test_frame.WindowHeight = maxformheight;
test_frame.HasScrollbars = TRUE;
else
test_frame.HasScrollbars = FALSE;
endif;
```
To prevent the window's height from exceeding a predetermined amount, the example code contains an if statement that checks the height. If it exceeds the limit (the value of the maxformheight variable), the WindowHeight attribute is set to the value of the maxformheight variable and scroll bars are added to the window, allowing the user to scroll to hidden fields. If the window's height does not exceed the limit, the scroll bars are not added.

**How You Can Complete the SQL Statements**

The Go operation finishes creating the text of the SQL statements begun in the select loop. Adding formatting characters and the final clause completes the selectstring variable. Adding the terminating semicolon completes the selectupdatestring and fetchstring variables. The following code from the DynamicFrame frame completes the text for the variables used in the SQL statements:

```csharp
/* Now complete the select and fetch strings */
selectstring = selectstring + HC_NEWLINE + HC_TAB +
    HC_TAB + 'from ' + table_choices;
selectupdatestring = selectupdatestring + ';';
fetchstring = fetchstring + ';';
```

The completed strings are used later when the frame script for the generated frame is constructed.

**How You Can Create the Frame Menu Items**

The generated frame created by the DynamicFrame frame contains a menu bar that has two selections, File and Edit. If the user selects File, two new choices appear, Commit and Close, and Rollback and Close. If the user selects Edit, three new choices appear: Delete and Next, Update and Next, and Next.

The bar across the top of the generated frame's window, which displays the initial menu choices, is an instance of the MenuBar class. Each initial menu option and the submenu options accessed through the initial option is an instance of the MenuGroup class. In the generated frame, each of the submenu choices is an instance of the MenuButton class.

In the FrameSource object, which contains the frame's definition, the menu bar is contained in the StartMenu attribute. Therefore, to construct a menu for a frame, first create the MenuBar object and assign it to the StartMenu attribute. The following code creates the menu bar and attaches it to the generated frame:

```csharp
    test_frame.StartMenu = MenuBar.Create();
```
How You Can Construct a File Menu Group

The following discussion steps through the construction of the File menu group.

1. Create the MenuGroup object:
   ```csharp
top_menu = MenuGroup.Create(); /* File Menu */
```

2. Define the TextLabel and internal name for the File menu group:
   ```csharp
top_menu.TextLabel = 'File';
top_menu.Name = 'file_menu';
```
   The text assigned to the TextLabel attribute is displayed by the frame in its Menu bar. The preceding code, therefore, causes the word "File" to appear as one of the selections available in the frame's initial menu. The value assigned to the Name attribute is the name of the reference variable pointing to that MenuGroup object.

3. Define the two submenu options that appear when the user selects the File option from the Menu bar: "Commit and Close" and "Rollback and Close." Defining these operations follows the same procedures as creating the MenuGroup:
   - Create the object for each item (in this case a MenuButton object).
   - Define the TextLabel and Name for each menu item.

   The following code creates the first submenu item:
   ```csharp
test_menu = MenuButton.Create();
test_menu.Name = 'commit_menu';
test_menu.TextLabel = 'Commit and Close';
```

4. In addition to letting the user to commit the transaction and close the frame by selecting the Commit menu button, the dynamic frame provides a second way to commit and close. The example frame provides alternative access to the Commit operation by defining a speed key, which lets the user select the Close operation from the keyboard rather than using the mouse.

   The following code defines the speed key:
   ```csharp
test_menu.SpeedKey = SK_CLOSE;
```

5. After each submenu option is defined, the following code attaches it to the Field MenuGroup object:
   ```csharp
test_menu.ParentMenuGroup = top_menu;
```
   The preceding statement makes the MenuGroup object represented by top_menu to be the parent of the MenuButton object represented by test_menu.
6. After defining all of the individual items for the MenuGroup, the following code attaches the MenuGroup itself to the MenuBar object:

```csharp
top_menu.ParenMenuGroup = test_frame.StartMenu;
```

The following code is the complete code that constructs both menu groups that belong to the menu bar:

```csharp
top_menu = MenuGroup.Create();   /* File Menu */
top_menu.TextLabel = 'File';
top_menu.Name = 'file_menu';

test_menu = MenuButton.Create();
    /* Commit and close menu item */
test_menu.Name = 'commit_menu';
test_menu.TextLabel = 'Commit and Close';
test_menu.SpeedKey = SK_CLOSE;
test_menu.ParentMenuGroup = top_menu;
    /* Attach to File menu */

test_menu = MenuButton.Create();
    /* Rollback and close menu item */
test_menu.Name = 'rollback_menu';
test_menu.TextLabel = 'Rollback and Close';
test_menu.ParentMenuGroup = top_menu;
    /* Attach to File menu */

top_menu.ParentMenuGroup = test_frame.StartMenu;
    /* Now, create the Edit menu */
top_menu = MenuGroup.Create();   /* Edit Menu */
top_menu.TextLabel = 'Edit';
top_menu.Name = 'edit_menu';

test_menu = MenuButton.Create();
    /* Delete menu item */
test_menu.Name = 'delete_menu';
test_menu.TextLabel = 'Delete and Next';
test_menu.SpeedKey = SK_DELETE;
test_menu.ParentMenuGroup = top_menu;
    /* Attach to Edit menu */

test_menu = MenuButton.Create();
    /* Update menu item */
test_menu.Name = 'update_menu';
test_menu.TextLabel = 'Update and Next';
test_menu.ParentMenuGroup = top_menu;
    /* Attach to Edit menu */

test_menu = MenuButton.Create();
    /* Next menu item */
test_menu.Name = 'next_menu';
test_menu.SpeedKey = SK_NEXT;
test_menu.TextLabel = 'Next';
test_menu.ParentMenuGroup = top_menu;
    /* Attach to Edit menu */
/* Attach to Frame */
top_menu.ParentMenuGroup = test_frame.StartMenu;
```

For more information about the various types of menu objects, see the Language Reference Guide online help.
How Generating the Frame Source Code Works

Part of the source code for the generated frame was written in the select loop that constructed the fields. The next section of the DynamicFrame frame script creates the generated frame's source code from the strings constructed in the select loop.

The starting frame uses the following steps to create the generated frame’s source code:

1. Creates the string object to hold the completed script and assigns it to the Script attribute of the FrameSource:
   ```java
   test_frame.Script = StringObject.Create();
   ```

2. Constructs the script, creating an initialize block and event code for five Click events that correspond to the five MenuButton objects constructed earlier in the program (Commit, Rollback, Delete, Update, and Next).

The example frame uses the ConcatVarchar method (defined for the StringObject class) to concatenate the various pieces of the initialize statement and each event block into one large string. The ConcatVarchar method appends the string specified by the text parameter to the StringObject and returns a reference to the original StringObject.

The following code constructs the script for the generated frame:

```java
/* Set up the initialize block */
test_frame.Script.ConcatVarchar(text =
    'initialize (table_cursor = CursorObject) =
    ' + HC_NEWLINE + HC_TAB + 'begin' +
    HC_NEWLINE + HC_TAB + HC_TAB + selectstring +
    HC_NEWLINE);

test_frame.Script.ConcatVarchar(text =
    HC_TAB + HC_TAB +
    selectupdatestring + HC_NEWLINE + HC_TAB +
    HC_TAB + 'CurFrame.SendUserEvent(eventname = ''Next'');' +
    HC_NEWLINE + HC_TAB + 'end;' +
    HC_NEWLINE + HC_NEWLINE);

/* Set up the file.commit script */
test_frame.Script.ConcatVarchar(text =
    'on click file_menu.commit_menu = ' +
    HC_NEWLINE + HC_TAB + 'begin' + HC_NEWLINE +
    HC_TAB + HC_TAB + 'close table_cursor;' +
    HC_NEWLINE + HC_TAB + HC_TAB + 'commit work;' +
    HC_NEWLINE + HC_TAB + HC_TAB + 'return;' +
    HC_NEWLINE + HC_TAB + 'end;' +
    HC_NEWLINE + HC_NEWLINE);
```
/* Set up the file.rollback script */
test_frame.Script.ConcatVarchar(text =
  'on click file_menu.rollback_menu =
    ' + HC_NEWLINE + HC_TAB + 'begin' +
    HC_NEWLINE + HC_TAB + 'close
    table_cursor;' + HC_NEWLINE + HC_TAB + HC_TAB
    + 'rollback work;' + HC_NEWLINE + HC_TAB
    + HC_TAB + 'return;' + HC_NEWLINE + HC_TAB +
    'end;' + HC_NEWLINE + HC_NEWLINE);

/* Set up the edit.delete script */
test_frame.Script.ConcatVarchar(text =
  'on click edit_menu.delete_menu =
    ' + HC_NEWLINE + HC_TAB + 'begin' + HC_NEWLINE
    + HC_TAB + HC_TAB + 'delete from ' +
    table_choices + ' where current of
    table_cursor;' + HC_NEWLINE + HC_TAB +
    + 'CurFrame.SendUserEvent(eventname =
      ''Next'' );' + HC_NEWLINE + HC_TAB + 'end;' +
    HC_NEWLINE + HC_NEWLINE);

/* Set up the edit.update script */
test_frame.Script.ConcatVarchar(text =
  'on click edit_menu.update_menu =
    ' + HC_NEWLINE + HC_TAB + 'begin' + HC_NEWLINE
    + HC_TAB + HC_TAB + updatestring +
    HC_NEWLINE + HC_TAB + HC_TAB + ' where
    current of table_cursor:' + HC_NEWLINE
    + HC_TAB + HC_TAB + 'CurFrame.SendUserEvent
    (eventname = ''Next'' );' + HC_NEWLINE +
    + HC_TAB + 'end;' + HC_NEWLINE + HC_NEWLINE);

/* Set up the edit.next script */
test_frame.Script.ConcatVarchar(text =
  'on click edit_menu.next_menu,' +
  'on userevent ''Next'' ='' +
  HC_NEWLINE + HC_TAB + 'begin' +
  HC_NEWLINE + HC_TAB + HC_TAB + fetchstring +
  HC_NEWLINE + HC_TAB + HC_TAB + 'if table_cursor.State = CS_NO_MORE_ROWS
  then' + HC_NEWLINE + HC_TAB + HC_TAB +
  + 'message ''No more rows.'''; ' + HC_NEWLINE
  + HC_TAB + HC_TAB + 'endif;' + HC_NEWLINE +
  + HC_TAB + 'end;' + HC_NEWLINE + HC_NEWLINE);

3. To facilitate debugging, writes the completed script to a file using the
   WriteToFile method (defined for the StringObject class):
   test_frame.Script.WriteToFile("test.script");

   The WriteToFile method invoked in the preceding example creates a file
   named test.script.
How You Can Build a Frame Dynamically

How You Can Add the Frame to the Application

After the frame has been fully defined, it must be attached to the application by setting the ParentApplication attribute of the new frame’s FrameSource object to the same value that is in the ParentApplication attribute of the currently executing frame. (The currently executing frame is the starting frame of the dynamic application.)

The following code from the example application sets the ParentApplication attribute of the generated frame:

```csharp
```

This statement assigns the contents of the ParentApplication attribute of the current frame’s FrameSource object to the corresponding attribute of the generated frame’s FrameSource object. For a more detailed explanation of the preceding syntax, see How You Can Attach the Frame to the Application (see page 396).

The DynamicFrame frame performs the additional step of adding the frame to a list of already generated frames. Because the sample application lets a user return to the initial frame, select another table, and select the Go menu item again, a global array variable is used to keep track of which frames have already been generated.

Because the Generated_Frame_List global variable stores the names of generated frames, an existing frame can be executed without requiring regeneration if the user selects a table for which a frame already exists.

The following code from the example application adds the newly generated frame to the list in the global array variable:

```csharp
i = Generated_Frame_List.LastRow + 1;
Generated_Frame_List[i].table_name = table_choices;
Generated_Frame_List[i].frame_name = table_choices + '_frame';
Generated_Frame_List[i].frame_source = test_frame;
```

The global array variable, Generated_Frame_List, has three attributes: table_name, frame_name, and frame_source. The previous code uses the LastRow attribute (defined for the ArrayObject class) to determine the row sequence number of the last row in the array. The current value of the LastRow attribute is incremented by one to create a new index into the array.
The reference to a previously non-existent row automatically appends a new, empty row to the end of the array. After the row has been added, each of its attributes is assigned the appropriate value for the newly generated frame. The process used to add a new row in the example code is called adding a row by first reference. For more information about adding rows to an array, see Working with Arrays, Table Fields, and Collections (see page 205).

**How You Can Execute the Frame**

The following code from the DynamicFrame frame assigns a name to the newly generated frame and executes it:

```plaintext
test_frame_name = table_choices + '_frame';
callframe :test_frame_name;
```

The callframe statement compiles and runs the new frame. When OpenROAD automatically compiles a dynamically created frame or procedure on first call, it places the text of any compilation errors into Proc4GLSource's Compile_Errors attribute. Check whether the Compile_Errors attribute is null after the first call to detect whether compilation errors occurred.

The following code from the example frame checks the Compile_Errors attribute and writes any errors to a file to facilitate debugging:

```plaintext
if test_frame.Compile_Errors is not null then
    test_frame.Compile_Errors.WriteToFile ('test.errors');
    Generated_Frame_List.RemoveRow(rownumber = i);
endif;
```

If the frame does not compile correctly, its name is removed from the list in the global array variable, Generated_Frame_List.

**Close Menu Item**

In addition to the File menu option, the DynamicFrame application's starting frame has a Close menu option. The event code for the Close menu item contains only the return statement to terminate the application. The following code is the complete Close event block:

```plaintext
on click menu.file_menu.close_menu =
begin
    return;
end;
```
Dynamic Statements Considerations

The following section contains information about what you should be aware of when using dynamic statements.

How You Can Populate a FlexibleForm, StackField, or MatrixField Dynamically

Duplicating or creating several fields and parenting them to a FlexibleForm, StackField, or MatrixField that is currently displayed in a frame can be extremely slow because of the amount of configuration and redrawing that must take place as each new field is created and added to its parent field. To increase the performance of creating several fields in this situation, you should unparent the FlexibleForm, StackField, or MatrixField before creating the additional children and reparent it immediately afterward.

For example, if you have a FlexibleForm (flexform) that you are adding children to in a loop, you can use the following code to dramatically increase the response time of this task:

```vba
/* Unparenting the FlexibleForm first will speed up the creation of all the children we are adding. */
field(flexform).ParentField = null;
for x = 1 to 100 do
  fld[x] = EntryField.Create();
  /* ** Set additional attributes for ** each EntryField here. */
  fld[x].ParentField = field(flexform);
endfor;
/* Now we reparent the FlexibleForm to cause it ** to reappear on the form (now it is only ** reconfigured and drawn once, instead of ** every time a child is added). */
field(flexform).ParentField = CurFrame.TopForm;
```
Chapter 16: Writing a Template Assistant

This section contains the following topics:

- Conceptual Overview (see page 419)
- How You Can Write an Assistant (see page 420)
- How You Can Ensure Compatibility Between the Assistant and ApplyTemplate (see page 425)

You can create and invoke sophisticated templates by writing a frame or field assistant and associating it with your template.

Conceptual Overview

Frame and field templates can be thought of as “boilerplates” that are used to create standard frames and fields throughout your application. They provide consistency in appearance and behavior among the frames and fields you create.

Assistants

Assistants are procedures that can be attached to a frame or a field template. They can be designed to prompt the user for varying amounts of information used to customize the frame or field according to the user's needs or to perform a certain function each time the field or frame is invoked.

How You Can Attach an Assistant to a Template

You can attach an assistant to an existing template through the frame or field template's Property Inspector. (For more information, see the User Guide.) An assistant can also be attached to a field template from OpenROAD Workbench's Field Template Editor.

How You Can Ensure Compatibility Between a Frame Assistant and ApplyTemplate

When you want to upgrade a frame that has been previously generated from a template, you can do so by using the ApplyTemplate utility. Doing so incorporates any enhancements or new features that you may have made to the template into your existing frames and fields.
**Note:** Fields generated from a field template are completely overwritten when the ApplyTemplate utility is run; therefore, any changes made to the field are lost.

Be aware that any changes you make to your frame or field must be made in such a way that they are not overwritten by the ApplyTemplate utility. OpenROAD provides a mechanism whereby you can identify the areas of the frame that are designated as “user” areas—where user modifications are preserved. The other areas belong to and are completely overwritten by the ApplyTemplate utility. This involves several considerations when writing and modifying your assistant procedure code. These concepts are discussed in the following sections of this chapter.

## How You Can Write an Assistant

In OpenROAD, the developer can associate a 4GL procedure with a frame or field template, which is then referred to as the template’s “assistant.” This template, when combined with an assistant procedure, enables the user to create frames and fields that are more powerful and immediately useful than those created from templates alone.

The following specifications for developing an assistant procedure include:

- The assistant must be a 4GL procedure. The 4GL procedure itself can, however, subsequently call frames to interact with users.
- The 4GL procedure has three parameters, all of which are mandatory.

For `frame` template assistant procedures, the parameters are:

### `fs`

**Parameter Type:** FrameSource

Specifies the instance of the frame that is currently being created or modified by the ApplyTemplate utility

### `return_code`

**Parameter Type:** Integer (ER_OK, ER_FAIL)

Specifies the return code. It is assigned the value ER_OK if the procedure completes successfully; otherwise, it is assigned ER_FAIL.
batch_mode

**Parameter Type:** Integer (TRUE, FALSE)

Specifies a Boolean parameter telling the frame assistant procedure whether it is being run in batch mode. If TRUE, the procedure cannot depend on any form of user interaction, including calling user frames for user input. If FALSE, the frame assistant can make use of frames for user input.

In the ApplyTemplate utility dialog, selecting the Batch option passes a value of TRUE; and for the Interactive option, FALSE.

For **field** template assistant procedures, the parameters are:

**ff**

**Parameter Type:** FormField

Specifies the instance of the field that is currently being created

**return_code**

**Parameter Type:** Integer (ER_OK, ER_FAIL)

Specifies a return code. It is assigned the value ER_OK if the procedure has completed successfully; otherwise, it is assigned ER_FAIL.

**batch_mode**

**Parameter Type:** Integer (TRUE, FALSE)

Specifies a Boolean parameter telling the field assistant procedure whether it is being run in batch mode. If TRUE, the procedure cannot depend on any form of user interaction, including calling user frames for user input. If FALSE, the field assistant can make use of frames for user input.

In the ApplyTemplate utility, selecting the Batch option passes a value of TRUE; and for the Interactive option, FALSE.

The following sections present examples of frame and field template assistants.

**Example—Frame Template Assistant**

In this example, a frame template assistant procedure has been designed to customize a frame template. The frame template assistant procedure prompts the user for a text string, assigns that string to a dynamically created FreeTrim field, and then places the field on the form.
First, the parameters are defined:

Procedure Create_Dyn_Fld
(
    fs = FrameSource not null,
    return_code = integer not null,
    batch_mode = integer not null
) =

Because the procedure must support batch mode, a default string is specified. Otherwise, the user is prompted for the string:

if batch_mode = TRUE then
text_string.value = 'Hello, world!';
else
    status = curprocedure.replypopup(messagetext = 'Enter text string.',
                                      reply = text_string);
    if status != PU_OK then
        return_code = ER_FAIL;
        return:
    endif;
endif;

Next, a text trim field is loaded with the specified text string:

trim_field = freetrim.create();
trim_field.textvalue = text_string.value;

Finally, the procedure must assign a value to the return code parameter and attach the composite field to the form:

trim_field.ParentField = fs.topform;
return_code = ER_OK;

How You Can Attach the Assistant to a Frame Template

In the same application that contains the Create_Dyn_Fld frame assistant procedure, a frame template can be created (for example, My_Frame_Template). To associate the frame template My_Frame_Template with its assistant, open the Property Inspector for the frame template in the Frame Editor, and specify Create_Dyn_Fld as the assistant procedure.

When a user frame is subsequently created from the template My_Frame_Template, the procedure Create_Dyn_Fld is called, which populates the new frame's form with a trim field entered by the user.
The full text of the frame assistant procedure is as follows:

Procedure Create_Dyn_Fld

(  
    fs = FrameSource not null,  
    return_code = integer not null,  
    batch_mode = integer not null
) =

declare  
    trim_field = FreeTrim default null;  
    text_string = StringObject;  
    status = integer not null;  
enddeclare  
begin  
    if batch_mode = TRUE then  
        text_string.value = 'Hello, world!';  
    else  
        status = curprocedure.replypopup(messagetext = 'Enter text string.',  
                                        reply = text_string);  
        if status != PU_OK then  
            return_code = ER_FAIL;  
            return;  
        endif;  
    endif;  
    trim_field = freetrim.create();  
    trim_field.textvalue = text_string.value;  
    trim_field.ParentField = fs.TopForm;  
    return_code = ER_OK;  
    return;  
end;

Example—Field Template Assistant

In this example, a field template assistant procedure has been designed to customize a field template.

The field template consists of a composite field that contains several entry fields and one text field for the title. The field template assistant procedure prompts the user for the title and assigns that text string to the text field of the composite field.

First, the parameters are defined:

Procedure Create_Label

(  
    ff = FormField not null,  
    return_code = integer not null,  
    batch_mode = integer not null
) =
Because the procedure must support batch mode, a default string is specified. Otherwise, the user is prompted for the string:

```sql
if batch_mode = TRUE then
    text_string.value = 'Hello, world!';
else
    status = curprocedure.replypopup(message_text = 'Enter text string.',
                                      reply = text_string);
    if status != PU_OK then
        return_code = ER_FAIL;
        return;
    endif;
endif;
```

Next, the text field with the variable name "label" is loaded with the specified text string:

```sql
field_obj = CompositeField(ff).FieldByName(name = 'label');
EntryField(field_obj).textvalue = text_string.value;
```

Finally, the procedure needs to assign a value to the return code parameter:

```sql
return_code = ER_OK;
```

### How You Can Attach the Assistant to a Field Template

In the same application that contains the frame assistant procedure Create_Label, a field template can be created (called, for example, My_Field_Template). This template should contain one composite field that consists of several entry fields, and one free trim field for the label with the variable name of "label." To associate the field template My_Field_Template with its assistant, open the Property Inspector for the field template in its editor, and specify Create_Label as the assistant procedure.

When a form field is subsequently created from the template My_Field_Template, the procedure Create_Label is called, which populates the new label with the text string entered by the user.

The full text of the field assistant procedure is as follows:

```sql
Procedure Create_Label
(
    ff = FormField not null,
    return_code = integer not null,
    batch_mode = integer not null
) =

declare
    text_string = StringObject;
    status = integer not NULL;
enddeclare
```
begin
{
if batch_mode = TRUE then
  text_string.value = 'Hello, world!';
else
  status = curprocedure.replypopup(message_text = 'Enter text string.',
                                  reply = text_string);
  if status != PU_OK then
    return_code = ER_FAIL;
    return;
  endif;
endif;

field_obj = CompositeField(ff).FieldByName(name = 'label');
EnterField(field_obj).textvalue =
  text_string.value;
return_code = ER_OK;
return;
}
end;

How You Can Ensure Compatibility Between the Assistant and ApplyTemplate

Before you write an assistant procedure, there are certain design guidelines that you should consider. This section discusses these considerations with regard to the ApplyTemplate utility.

How the Frame Creation Process Works

When a frame is created from a frame template with OpenROAD, the following steps are performed:

- The frame template specified by the user is fetched from its library image file.
- All of the static functions defined for the frame template—its form, menu, and 4GL scripts—are copied to the newly created frame.
- If an assistant has been defined for the frame template, the assistant is then called and is passed the newly created frame for dynamic customization.

The static functionality provided by the frame template and the dynamic functionality provided by the assistant combine to create a frame that is immediately unique and usable. The ApplyTemplate utility provides a mechanism to perform these three steps on a frame that has previously been created.
How You Can Use ApplyTemplate on a Frame

The need to use the ApplyTemplate utility on a generated frame arises in the following situations:

- When the static functionality provided by a frame template has changed, and the user wants to fold these changes into a frame that has already been generated
- When the user wants to change the portion of a generated frame created by a specific frame template assistant by running the assistant again

In both these cases, however, it is desirable for the ApplyTemplate utility to make its changes without affecting any changes made to the frame by the user subsequent to its creation.

How You Can Partition the Frame

To solve the problem of applying template changes without affecting user changes, the ApplyTemplate utility assumes that the frame template assistant has been designed so that the generated frame is effectively partitioned into two areas:

- That which belongs to and has been modified by the user
- That which belongs to and has been modified by the frame template assistant that generated the frame

After the ApplyTemplate utility is used on a frame generated from a frame template assistant, the area belonging to the user is not touched, but the area belonging to the frame template assistant is completely overwritten, and any changes made by a user to that area is lost.

A frame consists of three major parts: a form, a 4GL script, and a menu. For the purposes of the ApplyTemplate utility, each has a clearly defined boundary separating the user area from the frame template assistant area.

The boundary separating the two areas on the frame’s form is defined by a high-level composite field that resides on the form of both the frame template and the frames that are created from it. Thus, on the form of frames created from the frame template, anything inside the high-level composite field belongs to the frame template assistant, and anything outside of it belongs to the user.

High-level Composite Field

When designing a frame template for use with a frame template assistant and the ApplyTemplate utility, be sure that all form fields reside within the bounds of a high-level composite field.
A tag associated with this field is assigned to a specific macro variable defined for the frame template. The tag exists as a string value assigned to the ClientText attribute of the high-level composite field. The value of the string is equal to the value of the $TopComposite macro variable defined for the frame template.

By evaluating the $TopComposite macro variable for a frame template and the frame generated from it, the ApplyTemplate utility is able to identify which specific high-level composite field on the frame's form comprises the template-owned portion. By overwriting only this high-level composite, and no other, the ApplyTemplate utility can be guaranteed to not affect any user changes to the frame.

Because the high-level composite of the frame is overwritten by the corresponding high-level composite from the frame template, all of the fields generated by the frame template assistant when the frame was first created (or since the last time the ApplyTemplate utility was run) are overwritten or lost. The second function of the ApplyTemplate utility, therefore, is to regenerate the dynamic fields and queries created by the frame template assistant. This function, however, is not performed by the ApplyTemplate utility itself, but is instead performed entirely by the frame assistant, which is called from the ApplyTemplate utility.

**Batch Mode Compared to Interactive Mode**

The ApplyTemplate utility has a mode value, with the values of interactive or batch, which it passes to the frame template assistant when it is called. The ApplyTemplate utility itself does not behave differently when in these two modes.

**Note:** A frame template does not need to have changed for the ApplyTemplate utility to be run. The ability to regenerate fields and queries can also be used to reconfigure a frame to accommodate changes since the frame was first generated—for example, either in the database tables or in the application requirements.

The ApplyTemplate utility cannot itself guarantee that user customizations to a frame are not affected by sending it through its frame template assistant procedure again. The frame template assistant procedure must be designed in such a way so that it honors the $TopComposite boundary.
How the Field Creation Process Works

From OpenROAD Workbench’s Frame Editor, you can specify that a field be created from a field template. When a field is created from a field template, the following steps are performed:

- The field template specified by the user is fetched from its library image file.
- All of the static functionality defined for the field template—in the form of a 4GL field script and any visual characteristics—is copied to the newly created field.
- If an assistant has been defined for the field template, the assistant is then called and is passed the newly created field for dynamic customization.

The static functionality provided by the field template and the dynamic functionality provided by the assistant combine to create a field that is immediately unique and usable. The ApplyTemplate is a mechanism to perform these three steps on fields that have already been created.

How You Can Use ApplyTemplate on a Field

The need to use the ApplyTemplate utility on a generated field arises in the following situations:

- When the static functionality provided by a field template has changed, and the user wants to introduce these changes that have already been generated
- When the user wants to regenerate a field with new dynamic characteristics

Unlike frames generated from frame templates, fields generated from field templates are not partitioned into two areas, one belonging to the utility, and one belonging to the user. A field regenerated with the ApplyTemplate utility completely replaces the previous field, and any user customizations are lost.

Example—ApplyTemplate Compatibility

In this example, the frame template assistant procedure from the previous example has been modified for use with the ApplyTemplate utility. The difference between this example and the previous one is that the procedure must place the dynamically generated trim field inside of the top-level composite field of the frame template.

For this example, assume that the $_TopComposite macro variable of the frame template has been assigned a value of FreetrimComposite.
First, the parameters are defined:

Procedure Create_Dyn_Fld
(  
   fs = FrameSource not null,
   return_code = integer not null,
   batch_mode = integer not null
)=

Because the procedure must support batch mode, a default string is specified. Otherwise, the user is prompted for the string:

if batch_mode = TRUE then
   text_string.value = 'Hello, world!';
else
   status = curprocedure.replypopup(message_text = 'Enter text string.',
                                      reply = text_string);
   if status != PU_OK then
      return_code = ER_FAIL;
      return;
   endif;
endif;

Next, a text trim field is created and loaded with the specified text string:

trim_field = freetrim.create();
trim_field.textvalue = text_string.value;

Finally, the procedure must assign a value to the return code parameter and attach the composite field to the form. Because the trim field must be assigned to the frame template's top composite field, a recursive search procedure is called to search for a composite field with the name FreetrimComposite. When it is found, the trim field is added to that composite:

topcomposite = callproc
               find_top_composite(fs.topform);
trim_field.ParentField = topcomposite;
return_code = ER_OK;

How You Can Attach the Assistant to a Frame Template

In the same application that contains the Create_Dyn_Fld frame assistant procedure, a frame template can be created (for example, My_Frame_Template). To associate the frame template My_Frame_Template with its assistant, open the Property Inspector for the frame template in its editor, and specify Create_Dyn_Fld as the assistant procedure.

When a user frame created from the My_Frame_Template frame template is run through the ApplyTemplate utility, any modifications made to the frame outside of the composite field defined by the $_TopComposite macro variable is unaffected.
The full text of the frame assistant procedure is as follows:

Procedure Create_Dyn_Fld
(
    fs = FrameSource not null,
    return_code = integer not null,
    batch_mode = integer not null
)=

declare
    trim_field = FreeTrim default null;
    text_string = StringObject;
    topcomposite = CompositeField default null;
    status = integer not null;
    find_top_composite = procedure returning CompositeField;
enddeclare
begin
    if batch_mode = TRUE then
        text_string.value = 'Hello, world!';
    else
        status  = curprocedure.replypopup(messagetext
            = 'Enter text string.',
            reply = text_string);
        if status != PU_OK then
            return_code = ER_FAIL;
            return;
        endif;
    endif;
    topcomposite = callproc
        find_top_composite(composite = fs.topform);
    trim_field.ParentField = topcomposite;
    return_code = ER_OK;
    return;
end;

procedure find_top_composite
(
    composite = compositefield;
)=

declare
    child = compositefield default null;
    i = integer not null;
enddeclare
begin
    if composite.name = 'FreetrimComposite' then
        return composite;
    endif;
    for i = 1 to composite.childfields.lastrow() do
        if composite.childfields[i].isa(class = CompositeField) = TRUE then
            child = find_top_composite(composite = composite.childfields[i]);
            if child IS NOT NULL then
                return child;
            endif;
        endif;
    endfor;
end;
endfor;
    return NULL;
end:
Chapter 17: Preprocessing Your 4GL Code

This section contains the following topics:

Preprocessor (see page 433)
Error Processing (see page 439)

This chapter describes the OpenROAD preprocessor and contains information about using include scripts and macro variables with the preprocessor.

Preprocessor

The preprocessor provides the following extensions to 4GL that let you do the following:

- Include scripts of 4GL code anywhere in your frame script by using an include script that references the 4GL code
- Use macro variables that you define or ones that are defined by the system
- Specify conditional behavior for macro variables
- Specify whether you want code to be included or excluded at compile time

The preprocessor is invoked every time you compile a frame, procedure, or user class. It processes OpenROAD preprocessor statements that you specify in your scripts. These statements are:

- `#include`
- `#define`
- `#undef`
- `#if` – `#endif`
- `#ifdef` – `#endif`
- `#ifndef` – `#endif`

These statements and how to use them with include scripts and macro variables are described in the following sections.
Include Scripts

Include scripts are scripts that contain 4GL code that you can include anywhere in your frame, procedure, or user class script by using the #include statement. This statement references an include script component in the current or an included application.

Your 4GL code block does not appear in the frame script until after the script has been processed. In the preprocessed version, you see only the #include statement and the name of your script as described in The #include Statement (see page 434).

You can use include scripts for:

- Code sharing that cannot be accomplished by including the code in a called procedure or method—for example, blocks of code that include on event statements or conditions
- Variable definition

Include scripts can exist in your current application or in any included application. You create and edit include scripts in OpenROAD Workbench by using the Include Script Editor. You can also display a list of include scripts for an application in the Components portlet of the Develop tab. For more information about how to use this option, see the User Guide.

The #include Statement

The syntax for an include statement is:

```
#include scriptname
```

`scriptname`

Specifies the name of the include script you want to reference

Use an #include statement for each script of 4GL code you want to reference in a frame script. The 4GL code must not be in the file system, it must be in the database.

The # sign must begin in the first column. Spaces are not required between the # sign and the rest of the statement. However, the number of white spaces between "#" and "include" are prefixed to every line in the processed include script. For example, if you want to indent the included code by four spaces, then enter the #include as "#   include" (with three spaces following the #).

The #include statement can be nested to a depth of 20.
General Preprocessor Statement Rules

Macro variables are substitution parameters that you can define and give a value. The preprocessor, when it encounters a macro variable in the source file, substitutes the name with its defined value. In the case where the defined value is a character string, the value substituted does not contain the delimiting quotation marks. A macro variable's scope is global to the frame being generated from the point in the file where it is defined.

There are two types of macro variables that you can use in your 4GL code for preprocessing purposes:

- **User-defined macro variables**
  You define these in the Frame Editor or Procedure Editor, or in your source script.

- **Built-in macro variables**
  The OpenROAD preprocessor determines the value of these macro variables each time you compile. You can use these, but you should not change their values.

The following sections describe how to use preprocessor statements with these macro variable types.

User-defined Macro Variables

You can define and edit your own macro variables in the Macro Variable Editor. The macro variables you define become part of the frame definition or procedure.

User-defined macro variables can have a length of 32 alphanumeric characters. The "$" is considered part of the length. These names are not case-sensitive. Their value cannot evaluate to a preprocessor statement or the name of another macro variable. Macro variables' duration begins when they are defined, and they exist throughout the remainder of a source file.

If you define a macro variable in one field script and then use the macro variable in another field script, processing errors may occur because of the order in which the field scripts are processed. To avoid this error when using a macro variable throughout a frame, declare the macro variable in the frame script or create it interactively.

You can also define macro variables in a script for frames, procedures, or user classes with the #define statement. After a macro variable is defined, it is defined for the rest of the script until it is undefined (with the #undef statement).
For frames that may have separate scripts in many components (fields, menu items), be careful about defining a macro variable in one script and using it in another. The only thing you assume about script ordering is that composite scripts appear before the scripts of the composite’s child fields.

You use the `#define` statement to create your macro variables as detailed in The `#define` Statement (see page 437).

For more information about using the Macro Variable Editor, see the User Guide.

**Built-in Macro Variables**

OpenROAD defines built-in macro variables during preprocessing. You can use them in your 4GL scripts, but you must not change their values.

The built-in macro variables are:

- **$_ApplicationName**
  Specifies the name of the current application

- **$_CurFld**
  Specifies the full name of the current field, for example: fldname, comp.fldname, or tblfd[*].fldname (defined in field scripts only)

- **$_CurFldRow**
  Specifies the current row of a table field array, for example: tblfd[ ].fldname (defined in field scripts only)

- **$_ComponentName**
  Specifies the name of the current frame, procedure, or user class

- **$_ShortRemark**
  Specifies the short remark specified when the current frame was created

- **$_DefaultReturnValue**
  Specifies the default value appropriate for the return type of the current frame or procedure. This cannot be used with methods for a user class script.

For example, a field script can refer to `$_CurFld`, but the compiler only sees the name that is substituted, such as "composite.fldname." This is a generic way to refer to the current field in field scripts.

The following subsections describe the preprocessor statements that you can use in 4GL scripts.
The \#define Statement

The \#define statement is used to create new macro variables and change the value of existing ones. It has the following syntax:

```c
\# define $variable [expr | bool_expr]
```

- **$variable**
  - Specifies the name of the macro variable. The leading $ is required.

- **expr**
  - Specifies the character or integer expression assigned to the macro variable. Character strings must be delimited with single quotation marks.

- **bool_expr**
  - Specifies an expression that evaluates to TRUE or FALSE

For example, to define a macro variable, for a default color, you would use the following statement:

```c
\# define $defaultcolor 'CC_LIGHT_GRAY'
```

Guidelines for Using the \#define Statement

The following rules apply to the \#define statement:

- The \#define statement must be used by itself on one line.
- If no expression is given for a macro variable in a \#define statement, then the preprocessor creates a macro variable with no value.
- Only one macro variable can be created in each \#define statement.
- If you \#define a macro variable that already exists, then the existing macro variable is redefined with a new value. Therefore, the following statement is valid and increments $a:
  ```c
  \#define $a $a+1
  ```
- Multiple \#defines of the same variable are permitted.
- Character macro variables cannot contain embedded single quotation marks.
The #undef Statement

The #undef statement is used to undefine macro variables. It has the following syntax:

```
# undef $variable
```

$variable

Specifies the name of the macro variable you want to undefine. You can specify only one variable in each #undef statement.

You cannot undefine a built-in macro variable. If you use #undef for a macro variable that does not currently exist, the statement is ignored and processing continues.

The #if - #endif Statement

The #if statement can be used to conditionally include or exclude lines of source code at compile time and to test if a condition is TRUE. It has the following syntax:

```
# if (bool_expr) [then]
  ...
# else
  ...
# endif
```

bool_expr

Specifies an integer or character expression that evaluates to TRUE (1) or FALSE(0).

If the expression is TRUE, then any code between the #if and #else is processed and any code between the #else and #endif is ignored. If the tested condition is FALSE, then any lines between the test and an #else, or the test and the #endif (if the #else is omitted) are ignored. The preprocessor removes all of the following statements from the file that are passed to the compiler:

```
#if 0
/* shared procedures */
#include nameproc
#endif
```
Guidelines for Using the #if - #endif Statement

The following syntax rules apply to the #if statement:

- Arguments are not permitted in #if statements.
- The statement:
  
  \[
  \text{# if } \text{variable then} \ldots
  \]
  
  is equivalent to:
  
  \[
  \text{# if}(\text{variable}<>0)\text{then} \ldots
  \]
- The #if statement can be nested within other #if, #ifdef, and #ifndef statements up to a depth of 20.

The #ifdef - #endif Statement

The #ifdef statement can be used to test whether a name has been defined as a macro variable by #define. The syntax for this statement is:

\[
\text{#ifdef } \text{variable} \\
\ldots \\
\text{#else} \\
\ldots \\
\text{#endif}
\]

The #ifdef statement is TRUE when the macro variable has been defined.

The #ifndef - #endif Statement

The #ifndef statement can be used to test whether a name has not been defined as a macro variable by #define. The syntax for this statement is:

\[
\text{#ifndef } \text{variable} \\
\ldots \\
\text{#else} \\
\ldots \\
\text{#endif}
\]

The #ifndef statement is TRUE if the macro variable has not been defined.

Error Processing

When the preprocessor encounters an error, it stops and displays the errors with your source code. It displays your source code with all preprocessor statements up to the point where the preprocessor statement that caused the error appears.
If the compiler discovers an error in the preprocessed code, it displays the code without the preprocessor statements or macro variables. Your line numbers probably differ between your source code and the preprocessed code because the compiler dropped out the preprocessing statements.

**Example—Error Processing**

The following code fragment is an example of using macro variables in code:

```c
#define $_curfld 'enterfield'

#if $_curfld = 'enterfield' then
#else
#endif

#ifndef $_curfld
#else
#endif

#include 'entryapplication'
```
Chapter 18: Debugging Your Application

This section contains the following topics:

- How You Can Use the Debugger (see page 441)
- Start and Stop the Debugger (see page 442)
- How You Can Test an Application (see page 443)
- How Running in the Debugger Environment Works (see page 444)
- How You Can Use Break Conditions (see page 445)
- How You Can Log Trace Information (see page 456)
- How You Can Monitor a Running Application (see page 459)
- How Debugging an Application Works (see page 465)
- Debug an Imaged Application with the MonImage Utility (see page 482)

The OpenROAD interactive Debugger lets you monitor and manipulate all phases of execution such as event stacks, task tracing, stack back-tracking, action lists, and assignments to variables through a source-level display. The Debugger also lets you specify breakpoints or break events for easy identification of problems in a running application.

How You Can Use the Debugger

The OpenROAD Debugger is an application development tool that provides source-level debugging capabilities. The OpenROAD Debugger provides features that let you do the following:

- View and edit your scripts as an application executes
- Set break conditions
  - Break conditions such as breakpoints or errors cause the application to pass control to the Debugger.
- View and manipulate the application's log
  - You can control the type of trace information sent to the log, examine the log while the application is running, and direct the log output to an external file.
- View a dynamic graphic display of the current state of the application
- Run SQL queries
- View and manipulate the event queues
Start and Stop the Debugger

- Debug two or more frames at the same time to solve concurrency problems
- Reassign specific values to variables during debugging

You can perform these debugging functions individually or simultaneously. While the application is running in the Debugger, you can set breakpoints, step through code, monitor a session, do stack backtracing, examine data values, and examine the event queue. For more information, see the section that describes the type of testing you want to perform.

**Start and Stop the Debugger**

You can access the Debugger through OpenROAD menu commands and toolbar buttons and use it as your application runs.

You can set debugging functions before running your application or at any point while the application executes.

**To start the Debugger**

1. Click the Develop tab in OpenROAD Workbench.
2. Select the application to debug in the Applications portlet.
3. Click Run, Debug or click the Debug toolbar button.
4. The Debugging *application_name* window appears.

**To stop the Debugger**

1. In the Debugger, click File, Close.
   
   The Debugger: Closing with Stopped Application dialog appears.
2. Click Terminate Application.
   
   The Debugger closes.

**Note:** You cannot stop or kill a running application that is compute-bound.
How You Can Test an Application

OpenROAD Workbench lets you test your application at any point during the development process. Without leaving Workbench, you can run your application as the user sees it.

When you test an application, OpenROAD uses the working version, which includes any changes you have made to application components, even if they have not been saved or checked in. If there are compilation errors, OpenROAD does not run the application.

Debug a Single Application

You can debug applications one at a time in the OpenROAD Workbench.

To debug a single application
1. Click the Develop tab.
2. Select the application in the Applications portlet.
3. Click Run, Debug or click the Debug toolbar button.
4. The Debugging application_name window appears.

Debug Multiple Applications

You can debug several applications in the same database concurrently in OpenROAD Workbench. You can test any number of different applications (but not more than one instance of a single application) at the same time.

To debug multiple applications
1. Click the Develop tab.
2. Select the first application in the Applications portlet.
3. Click Run, Debug or click the Debug toolbar button.
4. The Debugging application_name window appears.
5. Select the second application in the Applications portlet.
6. Click Run, Debug or click the Debug toolbar button.
7. The Debugging application_name window appears.

If you are running more than one application, clicking File, Close stops only the selected application. To stop the other applications, close the windows containing those applications and click Terminate Application on the Debugger: Closing with Stopped Application dialog.
How Running in the Debugger Environment Works

This section describes how the OpenROAD Debugger affects a running application's environment.

Frame States

When you are running an application and using the OpenROAD Debugger, the behavior of the application may differ slightly from the behavior when the Debugger is not in use.

For example, in a running application, any changes to a frame that occur in an event block (data changes or visual property changes) are visible after the event block completes. However, when you use the Debugger to step through the code, you see such changes when the actual statement that causes the change completes. Therefore, you might see the frame in a state that you would not see in a free-running application.

Transactions

When you use the Debugger, you can stop the execution at any point. However, if you stop execution in the middle of a transaction, it may affect others' ability to work in the database because a transaction holds the locks it initiates until it is finished.

Errors

When an error break occurs, the application is in an indeterminate state. The break occurs at some point in the execution of the statement that caused the error, but it is impossible to determine exactly where. This means, for example, that any expression you examine might not show the expected or correct value.

Application Threads and Thread Identifiers

When you start an application, you open the first thread. Each time thereafter that your application issues an openframe statement, it begins a new thread in the application.

Generally, the user remains unaware of these individual threads in an application. However, when you use the OpenROAD Debugger, you must be aware of which thread you are working on, especially if your application uses the openframe statement to open several instances of the same frame.
To keep track of the various threads, the Debugger displays a thread identifier when appropriate. This identifier has the format:

framename:integer_number

framename
Identifies the starting frame of the thread

integer_number
Specifies a value that identifies the thread’s relative position in the sequence of threads in the application.

For example, if a thread identifier is framename:6, this thread is the sixth thread opened in the application. The integer_number also makes it possible to distinguish between threads originating from multiple openframe statements that open the same frame.

The threads are numbered consecutively from the start of the application, beginning with number 1. If you close a thread, its number disappears and is not reused. This behavior results in breaks in the sequence of thread identifier numbers as you step through your application.

Synchronization of Concurrent Threads

Although you can use the Debugger features to help you synchronize concurrent threads (for example, to determine whether frames are receiving the proper user and database events), the only way to ensure synchronization is to use the WaitFor method. For more information about this method, see the Language Reference Guide.

When you stop a thread to debug its code, user and database events "pile up" in the event queue of the thread’s current frame, and the thread sends no user or database events of its own. This behavior can distort the usual sequence of events.

How You Can Use Break Conditions

You can set source breakpoints or breaks for the following conditions:

- Events—user, database, or window manager events
- Errors—DBMS, data facility, communications, or other specific classes of errors
- Calls to frames and 4GL and 3GL components
- Method invocations in user or system classes
- Thread starts
When a break condition occurs, control passes to the Debugger so that you can examine the state of the program.

The OpenROAD Debugger recognizes the following types of break conditions:

- Source Breakpoints (see page 448)
- Event Break Conditions (see page 450)
- Error Break Conditions (see page 452)
- Call Break Conditions (see page 454)
- Method Invocation Break Conditions (see page 455)
- Thread Start Breakpoints (see page 455)

**Debug Tab**

You use the Debug tab of the OpenROAD Workbench to view, set, deactivate and reactivate, and delete break conditions. The application that is selected on the Applications portlet of the Develop tab is the application whose components will be displayed on the Debug tab.

Source components of the selected application are displayed in the Components portlet of the Debug tab. The component selected in the Components portlet is displayed in the Source Breakpoints portlet. This portlet is where you can view, set, and delete source breakpoints.

**Break Conditions**

You can set, deactivate, or delete break conditions. See the following sections for instructions.

**Set Break Conditions**

You set break conditions in a component script on the Debug tab of OpenROAD Workbench.

**To set a break condition**

1. Click the Develop tab and select the application you want to debug in the Applications portlet.
2. Click the Debug tab.
   The components of the selected application are displayed in the Source Components portlet on the left side of the window.
3. Select the component in whose script you want to set a breakpoint.
The script for the component is displayed in the Source Breakpoints portlet on the right side of the window.

4. Click the line number where you want to set the breakpoint.

You can also set source breakpoints using the View Processed Script option in the Frame Editor or the 4GL Procedure Editor.

For more information about types of break condition, see the appropriate section.

Deactivate and Reactivate Break Conditions

You can deactivate a break condition without deleting it.

To deactivate a break condition
1. Click the Develop tab and select the application containing the component with break conditions in the Applications portlet.
2. Click the Debug tab.
   The components of the selected application are displayed in the Source Components portlet on the left side of the window.
3. (Optional.) Select the component in whose script you want to deactivate a breakpoint condition.
   The script for the component is displayed in the Source Breakpoints portlet on the right side of the window.
4. Click the Activations portlet tab on the left side of the window.
5. Click the check box for the condition you want to deactivate so that it is unchecked.
   The Debugger will ignore a break condition while it is deactivated.

The deactivated breakpoint is still marked in the script, but is displayed only as an outline:

To reactivate a deactivated break condition
1. Click the Develop tab and select the application containing the component with break conditions in the Applications portlet.
2. Click the Debug tab.
   The components of the selected application are displayed in the Source Components portlet on the left side of the window.
3. Click the Activations portlet tab on the left side of the window.

4. Click the check box of the deactivated condition so that it displays a checkmark.

For instructions to delete break conditions, see How You Can Delete Break Conditions (see page 448).

**How You Can Delete Break Conditions**

There are several ways to delete break conditions:

- On the Call, Methods, or Other portlets tabs of the Debug tab, clear the break condition you want to delete.

- In the Activations portlet of the Debug tab, select the break condition you want to delete, then use any of the following methods to delete your selection:
  - Right-click on the condition and select Delete from the pop-up menu.
  - Press the Delete key.
  - Click Edit, Delete from the Workbench menu.

- To delete all break conditions listed in the Activations portlet, select one of the conditions and then right-click and select Clear from the pop-up menu.

For additional information about deleting source breakpoints, see How You Can Delete Source Breakpoints (see page 450).

**Source Breakpoints**

A *source breakpoint* is a defined place in a script where the application passes control to the Debugger. Source breakpoints can pass control to the Debugger only between statements. Therefore, when you are setting source breakpoints and you choose a line of code that is not the beginning of a statement, the Debugger searches back through the script until it finds the start of a statement and places the breakpoint there.

When a running application encounters a source breakpoint, it passes control to the Debugger just before the execution of the statement on which the breakpoint is positioned. If the line on which you placed the breakpoint contains more than one statement, the Debugger gains control before the first statement on the line.
Ways to Set Source Breakpoints

There are a number of ways to access a script to set source breakpoints:

**In the Frame Editor or 4GL Procedure Editor**

Click Tools, Processed Script/Breakpoints.

The script is displayed.

**On the Debug tab**

1. Click the Develop tab and then select an application in the Applications portlet.
2. Click the Debug tab.
3. Select the component in the Source Components portlet.

The script is displayed in the Source Breakpoints portlet.

**From the Develop tab**

1. Click the Develop tab and then select an application in the Applications portlet.
2. Select a component in the Components portlet.
3. Click Run, Debug.

The Debugger window opens, displaying the script.

**In the Debugger**

When a running application has stopped at a predefined breakpoint, the Debugger opens. You can set additional source breakpoints in the Debugger.

**Set Source Breakpoints**

You can set source breakpoints in a script in any of the facilities described in Ways to Set Source Breakpoints (see page 449).

**To set a source breakpoint**

Move to the line in the script where you want to put the breakpoint and click the line number.

The Debugger inserts a breakpoint at that line, as indicated by a Stop sign icon:

![Stop icon]

For more information on how to deactivate or reactivate a breakpoint, see Deactivate and Reactivate Break Conditions (see page 447).
Delete Source Breakpoints

A breakpoint appears as a Stop sign icon in a script. You can remove source breakpoints by deleting them from the script in either the Debugger or the Processed Script windows or on the Source Breakpoints portlet of the Debug tab.

To delete a source breakpoint
1. Open the facility to display the script with source breakpoints.
2. Click the Stop sign icon to delete a breakpoint.

Event Break Conditions

An event break condition causes the application to pass control to the Debugger at the beginning of the associated event block. You set event break conditions on the Breakpoints: Events, Errors and Threads portlet (Other portlet tab) of the Debug tab.

You can set event break conditions for the following events:

All Events
Set this condition to turn on all of the event's break conditions at once.

Recognized User Events
Set this condition for a defined OpenROAD user event.
OpenROAD activates the Debugger just before any user event block executes.

Unrecognized User Events
Set this condition for a user event that is not defined for OpenROAD.
OpenROAD passes control to the Debugger when a frame receives a user event for which it has no user event block or for which it is not explicitly waiting (that is, there is no WaitFor method waiting for that user event).
The Unrecognized User Event helps catch common programming errors that result from a frame’s receiving a user event for which it has no corresponding event block. The frame does not recognize the event and ignores it. This situation can occur because the event block is missing or the user event name is misspelled, either in the event block or in the SendUserEvent statement.

DB Events
Set this condition for database events.
The application passes control to the Debugger just before any database event block executes.
**Window Manager Events**

Set this condition for any event that is not a user or database event. Entry events, for example, are window manager events.

The application gives control to the Debugger just before the execution of any event block defined for a window manager event, that is, all event blocks except those for user or database events.

If a particular window event has no defined event blocks in the application, that event, if it occurs, does not cause a break in the application.

**Set Event Break Conditions**

You can set event break conditions on the Breakpoints: Events, Errors and Threads portlet (Other portlet tab) of the Debug tab.

**To set event break conditions**

1. Click the Develop tab and select the application in which you want to set event break conditions in the Applications portlet.
2. Click the Debug tab.
3. Click the Other portlet tab on the right side of the Workbench window.
   The Breakpoints: Events, Errors and Threads portlet is displayed.
4. In the Event Break Conditions section, select one or more types of events that you want to set as break conditions, or check All Events to set break conditions for all events.

Event break conditions are displayed in the Activations portlet of the Debug tab.

If you want to deactivate or reactivate a break condition, clear or select the appropriate check box in the Activations portlet.

**Break on Event Dialog**

When the Debugger encounters an event break condition that you set, OpenROAD stops the application execution and turns control over to the Debugger, which displays the Break on Event dialog containing the name of the event. The ability to edit user and database events is helpful when you are debugging because you can manipulate the event before it is executed.

Depending on the type of event that was intercepted, the format of this dialog and the options for further debugging will vary. If the event named is a user or a database event, you can edit the event and then click a button on the dialog to control the next step in the event break execution.
For a break on a user event, you can also edit the event's name and three of its message parameters (messageinteger, messagevarchar, and messagefloat). For a database event, you can edit the name or text of the event. For window manager events, you can only use the buttons to process the event.

If the event that caused the break is part of an event chain, the message at the top of the dialog does not necessarily name the actual event in the chain that caused the break. To see this name, click the Break button.

The buttons on the Break on Event dialog provide the following options:

**Break**
Continues the event break execution.
It displays the Debugger window positioned on the first statement in the event block for the event that caused the break. From here, you can perform all of the Debugger functions available from the Debugger window. For more information about the Debugger window, see How You Can Debug Applications.
In the case of chained events, the block on which the Debugger window is positioned may not be the block that is named in the message on the Event Break Conditions dialog.

**Ignore**
Skips the event named in the dialog.
If you select this option, execution continues as if the event did not occur.

**Continue**
Executes the event.
The application does not stop until the next break condition is encountered.

**Error Break Conditions**

An error break condition passes control to the Debugger when the error break that you set occurs. You can set an error break condition for any of the following error types:

- All errors
- DBMS errors
- Data facility errors that result from such situations as data type mismatches
- Communication errors that originate from the General Communication Facility (GCF)
- Specific error class or error number

The OpenROAD Debugger gains control at an indeterminate point in the execution of the statement that caused the error. After you return control to the application, error handling occurs as it would in a runtime environment. That is, the application handles the error as if you had not intercepted the error with the Debugger. In some cases, this behavior can take you to the end of the event block.

**Set Error Break Conditions**

You can set error break conditions in the Breakpoints: Events, Errors and Threads portlet (Other portlet tab) of the Debug tab.

**To set error break conditions**

1. Click the Develop tab and select the application in which you want to set error break conditions in the Applications portlet.
2. Click the Debug tab.
3. Click the Other portlet tab on the right side of the Workbench window.
   The Breakpoints: Events, Errors and Threads portlet is displayed.
4. In the Error Break Conditions section, select one or more types of errors that you want to set as break conditions, or check All Errors to set break conditions for all error types.
   Error break conditions are displayed in the Activations pane of the Current Break Conditions for Application window. If you want to deactivate or reactivate a break condition, select or clear the appropriate check box in the Activations portlet.

**To set an error break condition for a specific class of error or a specific error number**

1. Click the Develop tab and select the application in which you want to set event break conditions in the Applications portlet.
2. Click the Debug tab.
3. Click the Other portlet tab on the right side of the Workbench window.
   The Breakpoints: Events, Errors and Threads portlet is displayed.
4. In the Error Break Conditions section, select the Specific Error option.
5. Select the desired class from the Class drop-down list.

6. (Optional.) Specify a specific error number within the class: in the Number entry field, type the number in hexadecimal format.

   **Note:** The hexadecimal form of the error number is the four-digit number that appears directly after the facility code in Ingres error numbers. For example, if the error number is E_US0041, the hexadecimal number that you would enter is 0041.

   OpenROAD validates your error class and number entries, and alerts you if the class you specified does not exist or if your number is not a valid hexadecimal number.

The specific error break condition is displayed in the Activations portlet of the Debug tab. If you want to deactivate or reactivate a break condition, select or clear the appropriate check box in the Activations portlet.

---

**Call Break Conditions**

A call break condition causes the application to give control to the OpenROAD Debugger when the frame or procedure you specify is called. The Debugger takes control just before the statement that calls the frame or procedure.

   **Note:** You cannot set a call break condition on a database procedure call.

**Set Call Break Conditions**

You can set call break conditions in the Breakpoints: Calls to components in application portlet (Calls portlet tab) of the Debug tab.

**To set a call break condition**

1. Click the Develop tab and select the application in which you want to set call break conditions in the Applications portlet.

2. Click the Debug tab.

3. Click the Calls portlet tab on the right side of the Workbench window.

   The Breakpoints: Calls to components in application portlet is displayed, showing all of the called components in the application.

4. Select the Break option for as many calls as desired.

Call break conditions are displayed in the Activations portlet of the Debug tab. If you want to deactivate or reactivate a break condition, select or clear the appropriate check box in the Activations portlet.
Method Invocation Break Conditions

A method invocation break condition causes the application to pass control to the OpenROAD Debugger whenever it encounters the specified method in the code. The Debugger receives control immediately before the statement that invokes the method.

Set Method Invocation Break Conditions

You set a break condition for methods on the Breakpoints: Method calls portlet of the Debug tab. You can select methods from system classes or user classes.

To set a break condition on a method
1. Click the Develop tab and select the application in which you want to set method break conditions in the Applications portlet.
2. Click the Debug tab.
3. Click the Methods portlet tab on the right side of the Workbench window.
   The Breakpoints: Method calls portlet is displayed.
4. Select an object type: User Classes or System Classes.
   If you select System Classes, a list of system classes is displayed in the Class list. If you selected User Classes, a list of user classes is displayed if any user classes are defined for your application.
   When you choose a class, a list of methods defined for that class is displayed under Method.
5. Select a method in the Methods list and then click Set Breakpoint.
   A breakpoint is set for the selected method.
6. (Optional.) Repeat the previous two steps to set breakpoints for other methods.

Method break conditions are displayed in the Activations portlet of the Debug tab. If you want to deactivate or reactivate a break condition, select or clear the appropriate check box in the Activations portlet.

Thread Start Breakpoints

A thread start breakpoint condition causes the application to pass control to the OpenROAD Debugger when the application is first started and after each openframe and callframe statement that the application issues.
Set Thread Start Break Conditions

You set a break condition for a thread start on the Breakpoints: Events, Errors and Threads portlet (Other portlet tab) of the Debug tab.

**To set a breakpoint for the start of a thread**

1. Click the Develop tab and select the application in which you want to set thread break conditions in the Applications portlet.
2. Click the Debug tab.
3. Click the Other portlet tab on the right side of the Workbench window.
   The Breakpoints: Events, Errors and Threads portlet is displayed.
4. In the Thread Break Conditions section, select the Thread Start option.

The thread break condition is displayed in the Activations portlet of the Debug tab. If you want to deactivate or reactivate a break condition, select or clear the appropriate check box in the Activations portlet.

How You Can Log Trace Information

OpenROAD maintains one log for each application. The Monitor tab lets you choose, view, and save log information for any or all of the following options as your application runs:

**4GL**

**Events**

Traces events providing notification of the beginning and ending of events

**DB**

Traces messages from the DBMS server similar to those generated by the set printqry statement

**Calls**

Traces entries and exits, and displays the arguments passed to frames or procedures on call statements
How You Can Log Trace Information

Database

Error Messages
Displays all Ingres error messages

GCA
Displays messages between Ingres processes

Trace Window
Shows OpenROAD messages displayed in the standard output window or generated by the Trace method

Other

Time Profiling
Displays messages that tell you how much CPU time and how much real time have been used by the application and the application development environment.

The time-profiling option provides information about the time used for event handling and calls to frames or procedures. When time-profiling is enabled, each logged event or call is bracketed in the log by lines of the form:

CPU: aaaaaaa ms, REAL: ssss sec (BEGIN n)
CPU: bbbbbbb ms, REAL: tttt sec (END n)

aaaaaaa and bbbbbbb
Specify times in milliseconds

ssss and tttt
Specify times in seconds

n
Specifies an integer that identifies an event or call

To calculate the elapsed time for handling an event or executing a call, subtract the ending time from the beginning time.

Report

For future implementation

Lines to Display
Provides a slider bar that lets you set the number of lines that are accessible in the Trace Log portlet

Limits: 24–800
Default: 240
Trace Log Porlet

To access the Trace Log portlet, click the Monitor tab, and then click the Trace Log portlet tab. The Trace Log portlet displays the end of the application log.

**Note:** The Lines to Display slider field on the Trace Configuration portlet (Configure portlet tab) lets you set the number of lines that are displayed in the Trace Log portlet. For example, if the number is 240, you can view the last 240 lines of the log. For more information, see How You Can Log Trace Information (see page 456).

You can select the types of information you want logged by selecting the options on the Trace Configuration portlet (Configure portlet tab). For more information, see How You Can Log Trace Information (see page 456).

How You Can Use the Trace Log Portlet

When the Trace Log portlet on the Debug tab is selected, the Run menu provides the following options:

**Write Log to File**

Writes the current log to a file that you specify. If Lines to Display is $n$, this command writes at most $n$ lines.

**Redirect Output to File**

Writes all log data to a file that you specify. Log data is written to the file until you redirect the log output to a different file or you select a different application on the Develop tab.

**Clear Log**

Clears the data displayed in the Log for Application window. If you have redirected output to a file, this operation does not affect the contents of the file.

**Add Message to Log**

Inserts a text string into the log at the cursor position.

How You Can Control Query Tracing and Execution

When the Trace Log portlet on the Debug tab is selected, the Run, Trace Log menu lets you control query tracing and query execution plan tracing.

The following options are available on the Run, Trace Log menu:

**PrintQry on 'Go'**

Issues a set printqry statement when you start the application.

**QEP on 'Go'**

Issues a set qep statement when you start the application.
How You Can Monitor a Running Application

The Monitor tab displays the current state of a running application using two areas of the window:

- **Call Stack portlet**—A call stack of the currently executing thread in the application
- **Thread Map portlet**—A visual thread map diagram, with each node in the diagram representing one thread in the application

The Monitor tab is continually refreshed as the application executes so that the display is always accurate and current.

**Current Thread Display**

The Current Thread area displays the thread identifier of the current thread, that is, the last thread to execute. Below that, the Debugger displays a condensed version of the current thread's call stack.

This display can be useful when trying to determine the execution path in an application. In a large application, it may not always be obvious how a particular procedure gets executed. To identify the path or paths for the procedure, set a breakpoint in the procedure, turn on the Monitor, and run the application. Each time the breakpoint for the procedure is reached, the Monitor displays the call stack of the current thread in the application with the target procedure at the top.

**Thread Map**

The Thread Map portlet on the Monitor tab provides a visual representation of the current application threads and their relationships. Each rectangular node on the tree represents one thread. The lines connecting the nodes show the parent-child relationships, with the parent displayed above its children. If a frame was opened with an openframe statement that set the parent frame to null, the word NULL appears as the parent.
For each node, OpenROAD provides the following information about the thread it represents:

**Thread identifier**
Tells you which frame began the thread (for example, main_control:1). The number after the starting frame's name indicates when, in relation to other threads, the thread began. For example, if the number is 3, that thread is the third thread started in the application.

**Name of the currently active executable component of the thread**
Displays the name of the current frame, procedure, or user class method in parentheses

**Status of the thread**
Is displayed in the third line of the rectangle representing the thread

You can access the following options from the pop-up menu that appears when you right-click any thread box in the tree structure:

**Call Stack**
Displays the thread's detailed call stack.
For information about using the call stack, see Call Stack Window (see page 460).

**Event Queue**
Displays the event queue for the thread's current frame.
For information about using the event queue, see Event Queue Window (see page 462).

**Debugger Window**
Displays the window for the thread's current frame.
For information about this window, see How You Can Use the Debugger Window (see page 467).

---

**Call Stack Window**

Each thread in an application has a separate call stack, and you can open a Call Stack window for each thread. The call stack shows all calls to procedures, frames, and user class methods.
To open the Call Stack window for a thread

Use any of the following methods:

- In the Debugger window for the thread, click Tools, Call Stack.
- In the Debugger window, click the Call Stack toolbar button.
- On the Thread Map portlet of the Monitor tab, right-click a thread box and select Call Stack from the pop-up menu.

The Call Stack window displays lists of frames and procedures in the order in which they are called, with the most recently called frame or procedure at the top of the list.

After being opened, a Call Stack window remains open until you close it or until the thread terminates.

Note: While a thread is running, the Call Stack window is not updated automatically and does not always reflect the current state of the thread. However, you can always display the current state of the call stack by clicking File, Snapshot in the Call Stack window.

If a frame or procedure in the call stack was called with arguments, the name of the frame or procedure in the Call Stack window is preceded by an icon. You can click the icon to see the arguments passed to the procedure or the frame when it was called. If any of the arguments is an array or reference variable, the argument is displayed with an icon, and you can click the icon to expand the display and see the values of the attributes.

The values shown for the arguments are the values that are passed to the called frame or procedure from the calling frame or procedure. The call stack reflects any changes that the called frame or procedure makes to the following:

- Arguments of varchar data type
- Attributes of any argument that is a reference variable

With the exception of varchar variables, the call stack does not reflect any changes made by the called frame or procedure to arguments that have a simple data type. Additionally, if the called frame or procedure reassigns a reference variable that it received as a parameter, any subsequent changes to the attributes of that variable are not reflected in the call stack.
In addition to the names and argument values for frames and procedures, the Call Stack display provides other useful information:

- When a frame is opened with an openframe statement so that it is the first frame on the call stack, the name of the calling frame and its thread identification are displayed.

- The current frame is always shown as the last item on the call stack even if it was started with a gotoframe statement. If a frame is started with a gotoframe statement, it replaces the last frame or procedure on the stack that was started with a call statement. If the last frame on the stack resulted from a series of gotoframe statements with no intervening calls, expanding the information for the frame provides a summary of the previous gotoframe statements.

- When you click an icon associated with a frame, any attributes passed using a with clause are displayed along with any parameters that were passed. An attribute passed using a with clause can be distinguished from an argument because "WITH" appears at the beginning of the line.

- If an argument was passed by reference, the description of the argument ends with ",BYREF."

**Event Queue Window**

The Event Queue window lets you examine the queue of events waiting for a frame and to manipulate the queue by deleting and adding events.

**Event Queue Command**

To open the Event Queue window for a thread, use any of the following methods:

- In the Debugger window, click Tools, Event Queue.
- In the Debugger window, click the Event Queue toolbar button.

- In the Thread Map portlet of the Monitor tab, right-click the thread box and select Event Queue from the pop-up menu.

The Event Queue window shows all the user and database events queued for a given frame. When you select an event in the list, information about the selected event is displayed on the right side of the window.
The information displayed depends on the type of event:

- **User event**
  The information displayed includes the event name, the values of the event parameters, the sender, the time the event was sent, and the delay, if any. You can edit the event name and its parameters (MessageVarchar, MessageFloat, and MessageInteger).

- **Database event**
  The information displayed includes the event name, its owner, and the associated text. You can edit this information.

The Event Queue provides the following commands on the File menu:

- **Queue New DB Event**
  Displays the New DB Event dialog that lets you add a database event to the end of the queue

- **Queue New User Event**
  Displays the New User Event dialog that lets you add a user event to the end of the queue

- **Remove Delay**
  Removes the delay period if one was specified for a user event

- **Delete Event**
  Removes the selected event from the queue

- **Select Frame**
  Lets you select a new frame and changes the display in the Event Queue window to the event queue for the newly selected frame

- **Snapshot**
  Refreshes the Event Queue window

- **Close**
  Closes the Event Queue window

---

**How You Can Debug Database Queries**

The Query tab lets you run SQL queries. You can insert a query into the current database session of your application, or you can open a new session. If you issue a statement that returns data, the results are displayed in the Query Results portlet.
How You Can Specify a Database Session

The Database Connections portlet on the Query tab lets you specify the session in which your queries run. You can open a new session or select one of the sessions of the currently executing application.

Create a New Database Connection

You can create a new database connection and session in the Database Connections portlet of the Query tab.

To create a new database connection
1. Click to select the Database Connections folder.
2. Right-click the Database Connections folder and select New from the pop-up menu.
   The Connect to DB dialog appears.
3. Enter the name of the database and any flags, and then click OK.
   The database session is added under the Database Connection folder, and the connection is made.

You can open a session with the database that the application uses or any other database. It is not necessary for an application to be running to use this operation.

Open a New Session

You can open a new session with a database in the Database Connections portlet of the Query tab.

To open a new session with a database
1. Click the plus sign (+) to open the Connection Profiles or Database Connections folder.
   If you opened the Connection Profiles folder, open a connection profile to display the database to which you want to connect.
2. Right-click the database you want to connect to, and select Connect from the pop-up menu.
   The database session is started.

You now can enter query statements in the Database Queries portlet.
Specify One of the Sessions of the Current Application

You can specify a session with the currently executing application in the Database Connections portlet of the Query tab.

To specify one of the sessions of the currently executing application
1. Click the plus sign (+) next to the Running Applications folder. OpenROAD displays any applications currently running.
2. Click the plus sign (+) next to the running application whose database session you want to use. Each database session is listed by its session identifier and the database to which that session is connected.
3. Click the active database session.
4. Click OK.

Run a Query

You can run a query from the Database Query portlet on the Query tab. To issue query statements, you must be connected to a database session. For instructions, see one of the procedures under How You Can Specify a Database Session (see page 464).

To run a query
1. Enter your query statement in the Database Query portlet.
2. Click Run, Run. OpenROAD displays the results, if any, in the Query Results portlet.

To clear the query results, select the results and press Delete.

How Debugging an Application Works

The Debugger window is where you do your source debugging. This window displays the processed script of a frame or procedure.

There is potentially a Debugger window for each active thread in your application. The Debugger window for a thread typically appears the first time the application encounters a break condition in the thread.

You can open a Debugger window in a variety of contexts in OpenROAD Workbench. One way is described in Open the Debugger for an Application (see page 466). After it is opened, a Debugger window remains open until you close it with File, Close or until the thread terminates.
In the Debugger window, you can perform the following functions to help you debug your application:

- Set and remove source breakpoints
- Set watch values and open a Watch window
- Customize function keys and tooltip text
- Examine expressions and assign values to them
- Display a thread’s call stack
- Display a frame’s event queue
- View and edit a selected script
- Select trace log options

When a break condition occurs and control passes to the Debugger window, the window displays the processed script of the currently active frame or procedure. If the script is a frame script, the display includes any field scripts associated with the frame; the field scripts are appended to the end of the frame script.

**Open the Debugger for an Application**

You can open the Debugger from the Develop tab.

**To open the Debugger window**

1. Click the Develop tab.
2. Click the application you want to debug in the Applications portlet.
3. Click Run, Debug.
   The Debugger window appears.
How You Can Use the Debugger Window

The title of the Debugger window has the following format:

Debugging application, component:n

application
  Specifies the name of the application

component
  Specifies the name of the opening frame or procedure of the thread

n
  Specifies the thread number

The Debugger window has two main display areas separated by a line between the areas:
- If there is a current script, the processed script is displayed in the upper area.
- The lower area is for displaying data values.

To change the relative size of a display area, drag the separator up or down. If you resize the window, the two display areas retain their relative proportions.

The text above the script display indicates what component the script belongs to, the status of the script in the thread, and the level of the script in the thread. The text has the form:

Script for component name [ status in thread [ call level]]

component name
  Specifies the name of the frame or procedure

status in thread
  Indicates whether the component is part of the thread and whether it is the current component

call level
  Indicates the level of the component in the thread's call stack

The script information bar also contains the following icons on the left side:

Examine
  Lets you examine the value of an expression

Assign
  Lets you assign a new expression to a variable
How Debugging an Application Works

**Watch Value**

Opens the Watchpoint dialog, where you can set a value to watch. For more information, see Set a Watch (see page 478).

**Restore Down/Maximize**

Maximizes or restores the original window size of the Debugger.

The Debugger window’s menu selections and toolbar buttons let you move through the script, examine selected expressions, watch for changes in the values of expressions, assign values to expressions, and display the call stack and event queue. These selections are described in detail in How You Can Control Execution of Your Application (see page 470).

Additional menu selections let you select a new script, move around in the script, and edit a script. These operations are described in detail in the following subsections.

Some of the Debugger menu selections let you customize the Debugger by selecting settings to suit your preferences. The Debugger Options menu lets you determine when these settings are saved, if at all. Saving the settings means that they remain in effect for the current session as well as for future Debugger sessions.

**Script Display Icons**

The OpenROAD Debugger uses the following icons to mark the text lines in a script:

**Horizontal Arrow**

Points to the line containing the next statement to be executed. When a break occurs, execution normally stops before the execution of a 4GL statement. However, when an error break occurs, it is possible that execution has stopped at some point during the execution of a 4GL statement. If there is more than one statement on a line, the next statement to be executed may not be the first one.

**Bent Arrow**

Marks the call statement in progress in the script for a component in the call stack (but not the last component in the stack).
How Debugging an Application Works

**Stop Sign**
Indicates an active source breakpoint

**Stop Sign Outline**
Indicates an *inactive* source breakpoint, one that was toggled off in the Activations pane of the Current Break Conditions for Application window

*Note:* If an arrow points to a line where there is a source breakpoint, the arrow is superimposed on the Stop sign or Stop sign outline.

**Display a Different Script**

By default the Debugger window displays the script associated with the current frame or procedure. You can display a different script in the Debugger window.

**To display a different script**
1. Click File, New Script, Specify.
   
   OpenROAD opens the Select New Script dialog, which displays a list of available components in the current application.

2. Select a component and then click OK.
   
   The Debugger displays the script for that component in the Debugger window.

Other options on the New Script menu let you navigate among the scripts of components currently in the call stack:

**Up**
Displays the script for the previous component in the call stack if the currently displayed script is for a component in the call stack, this operation. For this purpose, a local procedure is treated as a separate component.

**Down**
Displays the script for the next component in the call stack if the currently displayed script is for a component in the call stack and it is not the last component in the call stack. For this purpose, a local procedure is treated as a separate component.

**Current**
Displays the currently executing script, if any
How Debugging an Application Works

How You Can Control Execution of Your Application

You can invoke the Debugger by performing the procedure in Open the Debugger for an Application (see page 466). When the Debugger has control, you can use the following navigation buttons on the Debugger window’s toolbar to control the execution of your application:

- **Step**
  
  Executes a single statement, stepping across frame or procedure calls. For more information, see Step Button (see page 470).

- **Step Into**
  
  Executes a single statement, but steps into a frame or procedure if the statement calls a frame or procedure. For more information, see Step Into Button (see page 471).

- **Step To End**
  
  Continues execution to the next end, return, or resume statement. For more information, see Step To End Button (see page 472).

- **Continue**
  
  Continues execution to the next break condition. For more information, see Continue Button (see page 472).

- **Stop Debugger**
  
  Stops debugging the application or component and closes the Debugger window.

You can use these buttons to move through your application—stopping when you want to examine the values of expressions or variables, to look at the call stack or event queue, or to perform other debugging activities. These buttons are described in greater detail in the following sections.

- **Step Button**
  
  Click the Step button when you want to step through the code one statement at a time. Typically, each time you click Step, the system executes a single statement. The behavior may vary, however, in certain contexts.

  As you step through a statement block or event block, the Debugger stops on the *end* statement that closes the block. This enables you to examine the results of the operations performed in the statement or event block.
When you step out of an event block, control returns to the application until another event occurs for which there is an event block. At this point, you may need to interact with the application to cause an event to happen. However, having stepped out of an event block, the Debugger stops at the first statement of the next block to be executed.

If you step into a statement that opens a pop-up dialog, the application has control until you close the pop-up.

If the next statement calls a frame or procedure, the Step button steps across the call; that is, the application receives control until the call completes. Control returns to the statement immediately following the call. If an intervening break condition is encountered before the call completes, the Debugger gains control. When the call does complete, however, the Debugger stops execution at the statement following the call statement.

Finally, if the statement is a gotoframe statement, clicking Step moves you to the script for the frame invoked by the gotoframe statement.

**Step Into Button**

The Step Into button behaves like the Step button in all cases except the following:

- When the next statement to be executed is a callframe statement.
  The Step Into button moves you into the script for the frame and stops at the first statement in the initialize block. If the called frame has no initialize block, there is no statement to execute and you must click Step, Step Into, or Continue to display the frame and continue execution.

- When the next statement is a callproc statement.
  The Step Into button moves you to the script for the called procedure and stops at the first executable statement.

- When the next statement invokes a method of a user class.
  The Step Into button moves you into the script for the method and stops at the first executable statement.

- When the next statement is openframe.
  If a new thread is created, a new Debugger window is opened. The Debugger stops execution at the first statement of the initialize block of the new frame. If there is no initialize block, you must click Step, Step Into, or Continue to display the frame and continue execution.
There is an exception if the called frame, procedure, or method is in an included application, and the included application is in an image file. In this case, Step Into behaves like Step because no source is available. If the included application is in the database, however, the source is available.

**Step To End Button**

The Step To End button continues script execution until the next end, return, or resume statement in the current block or procedure.

If a break condition is encountered before reaching an end, return, or resume, the break will occur. If you continue execution after handling the break, the Debugger still takes control at an end, return, or resume as if the intervening break had not occurred.

**Note:** The control of execution during the watch state is the same as when the application is running in a normal state except that the Step To End choice is not available. Control can be exercised by selecting Step, Step Into, or Continue while in the watch state of execution.

**Continue Button**

With two exceptions, the Continue button causes execution to continue until a break condition is encountered or until the application terminates.

One exception is when you have used the Step button, but a break occurs before reaching the statement following the statement where you issued the step command. In this case, if you click the Continue button, execution continues only until you reach the statement following the one where you stepped.

The other exception is when you have used the Step To End button, and a break occurs before reaching the next end, return, or resume in the procedure or block where you used the Step To End button. In this case, if you click Continue, execution continues only until you reach the first end, return, or resume statement in the block or procedure where you executed the Step To End command.
How You Can Examine Values of Expressions

As you debug an application, you can examine the value of expressions using the Debugger window.

**To examine an expression**

1. In the Debugger window, drag the cursor over the expression in the script to select it.
2. Do one of the following:
   - Click the Examine toolbar button.
   - Click Tools, Values, Examine.
   - Right-click the selected expression and select Examine from the pop-up menu.

If you have not selected an expression when you click Examine, or if the selected expression is invalid, the Selection Popup dialog appears, letting you correct or enter the expression that you want to examine.

When you examine an expression, the Debugger displays the value of the expression in the data display area of the Debugging window:

- If the value of the expression is a simple data type, the display has the form:

  \[ \text{expression} = \text{type}, \text{value} \ n \]

  - **expression**
    - Specifies the expression
  - **type**
    - Specifies its data type
  - **n**
    - Specifies its value
If the value of the expression is an object other than an array, the display has the form:

\[ \text{expression} = \text{classname} \ 'objectname' \]

or

\[ \text{expression} = \text{Unnamed classname} \]

- **expression**
  - Specifies the expression

- **classname**
  - Specifies the class of the object

- **objectname**
  - Specifies the name of the object

If the value of the expression is an ArrayObject, the display has the form:

\[ \text{expression} = \text{Array of classname} \]

- **expression**
  - Specifies the expression

- **classname**
  - Specifies the class of the objects in the array

Because two local procedures in the same main script can define identically named local variables, the Debugger tracks the scope for all expressions.

The scope of the currently executing line is used when the expression's value is displayed. This value is tracked in future displays. If you leave the script for which the scope of the variable is valid, the data display indicates that there is no valid value in the current scope.

If you do not want the Debugger to display these invalid variables, select the Hide Variables Not Valid in Current Scope option from the Examine Values pop-up menu. A checkmark appears to the left of the option.

When the value of an expression is an object, a plus sign (+) precedes the line where its value is displayed. Click the plus sign to expand the display and view the object's attributes. If any of its attributes are objects, they are likewise preceded by a plus sign and can be expanded in turn.

To close an expanded object, click the minus sign (-) that precedes the object.
When an ArrayObject has been expanded, an array type node follows the arrayobject class and its attributes. This node displays the class name of the row objects and the number of rows in the array. For an array of stringobject with 16 rows, the array type node is displayed as follows:

```
from stringobject[], 16 rows
```

When the array type node is expanded, the array row objects are displayed for the array.

For large arrays, the array row objects are displayed in groups of rows under group headings. Each heading displays the range of rows in its group. A group heading for rows 1 through 20 of an array has the following format:

```
arrayname[1] to arrayname[20]
```

- `arrayname` Represents the name of the array

For the settings that define a large array and the size of the groups for the array, see Select Array Display Settings.

**Write Expanded Values to the Trace Window**

Rather than expanding an object in the Debugger window, you can write the expanded values to the Trace window.

**To write expanded values to the Trace window**

1. Select an object in the script display area or the data display area.
2. Click Tools, Values, Expand in Trace Window, or right-click the highlighted object and click Expand in Trace Window from the pop-up menu.
   
   The Number of Levels dialog appears, which lets you select how many indent levels to display. If you select the Prompt to Continue Output option, a slider field appears, letting you select the number of rows to be examined before you are prompted to continue or cancel.
3. Click OK.

The Debugger updates the values of the examined expressions in the data display area when the executing application updates them or when you assign new values in the Debugger window.

If the execution of a statement makes an expression invalid, the display indicates that this is the case. For example, suppose you have examined the expression "arrayname[7].num." If a statement that clears the array is executed, the expression is no longer valid, as indicated by the display.
After you examine an expression, the resulting value remains in the data display (and is updated as necessary) until you explicitly delete it. To delete lines in the data display, click Edit, Delete or Edit, Delete All.

How You Can View Object Variables

You can determine how you want to view object variables in the Debugger by clicking View, Examine Values from the Debugger menu. Options on the Examine Values submenu include the following:

As List with Object Heading
- Lists attributes for the variable by object class with the class name in parentheses as a heading for each class

As List with no Heading
- Lists attributes for the variable by object class with no heading given

By Object Node
- Groups attributes by object class under an object node for the class. The object node for the class can be expanded to display the attributes that belong to the class.

Default: As List with Object Heading

Select a Display Option

You can select a display option for the Debugger window.

To select a display option
1. Click View, Examine Values in the Debugger window.
   - A pop-up menu appears.
2. Select or clearing an option from the menu by dragging the cursor over the option and clicking.
   - A checkmark to the left of the option indicates that it is selected.

In the expanded display, attributes are listed under the system class for which they are defined. The system classes are listed in hierarchical order from highest to lowest. For example, in most listings, the Object system class appears first with its defined attribute, Classname.
Select Array Display Settings

The Array Display Settings dialog lets you change the settings that determine how the rows of an array are displayed in the Examine window. The two settings you can select follow:

Display rows in groups of

- Lets you set the number of rows in a group
  - **Default:** 20 rows
  - **Limits:** Minimum value is 2 rows; maximum is 100

If number of rows is more than

-Lets you set the size of an array large enough to have its rows grouped together for display
  - **Default:** 40 rows
  - **Limits:** Minimum value is 2 rows; maximum is 100

To select array display settings

1. Click View, Examine Values, Array Display Settings.
   The Array Display Settings dialog appears.
2. Change one or both of the slider fields.
3. Click OK.

Any change to the settings takes effect immediately for all array objects appearing in the Examine window.

Assign Values to Expressions

You can use the Debugger window to assign specific values to variables within the current scope.

To assign a value to a variable

1. Click Tools, Values, Assign, or right-click the selected variable and select Assign from the pop-up menu.
   The Assign dialog appears.
2. Enter the name of the variable whose value you want to change and the value that you want to give to the variable.
3. Click OK.

The OpenROAD Debugger displays a confirmation message if the assignment is successful or an error message if it is not.
How You Can Use the Watch Facility

The watch facility examines selected local and global variables to determine whether a change in their value has occurred or their specified value has been reached.

Control of execution during the watch state is the same as when the application is running in a normal state except that the Step To End choice is not available. Control can be exercised by selecting Step, Step Into, or Continue while in the watch state of execution.

The watch facility places the Debugger in a watch state whenever there are entries in the Watch window. You can use the Debugger window to examine the value of watch items line by line until one or more of the watch values are reached or a watch change in value is triggered.

The Tools, Watchpoint menu command lets you select a variable as a watchpoint. The value entered for the variable is the watch value. After a watch value is reached, the Debugger is given control at the beginning of the next statement following the one that set the watch value.

While the Debugger has control, you can do the following:

- Examine new variables
- Enter new watch values
- Delete variables from either the Examine window (the data values display area) or the Watch window
- Terminate the watch by deleting all of the variables in the Watch window

Execution continues when you again select one of the execution control options.

Set a Watch

You can access the watch facility from the Debugger window.

**To access the watch facility**

1. Select a variable in the displayed script.
2. Click Tools, Watchpoint, Watch Value, or right-click the highlighted variable to display a pop-up menu and select Watch Value.
   
   The Watch Values dialog appears.
3. Enter the desired watch value, or select the Watch for Change option to start a watch for any change in value.
4. Click OK.
Limit a Watch to the Current Scope

You can limit the watch to the current scope in the Debugger window.

To limit the watch to the current scope

Click Tools, Watchpoint, Watch in Current Scope Only.

Open the Watch Window

You can open the Watch window from the Debugger window.

To open the Watch window

Click Tools, Watchpoint, Open Watch Window.

The Watch window appears to the right of the data value display area in the lower section of the Debugger window.

The watch item is displayed along with the value selection. If an asterisk (*) appears in the Value column for the watch variable, any change in value of the variable triggers a break in execution.

Close the Watch Window

You can close the Watch window from the Debugger window.

To close the Watch window

Click Tools, Watchpoint, Close Watch Window, or select a variable and right-click to display a pop-up menu from which you can select Close Watch Window.

Turn Off the Watch Facility

You can turn off the Watch facility from the Debugger window.

To turn off the Watch facility

Click Tools, Watchpoint, Turn Watch Off, or select a variable in the Watch window, and right-click to display a pop-up menu from which you can select Turn Watch Off.
How You Can Customize Function Keys and Tooltip Text

The Customize facility lets you change the function key assignments for any menu command for any of the windows in the Debugger environment. You also can display the function key name in the tooltip.

You can invoke the Customize command from the Debug menu or from the Tools menu in the Debugger. Keyboard and Toolbar options are available from the slide-off menu.

Customize the Keyboard

You can customize function keys or change key assignments using the Customize Function Keys window.

To customize function keys
1. Click the Develop tab and select an application in the Applications portlet.
2. Click Run, Debug.
   The Debugging window opens, displaying application code.
3. Click Tools, Customize, Keyboard.
   The Customize Function Keys window appears.
4. Select the facility to which you want to assign or reassign a function key and click Assign.
   The Assign dialog appears.
5. Press the key or key combination to assign to the facility.
   Errors, if any, are displayed above the Clear button.
   **Note:** To clear the assignment and start over, click Clear.
6. Click OK to accept the displayed assignment and close the Assign dialog.
7. (Optional.) Repeat the preceding steps to assign additional key commands.
8. Click File, Save to save the assignment and close the Customize Function Keys window.

Even though the assigned keys have been saved, the foregoing procedure can be used to modify key assignments at any time. The assigned keys can also be restored to the default assignments by clicking the Defaults button on the toolbar.
Add a Speedkey Name to Tooltip Text

You can add a function key assignment to tooltip text.

To add the name of the speedkey to tooltip text:

Click Debug, Customize, Toolbar, Display speedkey name in tooltip.

Edit a Script

When a processed script is displayed in the Debugger window, you can invoke the Script Editor to edit the main script or any included script or field script.

To edit a script

1. Place the cursor on the line you want to edit in the Debugger window.
2. Click Edit, Edit Script at Line.

The Script Editor displays the script that contains the line you selected. If the line with the cursor is in an included script or a field script, that script is the one to be edited.

Note: If you are already editing the script elsewhere in the OpenROAD Debugger or in the application development environment, you cannot edit it from the Debugger window.

When you make changes in a script associated with an open frame, the Debugger window changes its display to reflect changes in the script. At this point, although you see the changed script in the Debugger window, the application is still executing the old script. If the application stops on a break condition, the icons in the Debugger window may no longer point to the correct line because the icons are referencing line numbers in the old script.

Your changes to the script are not visible to the application until you close all instances of the frame and then reopen at least one instance of the frame. When you reopen the frame, OpenROAD automatically recompiles the script, making the changes available to the application.

Note: When you recompile, you must reset source breakpoints to ensure that they reference the correct line of the script.
Debug an Imaged Application with the MonImage Utility

Although you cannot debug an image file because the source code is not present, you can use the MonImage utility to identify where a problem is occurring in an imaged application. This utility lets you use the Log, Monitor, and DB Session for Application windows when you are running an image file.

Note: You can also use the equivalent facilities on the Monitor tab. For more information, see How You Can Monitor a Running Application (see page 459).

To debug an application using MonImage

1. Enter the following at the command line:
   
   w4gldev runimage iidebug.img
   
   The MonitorImage window appears.

2. Enter the image name that you want to run in the Image entry field.
   
   Alternatively, click Image, Select Image to access a standard File Selection dialog from which you can select the image file.

   When you enter or select the name of the image that you want to run, the utility automatically fills in the Starting Component and Database fields if you specified those defaults for that image.

3. Click Image, Go to run the image.
   
   If desired, you can change these defaults before running the image. If no defaults were specified, you must provide valid entries before running the image.

   The buttons at the bottom of the Monitor Image window display the Log for Application window, the Monitor window, and the DB Session for Application window, respectively. The Log for Application window often can be used to identify the source of a problem or to monitor the performance of your application.
Appendix A: SQL Syntax Summary

This section contains the following topics:

- **Dynamic Names in Syntax** (see page 483)
- **SQL Syntax Summary** (see page 484)
- **OpenSQL Syntax Summary** (see page 487)
- **Distributed Ingres Syntax Summary** (see page 487)

This appendix contains syntax summaries of SQL statements that have not been described in previous chapters. You can include these statements in your OpenROAD scripts.

A syntax summary is included in this appendix for:

- Standard SQL
- OpenSQL and the Distributed Ingres extensions.

If you are accessing databases through Ingres Enterprise Access, refer to the syntax in OpenSQL Syntax Summary (see page 487). Distributed Ingres Syntax Summary (see page 487) contains the syntax for Ingres Distributed Option architecture.

**Dynamic Names in Syntax**

Some of the parameters for these additional SQL statements can be dynamic names. In this appendix, dynamic names are shown underlined. They can be one of the following:

- A variable whose value is supplied at runtime.
  
  Variables are shown in *italics* in the syntax.

- A keyword that is specified at runtime.
  
  Keywords are shown in **bold** in the syntax.
  
  To specify a keyword as a dynamic name, use a string variable assigned with the exact spelling of the keyword.

In the syntax, each item underlined is specified as a separate dynamic name. For alternative choices (separated by "|"), only one of the dynamic name items can be specified.
SQL Syntax Summary

In addition to the 4GL statements in the *Language Reference Guide* online help, you can also include SQL statements in your 4GL scripts.

The following is a syntax summary of allowable SQL statements, with parameters that can be 4GL names underlined. If you have the Knowledge Management Extension, you will find SQL statements that support this feature in Knowledge Management Extension Syntax (see page 486).

```sql
copy [table] tablename
    (columnname = format [with null [(value)]]
    [, (columnname = format [with null [(value)]]))
into|from filename
[with-clause]

create [unique] index indexname on tablename
    (columnname [, columnname])
[with-clause]

create integrity on tablename [corrname]
    is search_condition

create table tablename
    (columnname format [, columnname format])
[with-clause]

create table tablename
    [(columnname [, columnname])]
    as subselect {union [all] subselect}
[with-clause]

create view viewname
    [(columnname [, columnname])]
    as subselect {union [all] subselect}
[with check option]

declare global temporary table [session.] tablename
    (columnname format [, columnname format])
[with-clause]

drop permit|integrity on tablename all | integer [, integer]

drop procedure procname

drop [table|view|index] objectname [, objectname]

grant all [privileges] on [table] tablename [, tablename]
to public [, username]

grant priv [, priv] on [table] tablename [, tablename]
to public [, username]
```
grant all [privileges] on procedure procname {, procname}
to public | username {, username}
grant priv {, priv} on procedure procname {, procname}
to public | username {, username}

modify tablename|indexname
to storagestructure | verb [unique]
[on columnname [asc|desc] {, columnname [asc|desc]}
[with-clause]

save tablename [until month day year]

savepoint savepointname

set autocommit on|off

set journaling|nojournaling [on tablename]

set result_structure
heap|cheap|heapsort|cheapsort|hash|chash|isam|
cisam|btree|cbtree

set lockmode session|on tablename
where [level = page|table|session|system|row]
[, readlock = nolock|shared|exclusive|session|system]
read committed|repeatable read]
[, maxlocks = n|session|system]
[, timeout = n|session|system]

set session with on_error = rollback statement | transaction

set transaction isolation level
[read uncommitted|read committed]
repeatable read|serializable]

set [no]logging

set [no]optimizeonly

set noprintqry|printqry

set noqep |qep

set joinop [no]timeout
Knowledge Management Extension Syntax

alter group groupid {, groupid}
add users (userid {, userid}) | drop users (userid {, userid}) | drop all
alter role roleid {, roleid}
with nopassword | with password = rolepass

create dbevent eventname

create group groupid {, groupid}
{with users = (userid {, userid})}

create role roleid {, roleid}
with nopassword | with password = rolepass

drop dbevent eventname

drop group groupid {, groupid}

drop permit on dbevent eventname all | integer {, integer}

drop role roleid {, roleid}

drop rule ruleid

grant all [privileges] | privilege {, privilege}
on [objectdescriptor] objectname {, objectname}
to public | [authtype] authid {, authid}

raise dbevent eventname [eventtext] [with [no]share]

register dbevent eventname

remove dbevent eventname

revoke all [privileges] | privilege {, privilege}
on database dbname {, dbname} | on current installation dbname {, dbname}
from public | from [authtype] authid {, authid}

set [no]rules|noprintrules

set nomaxio | maxio -

set nomaxrow | maxrow value

set nomaxquery | maxquery value

set [no]printdbevents

set [no]logdbevents
OpenSQL Syntax Summary

In addition to the four query language statements (insert, delete, select, and update) described in the Language Reference Guide online help, the following is a list of the OpenSQL statements that can be included in OpenROAD scripts:

- close
- commit
- create index
- create table
- create view
- drop
- fetch
- open
- rollback
- set

Distributed Ingres Syntax Summary

For distributed Ingres systems, the following list includes additional OpenSQL statements that can be included in your OpenROAD scripts:

- direct connect
- direct disconnect
- direct execute immediate
- register as link
- register as link with refresh
- remove
Appendix B: Operating System Differences

This section contains the following topics:

- How You Can Use File Specifications (see page 489)
- Operating System Commands (see page 489)

Operating system differences affect your OpenROAD programs when you:

- Specify file names
- Call operating system commands

When you are coding for more than one operating system, you can use the OperatingSystem attribute of SessionObject to check the current operating system for the running application. For more information about this attribute, see the Language Reference Guide online help.

How You Can Use File Specifications

File specifications, which are rarely used in 4GL, are required for the following:

- FileHandle attribute of BitmapObject and StringObject
- SQL copy statement
- call runimage statement

One way to use file specifications is to store them in an Ingres variable. At runtime, use the GetEnv method of the SessionObject system class to determine the file specification. For example:

```
filename = CurSession.GetEnv('MYAPP_FILENAME');
```

Another way to use file specifications is to store your file specifications in a database table.

Operating System Commands

The OpenROAD call system statement lets you execute a single operating system command from your 4GL code. When you use this statement in a portable application, you must account for the differences between commands on the various operating systems.
The following example presents a solution:

```pascal
if CurSession.OperatingSystem = SY_UNIX then
call system 'ls /usr/dracula/files';
elseif CurSession.OperatingSystem = SY_VMS then
call system 'directory
user$disk:[dracula.files]';
elseif CurSession.OperatingSystem = SY_MSDOS then
call system 'command dir \usr\dracula\files';
elseif CurSession.OperatingSystem = SY_NT then
call system 'cmd dir \usr\dracula\files';
else
  CurFrame.InfoPopup(messageText =
    'Error: mysterious operating system');
endif;
```
3GL procedure

A 3GL procedure is a procedure written in a third-generation language, such as C, that you can call by name from a script or 4GL procedure. 3GL procedures are used to perform operations that are outside the scope of OpenROAD but are available from a 3GL.

abstract class

General classes that are used for classification only are known as abstract classes in object-oriented programming.

aggregate function (set function)

An aggregate function returns a single value based on the contents of a column. Aggregate functions are also called set functions.

aggregation

The aggregation inter-class relationship describes an object that is composed of separate, smaller objects that function together.

array

An array is a named set of rows in which each row is a reference variable that points to an object of a given system or user class.

assistant

Assistants are procedures that can be attached to a frame or a field template. They can be designed to prompt the user for varying amounts of information used to customize the frame or field according to the user's needs or to perform a certain function each time the field or frame is invoked.

autocommit on

Autocommit on describes a transaction management state in which each database statement is an individual transaction automatically committed when it is successfully executed.
BitmapObject class

The BitmapObject class provides attributes and methods that enable you to manipulate the images displayed in an image field.

casting

Casting involves making the system class of a generic object explicit.

collection

A collection is an object that contains a group of related objects.

command

A command is an operation that you execute from an OpenROAD menu or at the operating system level.

concurrency control

Concurrency control refers to the management of shared data so that simultaneous users can access and update the data with little or no wait time while data integrity is maintained.

cursor

A cursor is a row marker that designates an individual row in the select statement’s result table (the set of rows returned by a non-singleton select).

data entry error handler

A data entry error handler is a global or local procedure written in 4GL that gains control as a result of a user typing invalid data into an entry field. The error condition is detected when the entry field loses input focus, or the GetFieldValue method is executed on the entry field.

database event

Database events are user-defined events. They are created and manipulated using SQL statements and stored in a database. Database events are a way for two programs that are connected to the same database to communicate.

database procedure

A database procedure is a data-oriented procedure stored in the database and executed within the database server that you can call by name in a script or 4GL procedure. Database procedures are often used to increase performance and help ensure data integrity and consistency.
dereferencing colon

A dereferencing colon preceding a dynamic name indicates that you are using the contents of a variable to supply the value for an OpenROAD name.

dynamic array variable

A dynamic array variable is a named set of rows.

dynamic frame

Dynamic frames are frames that perform operations based on decisions made by the user at runtime.

dynamic name

A dynamic name is a name that you can use in a statement either dynamically (when you run the application) or statically (when you create the application).

DynamicFrame frame

The DynamicFrame frame at runtime displays the starting frame that provides the user with a choice of database tables. The starting frame then builds a second frame that presents the columns of the chosen table as simple fields on a form.

effective user

The effective user is the user name under which the application is running.

encapsulation

The process of allowing only the object itself to have access to its attributes and methods is called encapsulation.

event block

An event block is a sequence of statements associated with one or more specific events.

event chain

An event chain is a linked series of events triggered by a starting event.

expression

Expressions are language constructs that resolve to a value, a set of values, or TRUE or FALSE.
external class

An external class defines the class properties, methods, and events for one or more external objects.

field variables

Field and menu item variables are variables that are associated with the value in a field or menu item. These variables are not associated with the object represented by the field or menu item.

frame

A frame is a window that consists of a form, with or without a menu, used to display and input data.

frame event

Frame events include events that are triggered by user interactions with the frame's window, such as the WindowResized event that is triggered when the user resizes the window. Frame events also include interactions with the background of the frame, such as the Details event, which is triggered for the frame when the user clicks the Details button on the area surrounding the fields.

frame mode

A frame mode is a name given to a particular set of field and menu biases.

FrameExec object

FrameExec is the OpenROAD system class that contains information about the running instance of a frame, such as the parent frame, the list of fields that are currently selected, and the starting menu for the frame.

FrameForm object

A FrameForm object is a special case of a subform that encloses the full set of FormField objects in a frame.

FrameSource object

The FrameSource object contains a frame's source definition, its initial settings when it is called or opened.

ghost frame

Ghost frames differ from other OpenROAD frames in that they do not contain forms (making them invisible to the user).
**global 4GL procedure**

A *global 4GL procedure* is a routine written in 4GL that you can call by name from a script or another procedure. The procedure is a component of an application and is stored in the application's database.

**global variable**

*Global variables* provide data that is pertinent to the entire application and are available for use in any script or procedure.

**hexadecimal constant**

*Hexadecimal constants* are a special version of string literals. They represent characters expressed in hexadecimal notation and are most often used for nonprintable characters such as a newline character.

**image field**

An *image field* is a rectangular boundary, like a picture frame, in which you can display bitmap images that are stored either in files or in the database.

**image trim**

Unlike an image field, *image trim* displays a single bitmap image.

**ImageField object**

The *ImageField object* describes the image field that you use to display the bitmap images. The ImageField object specifies how to display the image.

**include script**

*Include scripts* are scripts that contain 4GL code that you can include anywhere in your frame, procedure, or user class script by using the `#include` statement.

**inheritance hierarchy**

With *inheritance hierarchy*, a class that is a child of another class is considered its *subclass* and a class that is the parent of another class is considered its *superclass*. Each subclass inherits the attributes and methods for all its superclasses.

**listview field**

A *list view field* lets the end user view and optionally manipulate or edit a list of items.
local variable

Local variables contain data that is associated with a specific frame, procedure, method, field script, or event block.

macro variable

Macro variables are substitution parameters that you can define and give a value.

menu event

Menu events are events triggered when a user selects a menu item.

menu item variables

Field and menu item variables are variables that are associated with the value in a field or menu item. These variables are not associated with the object represented by the field or menu item.

named constant

A named constant is a literal value to which you give a name. You can then use the name in place of the constant in any 4GL expression.

null

A null is an undefined or unknown value. A null is not the same as a zero, a blank, or an empty string.

null constant

A null constant is represented by the special keyword null. You use this keyword to assign a null to a nullable variable or database column.

object

An object is a compound data structure that holds values that you can manipulate.

PaletteField object

PaletteField objects display a list of values, represented as images, from which the user selects a single value.

polymorphism

The process of a subclass's method superseding the behavior of an identically-named method defined for a superclass is called polymorphism.
**procedure**

A *procedure* is a named piece of code that performs a single task.

**procedure handle (ProcHandle object)**

A *procedure handle* (ProcHandle object) represents a global or local procedure and the scope in which to execute it.

**procedure statement**

The *procedure statement* defines a 4GL procedure. This statement declares the parameters for the procedure and provides the actual processing statements.

**reference variable**

A *reference variable* is a variable that points to an object of a given class, letting you access the value of an object.

**script**

A *script* is a set of statements that define the actions performed by a procedure or frame.

**select loop**

A *select loop* is a block of 4GL statements that are performed on each row returned by the select statement. The loop ends when there are no more rows or an endloop statement is encountered.

**simple variable**

A *simple variable* is a single data item. It contains only one value.

**source breakpoint**

A *source breakpoint* is a defined place in a script where the application passes control to the Debugger. Source breakpoints can pass control to the Debugger only between statements.

**statement**

A *statement* is an operation that you place within a program or called procedure.
**StringObject class**

The *StringObject class* is an in-memory text string of any length. It provides methods that enable you to manipulate a text by extracting, truncating, and concatenating strings. It also provides the attributes and methods you use for storing and updating the strings, either in standard text files or in the database.

**system variable**

*System variables* are built-in variables that are available in all frames and scripts.

**table field**

*A table field* is a composite field comprised of active fields arranged in rows and columns. Each column in the table field maps to one attribute of the class associated with the array and each table field row displays one row of the array. The fields are usually entry fields, but can be of any kind, such as image or toggle fields.

**transaction**

*A transaction* is one or more database statements processed as a single, indivisible unit.

**tree**

A *tree* contains a collection of treenode objects. The top treenode in the hierarchy is the root node.

**Tree class**

The *Tree class* is used to manage data that has an hierarchical structure.

**TreeNode class**

The *TreeNode class* represents a node that contains data for a single node in a tree.

**treeview field**

*A tree view field* presents the end user with a hierarchical list of items in a tree structure that can be expanded or collapsed.
**user class script**

*A user class script* contains methods that you can invoke to use with variables defined for a user class.

**user event**

*A user event* is an event that you trigger with 4GL code (rather than an event triggered directly by a user action).
Index

' (single quotation marks)
  constants • 37

- (minus sign)
  subtraction • 41

! (exclamation point)
  using in names • 21

#
#define statement • 437
#if-#endif statement • 438
#ifdef-#endif statement • 439
#endif-#endif statement • 439
#include statement • 434
#undef statement • 438
%
% (percent sign)
  in pattern matching • 44

* (asterisk)
  count function • 62
  exponentiation • 41
  multiplication • 41

/ (slash)
  division • 41
  in comments • 65

[ ] (square brackets)
  escape clause • 44

_ (underscore)
  in pattern matching • 44
_date function • 57
_RowState attribute
  changing • 212
  valid settings • 212
_StringParseKeyword procedure
  core library component • 90
_StringSub procedure
  core library component • 90
_StringTokenSub procedure
  core library component • 91
_time function • 57
+
+ (plus sign)
  concatenation operator • 42

3
3GL load module
  creating • 306
3GL procedures • 74, 80, 295
  4GL statement table • 299
  C • 82
  creating load module • 306
  errors • 151
  errors in embedded 4GL statements • 299
  example • 301
  linking • 306
  parameters • 80
  passing arrays • 298, 300
  passing parameters • 295, 296
  sending user events • 303
  switching sessions • 197

4
4GL objects
  passing • 98, 101, 105
4GL procedures
  calling • 84
  ghost frames • 111
Abs function • 53
Absolute value • 53
Active frames
  blocking • 106
  popup • 108
ActiveRow attribute
  scrolling • 239
  setting • 240
ActiveX
  and external classes • 135
Adding
  columns to table field at runtime • 377
  fields to composite at runtime • 377
  frames to application at runtime • 396, 399, 412
  images to database • 283
  rows to array • 216
AllBias attribute • 372
AllRows attribute • 221
Alphanumeric identifiers • 19
Applications
  adding frames at runtime • 416
  attaching frames at runtime • 396
  communicating using database events • 357
  controlling execution • 470
  debugging • 465
  debugging imaged • 482
  disconnecting database sessions • 197
  event queue • 318
  exiting • 99
  monitoring • 459
  naming • 19
  non-interactive • 111
  object oriented approach • 120
  referencing • 21
  running • 443
  running multiple • 443
  sending an event to • 351
  switching database sessions • 193
  testing • 443
ApplyTemplate utility • 419
AppSource system class • 144
Arithmetic expressions • 41
Array Display Settings dialog • 477
Array variables
  declaring • 209
  dynamic • 31
  passing • 101
ArrayObject system class
  _RowState attribute • 212, 219
  AllRows attribute • 221
  Clear method • 221
  declaring • 209
  Duplicate method • 216
  Find method • 224
  FirstRow attribute • 221
  LastRow attribute • 221
  RemoveRow methods • 221
  SetRowDeleted method • 221
  Sort method • 223
Arrays
  accessing in 3GL procedures • 299
  and table fields • 228
  assigning values • 216
  changing values • 219
  declaring • 32
  declaring data at runtime • 380
  declaring reference variable • 209
  defined • 31
  deleting • 219
  described • 205
  display settings • 477
  duplicating • 216
  expanding objects • 473
  finding values • 224
  inserting rows • 216
  marking rows deleted • 219
  multiple references • 216
  nested • 207
  passing to 3GL procedures • 298, 300
  referencing elements • 210
  retrieving status information • 221
  searching • 224
  single column • 212
  sorting • 223
  status information • 223
  table fields • 205
  updating values • 219
Assign dialog • 477
Assign Keys dialog • 480
Atan function • 53
AttributeObject system class • 143
| Attributes | defined • 28  
|           | referencing • 114 |
| Autocommit | cursors • 187  
|           | defined • 187 |
| Avg function | • 60, 61 |
| B | |
| Batch Applications | • 111 |
| Bias attribute | CurBias • 368  
|           | using • 366  
|           | valid value summary table • 366 |
| Biases | CurBias attribute • 368  
|           | described • 365  
|           | field biases summary table • 366  
|           | interactive category • 366  
|           | menu item biases table • 368  
|           | mode/bias attributes • 372  
|           | passive category • 366  
|           | setting • 368, 370 |
| Bitmap objects | ii_stored_bitmaps database table • 285  
|           | storing in database tables • 285 |
| BitmapObject system class | • 285  
| DBHandle attribute | • 284  
|           | described • 282  
| InsertIntoDB method | • 284  
| tables for storing | • 285  
| using with database | • 284 |
| Blanks | • 54  
|           | trailing • 54 |
| BlocksFrames attribute | • 106 |
| Boolean expressions | • 43  
| break conditions | • 446, 448  
| calls | • 454  
| errors | • 452  
| events | • 450  
| methods | • 455  
| source | • 448  
| threads | • 455 |
| Break on Event dialog | • 451  
| breakpoinst | setting • 445  
|           | source breakpoints • 448  
|           | types • 445 |
| Built-in macro variables | • 436  
| Byref keyword | • 101  
|           | passing parameters • 103, 105 |
| C | |
| C function | • 51 |
| C language | coding procedures • 82  
| call break conditions | • 454  
| setting | • 454 |
| Call method | • 88  
|           | ProcHandle system class • 88 |
| Call Stack window | • 460 |
| Call system statement | • 489 |
| Callframe statement | BlocksFrames attributes • 106  
|           | byref • 101  
|           | described • 100  
|           | passing parameters • 101  
|           | popup frames • 108  
|           | returning values • 101  
|           | with clause • 106 |
| Calling frames | popup frames • 108 |
| Calling procedures | 4GL • 84  
|           | database • 85  
|           | described • 83  
|           | dynamically • 84 |
| Callproc statement | • 74, 295  
|           | calling procedures • 84  
|           | returning values • 84 |
| Calls Break Conditions tab page | • 454 |
| Case | lowercase function • 54  
|           | uppercase function • 54 |
| Casting | • 330  
|           | described • 139  
|           | MessageObject attribute • 338 |
| CellAttribute system class | creating a table field • 228  
|           | HasDataChanged attribute • 247  
|           | individual TableField cells • 233  
|           | performance • 228  
|           | turning on • 246 |
| Changing | array declarations • 381  
|           | array values • 219  
|           | bias • 368 |
field appearance at runtime • 365
field capabilities at runtime • 365

Changing data
  BitmapObject method • 163
  using a QueryObject • 181
  using cursors • 161
  using QueryObject methods • 163
  using SQL • 153
  using StringObject method • 163

Changing frames at runtime
  described • 363, 364
  using biases • 365

Char function • 51
Character data • 54

Child frames
  accessing parent frame • 95
  assigning parents • 95
  closing • 103
  communicating with parent frame • 95, 104

ChoiceField system class
  cleaning ValueList • 399
  displaying new values • 401
  loading ValueList • 401

ChoiceList system class • 401

Class system class
  Create method • 392
  described • 143

Classes
  inheritance • 121
  naming • 19
  subclass • 121
  superclass • 121

ClassSource system class • 143, 144

Clear method • 221
ClipboardSee PC Clipboard • 244
Close statement • 162

Closing
  cursors • 162

Collections • 254
  and external classes • 136
  indexing • 254

ColumnField system class
  creating a table field • 228
  described • 228
  HasCellAttributes attribute • 246
  ProtoField attribute • 232
  Title attribute • 231
  TitleTrim attribute • 231

Columns attribute of DataStream
  query objects • 178
  SetCols method • 176
  Targets attribute • 167, 178
  targets required for query objects • 178

Columns in table fields
  adding at runtime • 376
  adding fields to a composite field • 376
  deleting at runtime • 379
  hidden • 245
  invisible • 245
  multiline headers • 245
  turning headers off/on • 245

Columns in tables
  naming • 19

Comments • 65

Communicating between frames
  database events • 361
  described • 333, 336
  FrameExec objects • 104
  global variables • 335
  terminate method • 334

Comparison operators • 43

Compiler
  limitations • 92

Composite fields
  creating at runtime • 376
  CurBias attribute • 369

CompSource system class • 144

Computation
  logarithms • 53

Concat function • 54

Concatenation • 42

ConcatVarchar method • 395

Concurrency
  controlling • 182
  openframe statement • 103
  transaction management • 182

concurrent threads
  synchronizing • 445

ConfirmPopup method • 109

Connect to Application DB Session dialog • 465

Connect to DB dialog • 464

Constants
  accessing from 3GL procedures • 299

Control button
  associated with table fields • 234, 241
  changing • 242
  multiple table fields • 252
  turning off and on • 241
using to copy data • 244
using with dynamic table fields • 389
Copy data to the PC Clipboard • 244
Core library components
  _StringParseKeyword • 90
  _StringSub • 90
  _StringTokenSub • 91
Cos function • 53
Count (aggregate function) • 62
Create dbevent statement • 359
Create method • 392
creating objects • 31
Creating
  composite fields at runtime • 375
database procedures • 155
dynamic expressions • 385
fields at runtime • 373
frames at runtime • 392
query objects • 178
CurBias attribute
  composite fields • 369
  settings for fields • 368
  settings for menu items • 368
CurEventScope • 39
CurEventScope system variable • 39
CurFrame • 39
CurFrame system variable • 39
  FrameExec object • 95
CurMethod method • 129
CurMode attribute • 371
CurObject system variable • 129
CurOps attribute • 243
CurProcedure system variable • 39
Current Break Conditions for Application window • 446
CurSession system variable • 39, 489
CursorObject system class
  close statement • 162
  DBSession attribute • 159, 191
declaring cursor variables • 160
delete statement • 161
deleting rows • 161
described • 157
fetch statement • 160
fetching data • 160
multiple sessions • 191
opening • 160
retrieving rows • 160
transactions • 163
update statement • 161
updating rows • 161
Customize Function Keys dialog • 480

D
Data entry error handlers
defined • 309
parameters • 309
Data integrity • 85
Data type
result • 64
Data types • 35
conversion functions list • 51
Database
  communicating with • 357
DeleteFromDB method • 186
deleting rows • 154
disconnecting sessions • 197
inserting rows • 152
InsertIntoDB method • 186
opening new session • 464
query statements • 147, 148
retrieving rows • 149
specifying session • 464
switching sessions • 193
transferring data to variables • 150
UpdateInDB method • 186
updating • 153
Database events
  communicating with • 357
create dbevent statement • 359
creating • 359
DBEventObject system class • 358
described • 357
duration of registrations • 359
frame communication • 361
multiple session • 188
multiple sessions • 358
obtaining event information • 358
permissions required to raise event • 360
application threads • 444
derector threads • 444
display settings • 477
descriptions • 477
arrays • 473
assigning values to expressions • 477
breaking on events • 451
breaking on thread start • 455
call break conditions • 454
CallStack window • 460
choosing different script • 469
Current Break Conditions for Application window • 446
DB Session for Application window • 463, 482
Debugger window • 449
error break conditions • 452
error breaks • 444
event break conditions • 450, 451
examining expressions • 473
frame states • 444
included applications • 471
Log for Application window • 458
Log window • 456, 482
logging trace information • 456, 458
Menu • 442
method invocation break conditions • 455
Monitor window • 459, 460, 462, 482
navigation buttons • 470
running applications • 444
script display icons • 468
synchronizing concurrent threads • 445
thread identifiers • 444
thread start breakpoints • 455
transactions • 444
window • 462
Debugging • 465
accessing system editors • 481
synchronizing concurrent threads • 445
DeclareData method • 380
Declaring
local variables • 116
Delete statement
examples • 154
DeleteFromDB method • 285
determining error status • 200
managing transactions • 186
Deleting
array rows • 219
choice field values • 399
columns from table field • 379
data using cursors • 161
data using QueryObject methods • 181
data using SQL • 154
database events from queue • 360
external events from queue • 349
fields from form • 378
user events from queue • 347
Deleting data
BitmapObject method • 163
StringObject method • 163
using QueryObject methods • 163
demo, external user event • 353
dialogs
Array Display Settings • 477
Assign • 477
Assign Keys • 480
Break on Event • 451
Connect to Application DB Session • 465
Connect to DB • 464
Customize Function Keys • 480
Event Break Conditions • 451
New DB Event • 462
New User Event • 462
Number of Levels • 475
Watch Values • 478
Disabling
user modifications to table field data • 241
Disconnect method • 197
DisplayPolicy attribute
valid settings • 286
Dow function • 51
Dynamic array variables • 31
Dynamic expressions
creating • 385
described • 385
using • 387
Dynamic names • 20
SQL statements • 483
Dynamic-link library • 74
E
editors
Script Editor • 481
system • 481
Embedded SQL procedures • 298
4GL arrays • 300, 301
example • 300
ghost frames • 303
handling errors in inquire_4GL statement • 299
sending user events • 303
statement summary • 299
Enabling
user modifications to table field data • 241
Encapsulation • 128
Entry fields
creating at runtime • 373
time break conditions • 452
Debugger • 452
setting • 453
Error handlersSee Data entry error handlers • 309
Error messages
customizing • 312
ErrorNumber attribute • 197
Errors
  3GL • 151
BitMapObjects • 204
called 3GL procedures • 151
DataStreams • 199
DeleteFromDB method • 200
ErrorStatus attribute • 201
handing database errors • 197
inquire_sql statement • 201
InsertIntoDB method • 200
retrieving error text • 201
StringObjects • 204
types • 452
UpdateInDB method • 200
Errors Break Conditions tab page • 453
Errors, database queries
  3GL procedures • 151
BitMapObject methods • 200, 204
DataStream methods • 199
DBMSError attribute • 197
error handler procedure • 202
error handler procedure example • 202
ErrorNumber attribute • 197
inquire_sql statement • 201
QueryObject methods • 199
retrieving error text • 201
StringObject methods • 200, 204
ErrorStatus attribute
  setting • 201
Event block
  BlocksFrames attribute • 106
interrupting • 117, 328
  processing • 336
  switching sessions • 195
Event blocks
  field and menu item scripts • 71
  using in frame scripts • 69
time break conditions • 450
  setting • 451
Event Break Conditions dialog • 451
Event queue
  application • 318
BlocksFrames attribute • 106
examining • 321
frame • 319
manipulating • 321
PurgeUserEvent method • 347
SendUserEvent method • 339
user events • 336
Event system class • 146, 338, 345
Events
BlocksFrames attribute • 106
chains • 323
child • 323
database events and chains • 323
database events and multiple sessions • 358
field events • 323
frame • 321
menu • 326
obtaining information about • 330
processing • 319
resume statement and chains • 328
user events and chains • 323
viewing event queue • 462
Events Break Conditions tab page • 451
Exec 4GL sample • 308
Exec SQL sample • 308
Execute Immediate statement • 155
Exit event
  SendUserEvent method • 341
Exit statement
  leaving applications • 99
transactions • 188
Exp function • 53
Exponential function • 53
Expressions • 36
  and non-printable characters • 37
  and procedures • 86
arithmetic • 41
assigning values • 477
boolean • 43
examining values • 473
hexadecimal characters • 37
like operator • 44
logical operators in • 42
nulls • 49
procedures • 47
string • 41
variables in • 47

External class libraries
 naming • 19
External classes • 135
and collections • 136
External communications
database events • 357
External events • 351
 communicating with • 347
demo • 353
event block • 350
focus behavior • 351
MessageInteger attribute • 351
passing data • 351
registering • 348
removing • 349
sending to applications • 351
unregistering • 349
using to communicate • 347
External programs
 communicating • 347
External user event demo • 353
 UNIX or Linux • 355
 Windows • 353
Extevent.c program • 351
ExtObjField class • 136

F

Fetch statement
 CursorObject • 160
Field function • 33, 65
described • 117
reference variable • 117
referencing field object • 364
TableField object • 229
Field templatesSee Templates • 419
FieldObject system class
 ChildFields attributes • 378
Fields
 assigning values to dynamic fields • 382
attaching to form • 394
biases • 366
creating at runtime • 373, 394, 405
creating example • 405
finding field in a table field • 249
getting values from dynamic fields • 382
listview • 257
mode/bias attributes • 372
removing from a form • 378
restoring to form • 379
scripts • 71
setting bias • 370
setting focus behavior • 341
treeview • 257
File specifications • 489
FilePopup method • 109
Find method
described • 224
parameters • 224
FirstRow attribute • 221
Float4 function • 51
Float8 function • 51
Flush method • 84, 101
Focus behavior
external events • 351
IIW4GL_SendUserEvent _ procedure • 351
user events • 341
valid values • 341
Focusbehavior
 parameter • 341
Form fields
naming • 19
Forms
creating in application • 394
Frame definition
 changing • 106
Frame script
creating dynamically • 395
Frame scripts
event blocks • 69
writing • 67
Frame templatesSee Templates • 419
FrameExec objects
 accessing • 95, 106
changing attribute values • 106
described • 95
openframe statement • 95, 104
passing • 95, 98
referencing its variable • 339
setting attributes • 106
FrameExec system class
    BlocksFrames attribute • 106
    changing attributes • 106
    CurBias attribute • 368
    CurMode attribute • 371
    described • 95, 144
    message attributes • 338
    MessageInteger attribute • 351
    mode/bias attributes • 370
    openframe statement • 103
    PurgeDBEvent method • 360
    PurgeUserEvent method • 347
    RegisterUserEvent method • 349
    SendUserEvent method • 339
    Terminate method • 334
    UnRegisterUserEvent method • 349
    user events • 337
    WaitFor method • 345

FrameForm system class • 393

Frames
    attaching to application • 396
    blocking • 106
    BlocksFrames attribute • 106
    calling • 100
    changing definitions • 106
    changing mode • 370
    changing properties at startup • 106
    child • 95, 103
    closing • 99, 333
    communicating with database events • 361
    creating dynamically in an application • 392
    creating the FrameSource object • 403
    CurMode attribute • 371
    current • 339
    current mode • 370, 372
    default mode • 370
    dynamic • 23
    form • 393
    FrameExec objects • 95
    FrameForm object • 393
    FrameSource object • 393
    iconifying • 108
    invoking • 95
    menu creation • 394
    modes • 370
    modes summary table • 370
    multiple sessions • 190
    multi-purpose • 363
    naming • 19
    opening new connections • 193
    parent • 95, 103
    passing • 104
    passing control • 104
    passing parameters • 95, 98, 103
    pop-up • 108
    referencing current • 95
    registering external events • 348
    returning from • 101
    running instance • 339
    source definition • 393
    switching sessions • 194
    synchronizing • 103
    terminating • 334
    transferring control • 98, 100
    variables • 25

FrameSource system class
    creating FrameSource object • 403
    creating in application • 393
    function keys
        customizing • 480
        Debugger assignments • 480

Functions
    abs • 53
    atan • 53
    avg • 60, 61
    c • 51
    char • 51
    charextract • 54
    concat • 54
    cos • 53
    data type • 51
    data type conversion • 51
    date • 51, 57
    dbmsinfo • 64
    field function • 65
    float4 • 51
    float8 • 51
    hex • 51
    ifnull • 63
    int1 • 51
    int2 • 51
    int4 • 51
    left • 54
    length • 54
    locate • 54
    log • 53
    lowercase • 54
    max • 60, 61
    min • 60, 61
    mod • 53
money • 51
numeric list • 53
pad • 54
right • 54
scalar • 51
shift • 54
sin • 53
sqrt • 53
string • 54
sum • 61
system-defined • 50
text • 51
trim • 54
uppercase • 54
varchar • 51

G

Generic errors • 197
GetProcHandle method • 87
Ghost frames
described • 111
Global constants
naming • 19
Global procedures
creating • 74
ProcHandle objects • 87
Global variables • 25
accessing from 3GL products • 299
described • 335
Gotoframe statement
BlocksFrames attribute • 106
popup frames • 108
returning values • 104, 105
with clause • 106
Group by clause • 62

H

Handles • 87
HasCellAttributes attribute • 228, 233, 246, 247
HasDataChanged attribute • 247
HasHeader attribute • 245
HasScrollBar attribute
for TableField • 237
Hex function • 51
Hexadecimal characters
expressions • 37
Iconifying frames • 108
icons
Debugger • 468
Ifnull function • 63
ii_stored_bitmaps database table • 285
ii_stored_strings database table • 288
IIDBMSerror system variable • 39
IIerrornumber system variable • 39
IIW4GL_SendUserEvent procedure • 351
imaged applications
debugging • 482
ImageField system class • 286
DisplayPolicy attributes • 286
Images
deleting from database • 285
display policy property • 286
palette fields • 280
retrieving from database • 285
storing in database • 284
storing in files • 282, 283
storing in the database • 282
table for storing • 286
valid formats • 283
working with • 277
ImageTrim class • 279
Include scripts • 434
Included applications • 87
InfoPopup method • 109
Inheritance • 121
Initialize (4GL statement)
using in script • 68
InputFocusField attribute • 140, 330
Inquire_4gl statement • 299
Inquire_sql statement • 201
described • 201
transactions • 163
Insert statement
described • 152
examples • 152
Inserting data
BitmapObject method • 163
StringObject method • 163
using QueryObject methods • 163
using SQL • 152
InsertIntoDB method • 289
determining error status • 200
for BitmapObjects • 284
for QueryObjects • 200
for StringObjects • 289
storing images • 289
Int1 function • 51
Int2 function • 51
Int4 function • 51
Interval function • 57
Invoking frames
described • 95

L

LastRow attribute • 221
Left function • 54
Length function • 54
Like operator • 44
escape clause • 44
Listview fields • 257
ListviewField class • 257
Literals
  hexadecimal constants • 37
  null constant • 37
  numeric • 37
  string • 37
Load modules
  and 3GL procedures • 74
Local procedures • 74
  ProcHandle objects • 87
Locate function • 54
Log for Application window • 458, 482
Log function • 53
Log window • 456, 482
Logarithmic function • 53
logging trace information • 456
Logical operators • 42
Lowercase function • 54

M

Macro variables
  #undef statement • 438
  built-in • 436
  creating new • 437
  user-defined • 435
Matrix fields
  adding fields at runtime • 375
  creating at runtime • 375
MatrixField system class
  CollapsePolicy attribute • 375
  InsertChild method • 375
Max function • 60, 61
Menu item
  scripts • 71
Menus
  AllBias attribute • 372
  attaching to frame • 411
  biases • 368
  creating • 411
  creating dynamically • 394
  creating in application • 394
Message statement
  BlocksFrames attribute • 106
described • 111
MessageFloat attribute • 338
MessageInteger attribute • 338
MessageObject attribute • 338
MessageVarchar attribute • 338
method invocation break conditions • 455
  setting • 455
MethodExec system class
described • 144
MethodObject system class • 143
Methods
  invoking • 119, 127
  user class scripts • 73
Methods Break Conditions tab page • 455
Min function • 60, 61
Mod function • 53
Modes
  changing • 370
  current • 370
  default frame mode • 370
  defined • 370
  mode/bias attributes • 372
  summary table • 370
Modulo arithmetic • 53
Money function • 51
Money literals • 37
MonImage utility • 482
Monitor window • 459, 460, 462, 482
MonitorImage window • 482
monitoring applications • 459
Multiple sessions
  database events • 358

N

Name resolution • 21
Named constants
defined • 38
Names
  application • 21
  dynamic • 20
  resolution • 21
  supplying at runtime • 20
New DB Event dialog • 462
New User Event dialog • 462
Non-interactive applications • 111
Non-printable characters
  expressions • 37
Null constant • 38
Nullability
  ifnull function • 63
Nulls
  aggregate functions • 61
  in expressions • 49
  reference variables • 30
Number of Levels dialog • 475
Numeric data type
  functions list • 53

O

Object system class • 142
Objects
  and attributes • 28
  and classes • 28
  casting • 139
  creating • 31, 116
  creating in an application • 392
  defined • 28
  manipulating generically • 139
  naming • 19
  passing • 98, 101, 105
  passing objects to/from 4GL • 298
  passing to 3GL • 298
  passing to frames • 101, 105
  referencing • 29
ObjectSource attribute • 396
ObjectSource system class
  creating in application • 396
Open method • 170
Open statement
  using • 160
Openframe statement
  concurrent frames • 103
  FrameExec object • 95, 98, 104
  passing parameters • 103
  with clause • 106

OpenSQL
  statement syntax summaries • 483
Operating system
  call system statement • 489
  file specifications • 489
  porting application • 489
OperatingSystem attribute • 489
Operators
  arithmetic • 41
  string • 42
OriginatorField attribute • 140, 330

P

Pad function • 54
Palette fields
  arranging display • 280
  described • 280
  loading images • 280
Parameters
  default null • 116
  naming • 19
  objects and arrays • 101
  openframe statement • 103
  passing 4GL objects and arrays • 101
  passing between frames • 95, 98
  passing by reference • 101, 103, 105
  passing by values • 101
  passing to 3GL procedures • 80, 296
  passing to 4GL procedures • 76, 84
  passing to database procedures • 85
  passing to popup frames • 109
  passing to procedures • 84
  passing with openframe statement • 103
Parent frames
  assigning child • 95
  closing • 103
  parent/child relationships
    displaying • 459
  ParentApplication attribute • 396
  ParentField attribute • 394
  ParentFrame attribute • 95
  ParentMenuGroup attribute • 394
Passing
  objects • 98, 101
Pattern matching • 44
PC Clipboard
  copying data from a table field • 244
Performance
and database procedures • 85
cell attributes • 228
default null • 116
null defaults • 116
Polymorphism
defined • 125
using • 130
Popup frames
predefined • 108
Pop-up frames
creating • 108
Preprocessor
#define statement • 437
#if-#endif statement • 438
#ifdef-#endif statement • 439
#ifndef-#endif statement • 439
#include statement • 434
#undef statement • 438
errors • 439
macro variables • 435
user-defined macro variables • 435
PreviousField attribute • 330
Proc4GLSource system class • 144
Procedure handles
described • 87
Procedure statement • 76
Procedures
3GL • 80
4GL • 420
as expression • 47
BlocksFrames attribute • 106
C • 82
calling • 74
calling in expressions • 86
creating • 74
database • 79
global 4GL • 76
including comments • 65
name resolution • 23
naming • 19
passing parameters to • 84
returning from • 77
ProcExec system class
DBSession attribute • 189
described • 144
popup frames • 109
ProcHandle system class
and error handlers • 88
described • 87
Prompt statement
BlocksFrames attribute • 106
described • 111
ProtoField attribute • 232
ProtoField system class • 228
PurgeDBEvent method • 360
PurgeUserEvent method • 347

Q
Queries
parameterized • 175
Query mode • 370
QueryBias attribute • 372
QueryObject object
described • 177
overview • 177
Update method • 181
QueryObject system class
Close method • 178
columns attribute • 178
CommitToCache method • 178
creating • 178
DBDelete methods • 178
DBInsert method • 178
DBUpdate method • 178
errors from methods • 199
example • 178
FetchRow method • 178
grouping • 178
having clause • 178
Load method • 178
NextRow method • 178
Open method • 178
PrevRow method • 178
required attributes • 178
SetCols method • 177
where clause • 178
QueryOps attribute
settings • 243

R
Radio fields
arranging display • 280
Raise dbevent statement • 360
Read mode • 370
ReadBias attribute • 372
ReadOps attribute
settings • 243
Reference variables • 28
creating objects • 31
declaring • 28, 114
defined • 28
getting attributes • 117
null defaults • 116
nulls • 30
passing • 101
referencing • 28
returning values • 86
setting attributes • 117
Referencing
current frames • 95
Referencing objects • 98
Refreshing display • 84, 101, 103
Register dbevent statement • 359
RegisterUserEvent method • 349
Remove dbevent statement • 359
RemoveRow method • 221
Removing
columns from a table field • 379
fields from a form • 378
ReplyPopup method • 109
Result data type • 64
Resume statement
interrupting an events block • 328
PurgeUserEvent method • 347
Terminate method • 334
Retrieving data
using cursors • 160
using QueryObjects • 177
using SQL • 151
using SQLSelect object • 166
Return statement
described • 101
procedures • 77
Returning values
and database procedures • 86
callproc statement • 84
gotoframe statement • 105
popup frames • 109
Right function • 54
RowCount attribute
described • 159
Rows (in tables)
counting • 62
Rundbapp • 111
Runimage • 111
S
sample, external user event • 353
Scalar functions • 51
Scope
ProcHandle objects • 87
returning values • 101
Script Editor
accessing from Debugger • 481
Scripts
creating dynamically • 395
creating in application • 395
displaying in Debugger • 469
editing in Debugger • 481
for field or menu items • 71
frame • 67
frame script syntax • 67
including comments • 65
user class • 73
user class method examples • 131
using statements • 74
writing • 67
Scroll bar
for table field • 235
turning off/on • 237
using arrowkeys • 237
Scrolling
events and • 239
to a specific row • 239
using ActiveRow attribute • 239
using scrollbar • 237
using SetInputFocus method • 239
Select statement
described • 149
execute immediate statements • 156
select loop • 149
singleton form • 149
variable table names • 151
SendSuperclass method • 130
SendUserEvent method
Delay parameter • 342
described • 339
example • 345
Focusbehavior parameter • 341
message parameters • 341
parameters • 339
resume statement • 341
sending from 3GL to 4GL • 303
<table>
<thead>
<tr>
<th>concat function</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>functions</td>
<td>54</td>
</tr>
<tr>
<td>left function</td>
<td>54</td>
</tr>
<tr>
<td>length function</td>
<td>54</td>
</tr>
<tr>
<td>locate function</td>
<td>54</td>
</tr>
<tr>
<td>lowercase function</td>
<td>54</td>
</tr>
<tr>
<td>padding</td>
<td>54</td>
</tr>
<tr>
<td>right function</td>
<td>54</td>
</tr>
<tr>
<td>shift function</td>
<td>54</td>
</tr>
<tr>
<td>size function</td>
<td>54</td>
</tr>
<tr>
<td>squeeze function</td>
<td>54</td>
</tr>
<tr>
<td>text function</td>
<td>54</td>
</tr>
<tr>
<td>trim function</td>
<td>54</td>
</tr>
<tr>
<td>uppercase function</td>
<td>54</td>
</tr>
<tr>
<td>varchar function</td>
<td>54</td>
</tr>
<tr>
<td>Sum function</td>
<td>60, 61</td>
</tr>
</tbody>
</table>

**System classes**
- described • 113, 142
- fields and menu items • 145
- inheritance • 121
- subclass • 121
- superclass • 121

**System editors**
- accessing from Debugger • 481

**System variable**
- CurEventScope • 39
- CurFrame • 39
- CurProcedure • 39
- CurSession • 39
- IIDBMSerror • 39
- IIerrornumber • 39
- IIrowcount • 39

**System variables** • 39

**System-defined functions** • 50

<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
</table>

**tab pages**
- Calls Break Conditions • 454
- Errors Break Conditions • 453
- Events Break Conditions • 451
- Methods Break Conditions • 455
- Source • 450
- Threads Break Conditions • 455

**Table fields**
- accessing cells in dynamic table field • 385
- adding data to dynamic table fields • 384
- allowing users to append rows • 243
- arrays • 205, 228
- changing at runtime • 376
- changing column headers • 377
- changing control button • 242
- columns • 232
- components • 229
- control button • 241
- creating at runtime • 376
- CurBias • 369
- declaring data at runtime • 380
- declaring data dynamically • 380
- described • 205, 227
- determining row number • 248
- effect of setting CurBias • 369
- field function • 229
- finding rows • 240
- hidden columns • 245
- individual cell attributes • 248
- individual cells • 232
- invisible columns • 245
- matching array rows to table field • 248
- multiline column • 245
- operations • 236
- prototype fields • 232
- referencing • 229
- scroll bar • 235
- single character find • 238
- turning CellAttributes on • 246
- turning column headers off/on • 245
- turning control button off/on • 241
- turning scroll bars on/off • 237
- using control button to copy data • 244

**TableBody attribute • 234**

**TableField system class**
- ActiveRow attribute • 239
- allowing operations • 243
- attributes • 229
- CurOps attribute • 243
- DeleteColumn method • 248, 379
- HasHeader attribute • 245
- HasScrollBar attribute • 237
- InsertColumn method • 248, 377
- SetInputFocus method • 239
- TableBody attribute • 234, 377
- TableHeader attribute • 231
- Title attribute • 230
- TitleTrim attribute • 230
- WhichRow method • 248
- WhichTableField method • 249
- with CellAttribute system class • 233
- with ColumnField system class • 232
- with ProtoField system class • 232

**Index** 517
TableHeader attribute • 231
Tables
  naming • 19
TargetField attribute • 140, 330
Templates
  ApplyTemplate utility • 419
  assistant procedures • 420
  assistants • 419
  creating fields • 428
  customizing • 423
  field • 419
  field creation process • 428
  frame • 419
  frame design considerations • 426
  naming • 19
Terminate method • 103, 334
testing
  applications • 443
Text function • 51
thread map • 459
  current thread • 459
  nodes • 459
thread start breakpoints
  setting • 455
Threads
  and ProcHandle objects • 89
  concurrent • 445
  identifiers • 444
  synchronization • 445
Threads Break Conditions tab page • 455
Time
  functions • 57
  interval function • 57
Title attribute
  for ColumnField • 231
  for TableField • 230
TitleTrim attribute
  for ColumnField • 231
  for TableField • 230
tooltips
  customizing • 481
  speed keys in • 481
trace information
  logging • 458
Transaction management
  concurrency • 182
Transactions
  autocommit • 187, 188
  BitmapObject • 186
  cursors • 163
database sessions • 185
DataStream • 186
determining transaction state • 201
errors • 187
Exit statement • 188
managing • 182
multiple sessions • 185, 190
StringObjects • 186
Tree class • 267
TreeNode class • 266
Treeview fields • 257
  bitmap labels • 274
TreeviewField class • 266
TriggerField attribute • 330
Trim function • 54
Truncation
dates • 57
U
UndeclareData method • 380
UnRegisterUserEvent method • 349
Update mode • 370
Update statement
  described • 153
  examples • 153
UpdateBias attribute • 372
UpdateInDB method
  determining error status • 200
  managing transactions • 186
UpdateIntoDB method • 200
UpdateOps attribute
  settings • 243
Uppercase function • 54
User class scripts • 73
User classes
  creating • 127
  creating methods • 129
  database tables • 123
  encapsulating • 128
  hierarchy • 126
  method examples • 131
  method supercedence • 125
  methods • 127
  object oriented approach • 120
  polymorphism • 125
  source objects • 143
User events • 337
  definition • 334
  event block • 337
event chains • 323
 event queue • 336
 examples • 339
 ghost frames • 111
 naming • 19
 parameters • 338
 purging • 347
 registering • 348
 sending from 3GL to 4GL • 303
 UserEvent event • 349
 waiting for • 345
 User1 mode • 370
 User1Bias attribute • 372
 User1Ops attribute • 243
 User2 mode • 370
 User2Bias attribute • 372
 User2Ops attribute • 243
 User3 mode • 370
 User3Bias attribute • 372
 User3Ops attribute • 243
 UserClassObject system class • 143
 User-defined macro variables • 435
 UserEvent event • 337
 described • 337
 external events • 350
 UserObject system class • 143
 Utilities
 ApplyTemplate • 419
 MonImage • 482

V

Values
 assigning with dynamic expressions • 387
 getting from unknown fields • 388
 getting with dynamic expressions • 388
 passing to procedures • 84
 returning • 101, 104
 returning from database procedures • 86
 Varchar function • 51
 Variables • 25
 calling procedures • 84
curframe • 95
data types • 35
declaring • 116
default values • 114
 dereferencing colon • 20
display type • 117
global • 25, 335
 in expressions • 47

W

WaitFor method
 BlocksFrames attribute • 106
 described • 345
 Event object • 345
 SendUserEvent method • 341, 345
 watch
 values • 478
 Watch Values dialog • 478
 Watch window • 479
 watchpoint
 Debugger • 478
 WhichRow method • 248
 WhichTableField method
 described • 249
 Windows
 Call Stack • 460
 Current Break Conditions for Application • 446
 DB Session for Application • 463, 482
 Debugger • 446, 449, 465, 477
 Log • 456, 482
 Log for Application • 458
 Monitor • 459, 460, 462, 482
 MonitorImage • 482
 sizing • 410
 Watch • 479