

Construct Spectrum SDK Reference

Manual Order Number: SPV451-030IBW

This document applies to Construct Spectrum SDK Version 4.5 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Readers' comments are welcomed. Comments may be addressed to the Documentation Department at the address on the back cover or to the following e-mail address:

Documentation@softwareag.com

© Copyright Software AG 2003

All rights reserved

Printed in the Federal Republic of Germany

The name Software AG and/or all Software AG product names are either trademarks or registered trademarks of Software AG. Other company and product names mentioned herein may be trademarks of their respective owners.

TABLE OF CONTENTS

PREFACE

Prerequisite Knowledge.	14
Purpose and Structure of this Documentation	14
How to Use This Documentation	16
Create a Web Application	16
Create a Client/Server Application	17
Without Using the Client Framework	17
Other Resources.	18
Related Documentation	18
Construct Spectrum SDK	18
Construct Spectrum	19
Natural Construct.	19
Other Documentation	20
Related Courses	20

1. INTRODUCTION

What is Construct Spectrum?	22
Partner Products.	22
Data Dictionary and Repository	22
Middleware	22
Programming Languages.	23
Multiple Development Environments	23
Construct Spectrum Development Environments	23
Construct Spectrum Administration Subsystem	24
Construct Windows Interface	25
Visual Basic.	26
Client/Server Applications	26
Web Applications	27
Types of Construct Spectrum Applications.	28
Architecture of Construct Spectrum Applications	29
Mainframe Server Components.	30
System Functions.	32
Windows Components.	33
Internet Information Server (IIS) Components	35
Internet/Intranet Components	35
Overview of the Development Process	36

2. SETTING UP THE MAINFRAME ENVIRONMENT

Overview	38
Setting Up Predict Definitions.	39
Field Headings.	39
Business Data Types (BDTs)	40
Default GUI and HTML Controls	40
Verification Rules	40
Default Primary Keys and Hold Fields	41
Define a Default Primary Key	41
Define a Default Hold Key	41
Default Business Object Description.	42
Descriptive Browse Fields.	42
File Volume Information in Client/Server Applications	42
Creating a Domain and Setting Up Security	43
Step 1: Define the Steplib Chain	43
Step 2: Define the Domain	45
Step 3: Define Security for the Domain	47

3. FEATURES OF THE WIZARDS

Using the Configuration Editor	50
Invoke the Configuration Editor	50
Modify the Profile Settings	52
Create a New Configuration Profile	55
Modify the Path Settings	57
Working with Code	58
Implied User Exits	58
Preserve Customizations to Generated Code	58
Regenerating Modules	59
Regenerate Individual Modules	59
Regenerate Multiple Modules	60
Regenerate External Files	61
Editing Modules	62
Generating and Reviewing Reports	63
Access Reports	63
Review a Stored Report.	64
Specify Report Detail	66
Use Reports with a Code Comparison Tool	67
Using The Spectrum Cache	68
Overview	68
Mark Nodes to be Refreshed	70
Remove Nodes From the Spectrum Cache	70

4. USING THE BUSINESS-OBJECT-SUPER-MODEL	
Overview	72
Before You Begin	73
Check the Model Defaults	73
Set up Default Values in Predict	73
Establish a Naming Convention	74
Set Up the Application Environment	75
Generating Packages	76
Step 1: Define the Standard Parameters	77
Step 2: Define the General Package Parameters	78
Step 3: Define the Specific Package Parameters	79
Step 4: Create Another Package (Optional)	81
Step 5: Generate the Modules	81
Generation Subsystem	82
Troubleshooting	83
5. USING ACTIVEX BUSINESS OBJECTS	
Overview	86
Using the ABO Project Wizard	87
Create the ABO Project	87
Framework Components for the ABO Project	91
Using the ABO Wizard	92
Customizing the ABO	98
Customize Properties Generated for the ABO	98
Opt Column	99
Customize the ABO within User Exits	100
GetAppService_.SetMethodAndBlocks	100
ICSTBrowseObject_LogicalKeyInfo.Extra	100
ICSTPersist_InstanceData.Get.Extra	100
ICSTPersist_InstanceData.Let.Extra	100
ICSTPropertyInfo_PropertyInfo.Get.Extra	101
<CounterPropertyName>.Get.NullList	101
6. USING THE SUBPROGRAM-PROXY MODEL	
Overview	104
Accessing System Files	104
Generating a Subprogram Proxy	105
Step 1: Specify Standard Parameters	106
Step 2: Specify the Number of Occurrences Returned	108
Step 3: Add User Exits	109
Step 4: Generate the Subprogram Proxy	110

Generating Methods	111
Access the Application Service Definitions	112
Add a Method	113
Step 1: Create the Method	113
Step 2: Update the Application Service Definition	113
Step 3: Update the Library Image File	114
Override the Steplib Chain for the Domain	115
Overriding Block Handling	116
Default Block Handling	116
Maintenance Subprogram Blocks Sent to Server	116
Maintenance Subprogram Blocks Returned to Client	117
Browse Subprogram Blocks Sent to Server	118
Browse Subprogram Blocks Returned to Client	118
Specify Overrides	118
Step 1: Define Block Handling On Server	118
Disable a Block Unconditionally	118
Send Blocks to the Client Conditionally	119
Step 2: Define Block Handling On Client	119
Versioning Support	120
Security Implications	120
Debugging Support	120

7. USING BUSINESS DATA TYPES (BDTS)

Overview	122
Understanding and Using BDTS	123
Benefits of Using BDTS	123
Relationship With Visual Basic Data Types	123
Composition of a BDT	124
Name	124
Conversion Routine	124
Modifiers	124
Elements of a BDT	125
BDT Controller	125
How the Client Framework Uses BDTS	125
Conversion Routines	126
ConvertToDisplay Method	127
ConvertFromDisplay Method	128
ConvertInPlace Method	128
CreateSampleString Method	129
Modifiers	129
Natural Formats	130

Handling Errors Returned from a BDT Conversion Routine	131
How Web Applications Use BDTs	132
BDTs Supplied With Construct Spectrum	133
Alpha	133
Boolean	133
Time	134
Numeric	134
Currency	135
Date	136
Referencing BDTs in Predict	137
Defining BDTs	138
Name	138
Modifiers	138
Natural Formats	138
Variant Data Types	139
Returning Conversion Error Information	140
Handling Runtime Errors	141
Creating and Customizing BDTs	141
BDTs and the Client/Server Framework	141
Understanding the BDT Objects	141
Create BDT Conversion Routines	143
Register a BDT	145
Deregister a BDT	146
Locate the Conversion Routine for a BDT	146
Create a Natural-to-BDT Mapper	147
Other Considerations	148
Use One Conversion Routine with Multiple BDTs	148
Placement of the Conversion Routine	148
Override a Supplied BDT	149
Reference BDTs in Your Application	149
BDTs and the Web Framework	150
Implement BDTs in the Web Framework	151
Register BDTs in the Web Framework	151
Register BDT Classes Using the Windows Registry	152
Explicitly Register BDT Classes	153
BDT Conversion Object	154
Create the BDT Class	156
Other BDT Controller Methods	157
Create a Natural-to-BDT Mapper	157
Create One BDT Class with Multiple BDTs	159

8. DEBUGGING YOUR CLIENT/SERVER APPLICATION

Overview	162
Communication Errors	162
Communication Error Handling	162
Traditional Debugging Tools	163
Construct Spectrum Debugging Tools	164
Types of Errors	165
Visual Basic Runtime Errors	165
Communication Errors	166
Natural Runtime Errors	166
Construct Spectrum-Related Errors	166
Errors that Do Not Return an Error Message	166
Generating Debug Data	167
Save Parameter and Debug Data	167
Set Trace Options	167
Trace Option(1)	168
Create Debug Data	170
Trace Option(2)	172
Specify Where to Save Debug Data	172
Access the Maintain User Table Panel	173
Running Spectrum Dispatch Services Online	174
Use the INPUT Statement as a Debugging Tool	174
Using Natural Debugging Tools	175
Invoke Subprogram Proxies Online	175
Access the Invoke Proxy Function	176
Debugging Tools on the Client and Server	177
Diagnostics Window	177
Translations Program	181
Troubleshooting	183
Registry Usage	183
SDC.ini	183
SDCApp.ini	184
Check for Necessary DLLs	184
Construct Spectrum Add-In	184
Useful SDC Properties	185
Application Object	185
NaturalDataArea Object	185
Dispatcher Object	186
RequestProperty Property	186

9. DEPLOYING YOUR CLIENT/SERVER APPLICATION

Transferring Data.	190
Data Transfer Utilities	190
Construct Spectrum Administration Subsystem	190
Distributing Your Application.	191
Step 1: Create the Executable File.	191
Step 2: Collect Files For Installation.	191
Step 3: Install the Client Application	192
Step 4: Run the Application	192

10. USING THE SPECTRUM DISPATCH CLIENT

Overview	194
Calling a Natural Subprogram.	195
Step 1: Create Parameter Data Area Instances	195
Step 2: Assign Values to the Fields.	195
Step 3: Use the CallNat Method on the Client	196
Step 4: Check the Success of the CALLNAT.	196
Summary	196
Spectrum Dispatch Client Components.	197
Natural Data Area Simulation	198
Data Area Definitions	198
Data Area Simulation Objects	200
Application Object.	201
Create NaturalDataArea Objects	202
NaturalDataArea Class	202
Case Sensitivity	206
Alphanumeric Fields	206
Fully Qualified Field Names	206
Redefined Fields	207
Errors When Compiling.	207
Read Arrays and Structures	208
Runtime Errors	209
DataDefinitionArea Class	209
NaturalFieldDef Class	210
Client/Server Communication	213
Level 1 Block Optimization	213
Application Service Definitions	215
Dispatcher Objects and Dispatch Service Definitions.	218
Service Selection	219
Remote Subprogram Invocation	220
Timeout, Retry, and Resume Handling	221
Compression and Encryption.	223

Tracing	224
Database Transaction Control	224
Error Reporting	225
User Identification and Authentication	226
Library Image Files and the Steplib Chain	227
Advanced Features	228
FieldRef Property	228
1:V Fields	233

11. CREATING APPLICATIONS WITHOUT THE FRAMEWORK

Setting Up the Server Components	236
Create or Select Application Services	236
No Terminal I/O	236
Subprogram Interface	236
No Global Data Area (GDA)	236
Parameter Data Area (PDA) Data Size Limitation	237
Subprogram Behavior	237
Externalize Parameters	237
Timing Issues	237
Example of Creating a Simple Natural Subprogram	238
Generating Subprogram Proxies	240
Subprogram-Proxy Model	240
Application Service Definition	242
Creating the Library Image Files (LIFs)	244
Construct Spectrum Add-In	244
Before You Start	244
Download Definitions	245
Developing the Client Application	248
Step 1: Create a New Project	249
Step 2: Add a Reference to the SDC Object Library	249
Step 3: Write Code to Initialize the SDC	250
Step 4: Create the User Interface	251
Step 5: Write Code to Call the Subprogram	252
Step 6: Run the Application	253

APPENDIX A: GLOSSARY255

APPENDIX B: UTILITIES271

Response Subprogram	272
Features and Benefits	272
Response Length Limitation	272
Supported Methods	272

Message Protocol	273
Call Interface	273
SPAREPLY Data Area	274
SPAREPM Data Area	276
Spectrum Interface Subprogram	278
Features and Benefits	278
Broker Error Handling	278
Error Logging	278
Shutdown Requests	278
Server Timeouts	278
Command Handling	279
SPUETB Interface	279
Data Areas	280
SPAETB Data Area	280
ETBCB Data Area	285
SEND-BUFFER	285
RECEIVE-BUFFER	285
RESERVED-AREA	286
CDPDA-M	286
Using SPUETB	286
CMD TRACE	286
Valid Keywords	287
Trace Response	288
Test the Trace Facility	288
CMD CALLNAT	288
Conversation Factory Utility	289
Character Translation Subprogram	290
Determine a Character Set	290
Multi-Tasking Verification Utility	291
Log Utilities	292
Spectrum Log Utilities	292
Construct Spectrum Control Record Log Utilities	293
Domain Log Utilities	294
Spectrum Group Log Utilities	295
Application Service Definition Log Utilities	296
Spectrum Steplib Log Utilities	297
User and Group Log Utilities	298
INDEX	299

PREFACE

Welcome to *Construct Spectrum SDK Reference*, a reference tool for developers using the Construct Spectrum software development kit (SDK). This preface will help you get the most out of the documentation and find other sources of information about creating Construct Spectrum applications.

The following topics are covered:

- **Prerequisite Knowledge**, page 14
- **Purpose and Structure of this Documentation**, page 14
- **How to Use This Documentation**, page 16
- **Other Resources**, page 18

Prerequisite Knowledge

This documentation does not provide information about the following topics. We assume that you are either familiar with the topics or have access to other sources of information about them.

- Natural Construct
- Microsoft Visual Basic
- Predict
- Natural programming language and environment
- Entire Broker
- Entire Net-Work

See **Other Resources**, page 18, for sources of information about Natural Construct and Construct Spectrum.

Note: The examples used in this guide are from the Construct Windows interface. For examples from the Generation subsystem, see the appropriate chapter in *Natural Construct Generation*.

Purpose and Structure of this Documentation

Construct Spectrum SDK Reference is designed to help developers create client/server and web applications and to customize, debug, and deploy applications. For information about how to use this document, see **How to Use This Documentation**, page 16.

The following table describes the information contained in each chapter:

Chapter	Title	Topics
1	Introduction , page 21	Describes the components of Construct Spectrum and the architecture of the applications you can create using the software development kit (SDK).
2	Setting up the Mainframe Environment , page 37	Describes the tasks you must perform on the mainframe before generating a client/server or web application.
3	Features of the Wizards , page 49	Introduces you to the Construct Spectrum Add-In tools used to build, customize, and support Spectrum web and ABO projects.

Chapter	Title	Topics (continued)
4	Using the Business-Object-Super-Model , page 71	Describes how to generate multiple Natural components using the Business-Object-Super-Model.
5	Using ActiveX Business Objects , page 85	Describes how to generate ActiveX business objects (ABOs) — Visual Basic classes that wrap the Spectrum calls required to communicate with Natural subprograms exposed by subprogram proxies.
6	Using the Subprogram-Proxy Model , page 103	Describes the subprogram proxy, how to generate proxies using the Subprogram-Proxy model, and how to customize your proxy.
7	Using Business Data Types (BDTs) , page 121	Describes business data types (BDTs) as they relate to client/server and web applications.
8	Debugging Your Client/Server Application , page 161	Describes how to debug your Construct Spectrum-generated client/server applications.
9	Deploying Your Client/Server Application , page 189	Describes how to deploy your Construct Spectrum-generated client/server applications.
10	Using the Spectrum Dispatch Client , page 193	Describes the Spectrum Dispatch Client, which allows you to make calls from a client to Natural subprograms running on a server.
11	Creating Applications Without the Framework , page 235	Describes how to create a Construct Spectrum application without using Construct-generated components.
Appendix A	Appendix A: Glossary , page 255	Contains a glossary of terms used throughout the Construct Spectrum documentation set.
Appendix B	Appendix B: Utilities , page 271	Describes the utilities supplied with the Spectrum Administration subsystem.

How to Use This Documentation

Construct Spectrum SDK Reference provides information about core development tasks that the majority of Construct Spectrum users perform, whether they are creating:

- Construct Spectrum web applications
- Construct Spectrum client/server applications
- Client/server applications without using the Construct Spectrum client framework

The following sections describe how to use this and related guides to perform these types of tasks.

Create a Web Application

To use Construct Spectrum to create the components of a web application, read:

- **Introduction**, page 21, for an overview of the product, development process, and the applications you can develop.
- **Setting up the Mainframe Environment**, page 37, for detailed information about how to define domains and security options to control what data users of your application will access on the mainframe.
- **Features of the Wizards**, page 49, for information about setting configuration options for the wizards, using the client-side cache, and modifying code frames.
- **Using the Business-Object-Super-Model**, page 71, for detailed information about how to use this model wizard to generate the Natural components of your application.
- **Using ActiveX Business Objects**, page 85, for detailed information about creating ABOs and an ABO project to contain them using the wizards supplied with Construct Spectrum.
- *Construct Spectrum SDK for Web Applications* for detailed information about creating the web components of your application.

To customize and regenerate application components, read:

- **Using the Subprogram-Proxy Model**, page 103
- **Using Business Data Types (BDTs)**, page 121

Create a Client/Server Application

To use Construct Spectrum to create a client/server application to run on Windows 95, Windows 98, Windows 2000, or Windows NT, read:

- **Introduction**, page 21, for an overview of the product, development process, and the applications you can develop.
- **Setting up the Mainframe Environment**, page 37, for detailed information about how to define domains and security options to control what data users of your application will access on the mainframe.
- *Construct Spectrum SDK for Client/Server Applications* for detailed information about using the VB-Client-Server-Super-Model to generate your application components. It explains how to set up a Visual Basic project and customize maintenance and browse dialogs. Also refer to this guide if you want to move existing server-based applications to the Construct Spectrum client/server architecture.

To customize and regenerate applications components, read:

- **Using the Subprogram-Proxy Model**, page 103
- **Using Business Data Types (BDTs)**, page 121
- **Debugging Your Client/Server Application**, page 161
- **Deploying Your Client/Server Application**, page 189

Without Using the Client Framework

To create a client/server application without using the Construct Spectrum client framework, read:

- **Introduction**, page 21, for an overview of the product, development process, and applications you can develop.
- **Using the Spectrum Dispatch Client**, page 193, for detailed information about the role of the SDC in client/server communication.
- **Creating Applications Without the Framework**, page 235, for step-by-step procedures to create your application.

Other Resources

This section provides information about other resources you can use to learn more about Construct Spectrum and Natural Construct. For more information about these documents and courses, contact the nearest Software AG office or visit the website at www.softwareag.com to order documents or view course schedules and locations. You can also use the website to email questions to Customer Support.

Related Documentation

This section lists other documentation in the Construct Spectrum and Natural Construct documentation set.

Construct Spectrum SDK

- *Construct Spectrum SDK for Microsoft .NET Framework*
This guide is for developers creating Microsoft .NET Web services to invoke Natural subprograms (business objects) over the Inter/Intranet via the W3C SOAP standard.
- *Construct Spectrum SDK for Web Applications*
This documentation is for developers creating the web components of applications. It describes how to use the Construct Spectrum wizards in Visual Basic to generate HTML templates, page handlers, and object factory entries. It also contains detailed information about customizing, debugging, deploying, and securing web applications.
- *Construct Spectrum SDK for Client/Server Applications*
This documentation is for developers creating client components for applications that will run in a Natural mainframe (server) and Windows (client) environment.
- *Construct Spectrum Messages*
This documentation is for application developers, application administrators, and system administrators who want to investigate messages returned by Construct Spectrum runtime and SDK components.

Construct Spectrum

- *Construct Spectrum Administration*
This guide is for administrators who want to use the Construct Spectrum Administration subsystem to set up and manage Construct Spectrum applications.
- *Construct Spectrum and SDK Vn Release Notes*
These notes contain information on new features, enhancements, and other changes in this version of Construct Spectrum.
- *Construct Spectrum Reference*
This documentation is for application developers and administrators who need quick access to information about Construct Spectrum application programming interfaces (APIs) and utilities.
- *Construct Spectrum and SDK Installation Guide for Windows*
This documentation describes how to install and set up the Construct Spectrum runtime and SDK components on the client.
- *Construct Spectrum and SDK Installation Guide for Mainframes*
This documentation describes how to install and set up the Construct Spectrum runtime and SDK components on the mainframe.

Natural Construct

- *Natural Construct Installation Guide for Mainframes*
This documentation provides essential information for setting up the latest version of Natural Construct, which is needed to operate the Construct Spectrum programming environment.
- *Natural Construct Generation*
This documentation describes how to use the Natural Construct models to generate applications that will run in a mainframe environment.
- *Natural Construct Administration and Modeling*
This documentation describes how to use the Administration subsystem of Natural Construct and how to create new models.
- *Natural Construct Help Text*
This documentation describes how to create online help for applications that run on server platforms.
- *Natural Construct Getting Started Guide*
This guide introduces new users to Natural Construct and provides step-by-step instructions to create several common processes.

Other Documentation

This section lists documents published by WH&O International:

- *Natural Construct Tips & Techniques*
This book provides a reference of tips and techniques for developing and supporting Natural Construct applications.
- *Natural Construct Application Development User's Guide*
This guide describes the basics of generating Natural Construct modules using the supplied models.
- *Natural Construct Study Guide*
This guide is intended for programmers who have never used Natural Construct.

Related Courses

In addition to the documentation, the following courses are available from Software AG:

- A self-study course on Natural Construct fundamentals
- An instructor-led course on building applications with Natural Construct
- An instructor-led course on modifying the existing Natural Construct models or creating your own models

INTRODUCTION

This chapter describes the components of Construct Spectrum and the architecture of the applications you can create with the software development kit (SDK). An overview of the general steps involved in developing applications will prepare you for the detailed procedures in this and related guides.

The following topics are covered:

- **What is Construct Spectrum?**, page 22
- **Types of Construct Spectrum Applications**, page 28
- **Architecture of Construct Spectrum Applications**, page 29
- **Overview of the Development Process**, page 36

What is Construct Spectrum?

Construct Spectrum and the software development kit (SDK) comprise a set of middleware and framework components, as well as integrated tools, that use the specifications you supply to generate the components of a distributed application.

Construct Spectrum comprises two products:

- The SDK is a set of tools, wizards, and framework components you can use to build client/server and web applications.
- Construct Spectrum is a middleware product that facilitates communication between client and server.

Partner Products

Construct Spectrum works with several other products to help you build applications. The following sections provide a brief overview of these products. For more information about these products, consult the appropriate documentation.

Data Dictionary and Repository

Construct Spectrum works closely with Predict, a data dictionary and repository that manages metadata about the information contained in the database your applications use. Predict's "views" of data, and the relationships between data, help you define the business objects your applications access and maintain. Predict verification rules and keywords validate and format data and its field definitions automatically select controls for your applications. You can also use Predict to define the defaults Construct Spectrum uses to generate your applications.

Middleware

Construct Spectrum uses Entire Broker, either with Entire Net-Work or configured to use TCP/IP, to communicate between the client and server components of the application.

Your applications also use Construct Spectrum's middleware components — the Spectrum Dispatch Client (SDC) and Spectrum dispatch service — to encapsulate calls to Entire Broker on the client and server and to perform such functions as data translation, encryption, and compression. When the client makes a communication request, the SDC translates the request into a compact, secure message and transmits it to the server via Entire Broker. On the server, the Spectrum dispatch service converts the incoming processing request by the server application while enforcing multi-level security. Construct Spectrum then uses a similar technique to return the processed result to the client.

Programming Languages

Construct Spectrum applications incorporate Natural and Visual Basic code. You can also develop client/server applications using other OLE-compliant languages.

To present data dynamically for web applications, generated web pages use JavaScript and HTML, including the supplied Construct Spectrum HTML replacement tags. For information, see **Using HTML Replacement Tags**, page 121, *Construct Spectrum SDK for Web Applications*.

Multiple Development Environments

Besides its own development environments, Construct Spectrum provides tools that are integrated with the Natural and Visual Basic development environments. This allows you can take advantage of the functionality of each, such as the Natural code editors or the Visual Basic debugging facilities.

The following section provides information about the Construct Spectrum development environments.

Construct Spectrum Development Environments

As you develop applications, you will work in at least three environments:

- Construct Spectrum Administration subsystem
- Construct Windows interface
- Visual Basic, using the Construct Spectrum Add-Ins

The following sections describe these environments.

Construct Spectrum Administration Subsystem

Use the Construct Spectrum Administration subsystem on the mainframe to manage system and application data for your applications:

```

BS__MAIN  ***** Construct Spectrum Administration Subsystem *****   CDLAYMN1
Jul 30                - Main Menu -                                     10:14 AM

                Functions
                -----
                SA   System Administration
                AA   Application Administration

                ?   Help
                .   Terminate
                -----
Function .....  ___

Command .....  _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      help retn quit          flip                               main

```

Construct Spectrum Administration Main Menu

For information about the Construct Spectrum Administration subsystem on the mainframe, see *Construct Spectrum Administration*.

Construct Windows Interface

Use the Construct Windows interface (CWI) on your PC to generate Natural and Visual Basic modules for your application:



New Specification Window — Construct Windows Interface

The wizards available in the CWI are available as models in the Generation subsystem in your Natural Construct mainframe environment. For details about the supplied models, see *Natural Construct Generation*.

Visual Basic

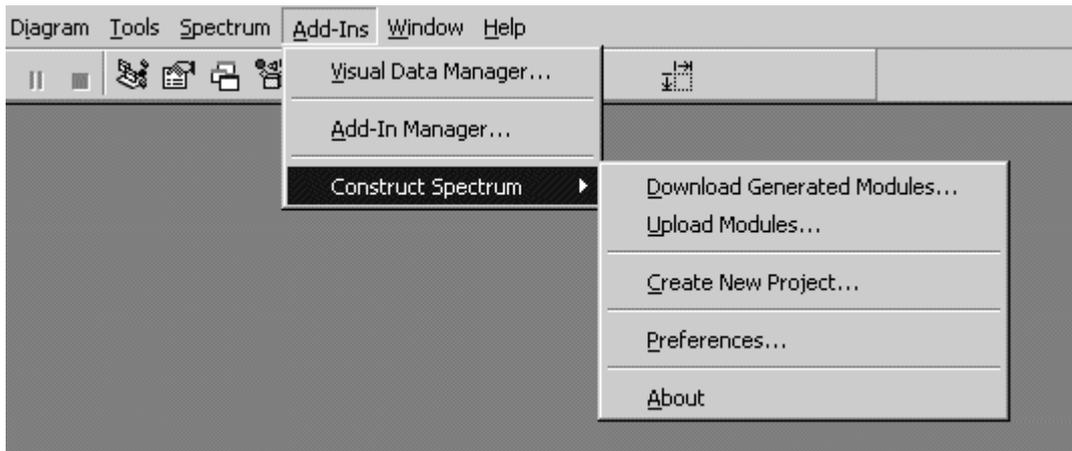
Use the Construct Spectrum Add-Ins in Visual Basic to create projects, work with Visual Basic modules, and generate ActiveX business objects and web components.

Client/Server Applications

For a client/server application, use the Construct Spectrum options on the Visual Basic Add-Ins menu to:

- Download generated modules from the mainframe server
- Upload modules to the mainframe server
- Create a new project
- Set preferences

The following example shows the Construct Spectrum options on the Add-Ins menu:



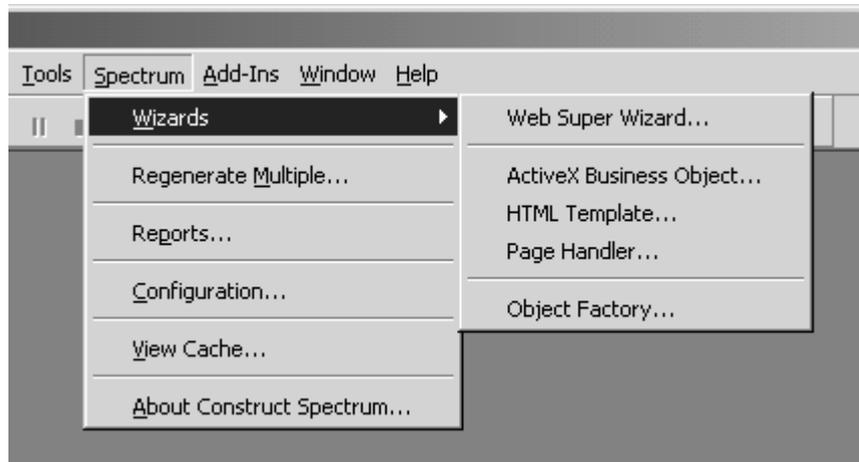
Construct Spectrum Options — Visual Basic Add-Ins Menu

Web Applications

For a web application, use the desktop Construct Spectrum project wizards to create ABO and other projects. Use the options on the Spectrum menu in Visual Basic to:

- Access wizards to generate web components
- Regenerate modules
- Define report options
- Set configuration options
- View the cache of server data

The following example shows the Spectrum menu options:



Spectrum Menu in Visual Basic

You also work with an HTML editor of your choice, the Microsoft Management Console to manage your Microsoft Internet Information Server on Windows NT, and/or the Personal Web Server (if you are using Windows to develop applications).

Information about how to access and use these environments is presented where required throughout this documentation.

Types of Construct Spectrum Applications

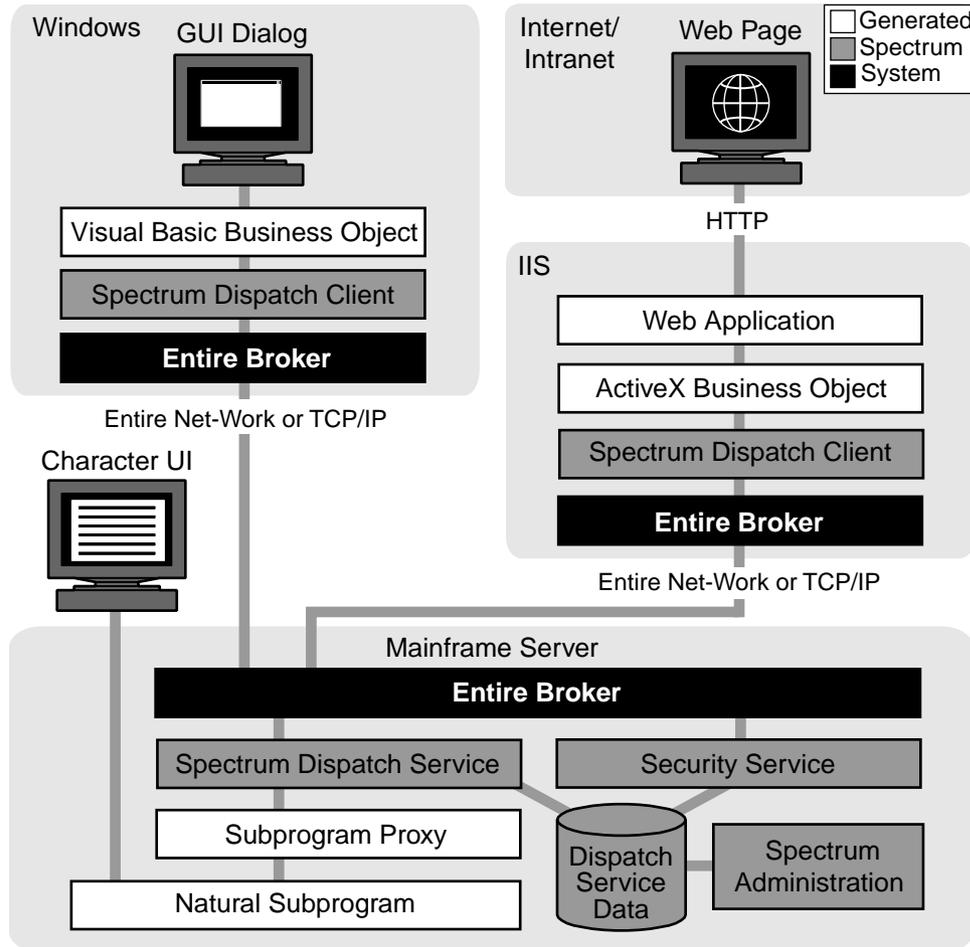
Using Construct Spectrum SDK, you can create two kinds of applications:

- Client/server applications that run on Windows or Windows NT (client) and access Natural components and data on a mainframe (server).
Client/server applications are composed of Natural modules that encapsulate maintenance and query functions on the server, Visual Basic components that function on the client and present the user interface, and runtime components that communicate between client and server.
- Web applications that run on IIS and can be accessed with Microsoft Internet Explorer and Netscape Navigator.
Web applications are composed of Natural modules that encapsulate maintenance and query functions on the server, ActiveX business objects that communicate between client and server components, page handlers that manage the processing of HTML templates, and HTML templates that present web pages.

This guide describes how to develop components and functionality that are common to the different types of applications. Information specific to client/server applications is contained in *Construct Spectrum SDK for Client/Server Applications*. Information specific to web applications created with Construct Spectrum is contained in *Construct Spectrum SDK for Web Applications*.

Architecture of Construct Spectrum Applications

The following diagram shows the architecture of Natural character-based applications, client/server applications, and web applications:



Architecture of Construct Spectrum Applications

The following sections describe these components according to the platforms on which they run: mainframe server, Windows, IIS, and internet or intranet.

Mainframe Server Components

Component	Description
Natural subprograms	<p>Perform maintenance and browse functions on the mainframe server. The same set of business objects can be accessed from character-based Natural applications, client/server applications, and web applications. This ensures that the integrity of business data is preserved, independent of the presentation layer.</p> <p>Natural subprograms may be either generated by Construct models or written by hand. The models that generate subprograms and their parameter data areas (PDAs) are the VB-Client-Server-Super-Model, Business-Object-Super-Model, Object-Maint-Subprogram model, and Object-Browse-Subprogram model.</p> <p>Natural subprograms can also be written by hand if they follow certain guidelines. For example, screen I/O functions are not allowed, and records cannot be held between conversations.</p>
Character UI	<p>Non-distributed Natural applications created with Natural Construct accessing subprograms directly.</p>
Subprogram proxy	<p>Acts as a bridge between a subprogram and the Spectrum dispatch service. The subprogram proxy:</p> <ul style="list-style-type: none"> • Provides a common interface so that the Spectrum dispatch service can pass the same set of parameters to any subprogram proxy • Issues a CALLNAT to the subprogram • Converts the parameter data of the subprogram into a format that can be transmitted between the client and server • Supports optimization of the data passed through the network so that only input parameters need to be sent to the Spectrum dispatch service and only output parameters need to be returned to the client • Validates the format and length of the data received from the client • Supports debugging features to help uncover inconsistencies between the data sent by the client and the data expected by the subprogram proxy <p>For more information, see Using the Subprogram-Proxy Model, page 103.</p>

Component	Description (continued)
Spectrum dispatch service	<p>Provides a common interface and Entire Broker services for Natural subprograms in the application. The main functions of the Spectrum dispatch service are to:</p> <ul style="list-style-type: none"> • Receive requests from the client by way of Entire Broker • Optionally decompress or decrypt (or both) and translate the request message (see System Functions, page 32) from the client's character set (ASCII) to the server's character set (either ASCII or EBCDIC) • Check security to ensure that the client is allowed to issue such a request • Determine the name of the subprogram proxy that handles the request • Issue a CALLNAT to the subprogram proxy, passing the received message as a parameter string • Optionally compress, encrypt (or both) the message to be returned (see System Functions, page 32) • Send information received from the subprogram proxy back to the client application
Dispatch service data	<p>Information defined and maintained in the Construct Spectrum Administration subsystem and accessed by Spectrum dispatch services anywhere on the network by way of Entire Broker.</p>
Construct Spectrum Administration subsystem	<p>Allows system administrators, application administrators, and application developers to set up and manage system and application environments. For more information, see <i>Construct Spectrum Administration</i>.</p>
Security service	<p>Checks client requests against security settings defined in the Construct Spectrum Administration subsystem. This stand-alone service operates independently of any one Spectrum dispatch service. Its independence allows the security service to process, in one central location, the requests of several Spectrum dispatch services, which may be located on nodes throughout the network.</p> <p>For more information about security services and security settings, see <i>Construct Spectrum Administration</i>.</p>
Entire Broker	<p>Transfers messages between Windows or the web server and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.</p>

System Functions

All Spectrum dispatch services defined in the Construct Spectrum Administration subsystem have access to the following common system functions:

Function	Description
Return debugging information	Ensures that all requested debugging information is generated into the source area. Debugging information is requested by setting a Trace-Option in the subprogram proxy. The debugging information is stored as a source member that can be examined or used to initiate the request locally on the server, removing the client and the network from the test. For more information about trace options, see Debugging Your Client/Server Application , page 161.
Encrypt and decrypt data	Supplies an interface that can be called by the external (assembler or C) routines used to encrypt and decrypt data.
Compress and decompress data	Supplies an interface that can be called by the external (assembler or C) routines used to compress and decompress data.
Error handling	Manages the capturing of runtime errors, returning the errors to the client. If possible, this function also restarts the service that ended with the runtime error.
Message handling	Returns a message string based on a message number and substitution values. The function accepts and updates the data used by the Spectrum dispatch service to return the message.
Data translation	Translates data received from the client into EBCDIC or ASCII, depending on the requirements of the server.

Windows Components

Construct Spectrum client/server applications run on Windows or Windows NT. The Windows components are:

Component	Description
Entire Broker	Transfers messages between the client and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.
Spectrum Dispatch Client (SDC)	Component Object Model (COM) middleware component that enables Construct Spectrum applications to read from, and write to, variables in a Natural parameter data area (PDA) and to issue CALLNAT statements to Natural subprograms.

The main functions of the SDC are:

- **Natural parameter data area simulation**
The SDC defines the parameter data of Natural business objects as a series of Natural data fields, which may include structures, arrays, and redefines. To call a business object, the application must be able to assign values to these parameter data fields before calling the business object and then read the fields after the data is returned from the server.

To facilitate this, the SDC simulates Natural parameter data areas, allowing the application developer to create code that allocates a data area and reads and writes the fields in the data area. The Construct Spectrum Add-In can download Natural parameter data areas (residing in a library on the server) to the client. This lets the SDC know the structure (field names and formats) of a parameter data area. Parameter data areas are stored in the library image file on the client and only need to be downloaded after creation or whenever they change on the server.
- **CALLNAT simulation**
The SDC allows an application to issue a CALLNAT to a Natural subprogram. To allow this, specify the logical name of the subprogram to be called, and the list of parameter data areas to pass to the subprogram, in the client code.
- **Encapsulation of Entire Broker calls**
The SDC uses Entire Broker calls to communicate with the Spectrum dispatch service. These calls are not exposed within the application layer, so the application developer never needs to code Entire Broker calls.

Component	Description (continued)
	<ul style="list-style-type: none"> <li data-bbox="639 331 1369 617">Database transaction control Often, two or more calls to subprograms occur within the same database transaction such that an END TRANSACTION statement can be issued if all calls complete successfully. Also, it is advantageous to have the client application control the point at which the END TRANSACTION or BACKOUT TRANSACTION statement occurs. The SDC and the Spectrum dispatch service cooperate to provide these capabilities. <p data-bbox="639 636 1369 695">For more information, see Using the Spectrum Dispatch Client, page 193.</p>
Visual Basic business object	Visual Basic class that acts as an intermediary between a dialog and the SDC. This class invokes the methods of subprograms on behalf of dialogs and instantiates all the data areas required to communicate with a subprogram. Visual Basic business objects can also perform local data validation to provide immediate feedback to the user without involving a network call.
GUI dialog	GUI dialogs represent graphical interface screens that communicate with the user and interact with the Visual Basic business objects and other framework components to implement business processes.

Internet Information Server (IIS) Components

Web applications created with Construct Spectrum work with IIS. The IIS components are:

Component	Description
Entire Broker	Transfers messages between the web server and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.
Spectrum Dispatch Client	Component Object Model (COM) middleware component that enables web applications to read from, and write to, variables in a Natural parameter data area (PDA) and to issue CALLNAT statements to Natural subprograms. Its main functions are simulating PDAs and CALLNATs, encapsulating Entire Broker calls, and controlling database transactions. As the client counterpart of Spectrum dispatch services, it is also responsible for such things as data marshaling, encryption, compression, error-handling, and all Entire Broker communication. For more information, see Using the Spectrum Dispatch Client , page 193.
ActiveX Business Object	Object that encapsulates all communication with the SDC, making it efficient to invoke Natural services from the client. Each back-end business object is represented on the web server as an ActiveX object. For more information, see Using ActiveX Business Objects , page 85.
Web application	Consists of framework components supplied with all Construct Spectrum web projects and components that you generate using Construct Spectrum wizards. Generated components are HTML templates, page handlers, and object factory entries. For more information, see Architecture of a Web Application , page 23, <i>Construct Spectrum SDK for Web Applications</i> .

Internet/Intranet Components

Construct Spectrum-generated web applications support Internet Explorer and Netscape Navigator browsers at version 4 or higher. For additional functionality:

- Internet Explorer V5 provides improved HTML rendering and the ability to bookmark web pages in Frames mode.
- Internet Explorer V5.5 and Netscape Navigator V6 support fly-out menus.

Overview of the Development Process

This section provides an overview of the steps involved in developing a Construct Spectrum application. For detailed information, see the following sources:

- For an overview of developing web applications, see **Overview of the Development Procedure**, page 28, *Construct Spectrum SDK for Web Applications*
- For an overview of developing client/server applications, see **Overview of the Development Procedure**, page 30, *Construct Spectrum SDK for Client/Server Applications*
- For an overview of developing client/server applications without the Construct Spectrum framework components, see **Creating Applications Without the Framework**, page 235

➤ To develop a Construct Spectrum application:

- 1 Plan your application
You will save time and effort by planning as completely as possible the purpose, functionality, security, and user interface of your application.
- 2 Set up your application environment
Based on the functionality of your application, ensure that the file, field, and relationship definitions in Predict support the business objects and business rules your application will use. Also set up a domain and steplib chain in the Construct Spectrum Administration subsystem so the application can access the appropriate data. You may also want to define users, groups, and security settings in this step.
- 3 Generate application components
Use the Construct Spectrum models and/or wizards to enter specifications for your application components and generate them. For the first iteration, use the super model wizards to create multiple components. For Natural modules and client/server Visual Basic components, use either the models in the Generation subsystem on the mainframe or the model wizards in the Construct Windows interface. To create a web application, also use the wizards Construct Spectrum adds to Visual Basic. This step also involves creating new Visual Basic projects and populating them with components.
- 4 Customize, test, and debug the application
Customize the look and functionality of your application. This iterative process may require you to regenerate modules using the individual supplied models.
- 5 Deploy the application
When your application is fully functional, you are ready to distribute it to users. This step can involve creating an installation kit and deploying the Construct Spectrum Administration subsystem.

SETTING UP THE MAINFRAME ENVIRONMENT

This chapter describes the tasks you must perform on the mainframe before generating a client/server or web application.

Note: Before performing the tasks described in this chapter, ensure that all required software is installed and configured on your server and client. For information, see *Construct Spectrum and SDK Installation Guide for Mainframes* and *Construct Spectrum and SDK Installation Guide for Windows*.

The following topics are covered:

- **Overview**, page 38
- **Setting Up Predict Definitions**, page 39
- **Creating a Domain and Setting Up Security**, page 43

Overview

Before you can generate applications, you must complete some setup tasks to ensure that your application can access the database records it needs and that users will be able to access the application. The following tasks are involved:

- Set up file and field definitions in Predict. You can also affect how field names and controls are derived and how validations are performed by adjusting Predict settings.
- Create and associate a steplib chain and domain in the Construct Spectrum Administration Subsystem.
- Set up security privileges for the domain. This involves defining users and groups and linking them to the domain in the Construct Spectrum Administration subsystem.

This chapter describes these steps in more detail.

Setting Up Predict Definitions

With any application created with Natural Construct or Construct Spectrum, you must set up file and field definitions in Predict. This includes setting up your application files and defining their intra- and inter-object relationships.

For information about these tasks, see **Design Methodology**, page 143, and **Use of Predict in Natural Construct**, page 697, *Natural Construct Generation*.

Predict features that have special implications for Construct Spectrum applications include:

- Field headings
- Business data types
- Default GUI and HTML controls
- Verification rules
- Primary keys and hold fields
- Default business object description
- Descriptive browse fields

Tip: You can postpone the setup tasks described in this section until a later iteration of your application. These tasks may be optional and, in all cases, Construct Spectrum applies its own values for these setup items based on your existing Predict file and field definitions.

Field Headings

If a field definition has a heading in Predict, the heading is used to derive the caption for the control on the dialog or page. If no heading is coded in Predict, the caption is generated by converting the field name to mixed case and changing special characters (dashes and underscores) to spaces.

When creating a client/server application, you can change the captions on the form in Visual Basic.

When creating a web application, you can modify captions using the HTML Template wizard in Visual Basic. For more information, see **Creating and Customizing an HTML Template**, page 91, *Construct Spectrum SDK for Web Applications*.

Business Data Types (BDTs)

Business data types (BDTs) associate additional formatting with data fields to help ensure that data is presented consistently and validated in your application. By default, generated modules implement basic format and length-checking to ensure that all values stored on the client are of a valid format and length. BDTs extend this concept by allowing the use of user-defined data types related to business representations of the data. For example, a numeric field might be intended to store a currency amount, a net weight, a date, or a quantity. Each of these values might be presented to the user and validated in a different way, although they are all defined as numeric fields. For example, a credit card number could be stored on the database as a 16-digit value. However, when this value is placed on a page, it could be shown using the 9999-9999-9999-9999 format. Furthermore, the user could update the value with or without the dashes, and the BDT will ensure that the unformatted value is assigned back to the database.

To associate a database field with a BDT, assign a special BDT keyword to the field in Predict. For more information, see **Using Business Data Types (BDTs)**, page 121.

Default GUI and HTML Controls

Construct Spectrum applies complex derivation rules to determine the most appropriate control to represent a database field. Nevertheless, there may be times when the default control is not ideal for a particular application. In these cases, you can override the default control by assigning the database field a special keyword. If you are creating a web application, you can change some controls in the HTML Template wizard.

For more information, see **Overriding GUI Controls**, page 133, *Construct Spectrum SDK for Client/Server Applications*, or **Creating and Customizing an HTML Template**, page 91, *Construct Spectrum SDK for Web Applications*.

Verification Rules

Verification rules are used to force the application user to make a selection based on one or more predetermined choices. For example, if your application has a field where a valid month must be entered, you can specify a verification rule for the field so that only a valid month will be accepted.

One criteria that Construct Spectrum uses to determine the most appropriate GUI or HTML control for a particular field is the presence of verification rules attached to the field. In the previous example of presenting valid months, Construct Spectrum would attach a drop-down combo box to the field in the dialog or page. The user could select a valid value from the drop-down combo box.

For more information, see **Overriding GUI Controls**, page 133, *Construct Spectrum SDK for Client/Server Applications*, or **Creating and Customizing an HTML Template**, page 91, *Construct Spectrum SDK for Web Applications*.

Default Primary Keys and Hold Fields

Predict keywords can also be used to designate default primary key values and logical hold fields for a super model, which reduces the specifications the user must enter.

Define a Default Primary Key

To define a default primary key, specify a descriptor name in the Sequence field for the file in Predict. Natural Construct observes the following priorities when defaulting a primary key name for a file:

- 1 If the value of the default Sequence field for the file is unique and a valid descriptor, Natural Construct uses this value as the primary key.
- 2 If the value of the default Sequence field is not unique, Natural Construct reads through the file and uses a unique descriptor field value as the primary key.
- 3 If the file does not have a unique descriptor field, but has only one descriptor field, Natural Construct assumes the value is unique and uses it as the primary key.

Define a Default Hold Key

To define a default logical hold field, attach the HOLD-FIELD keyword to the field in Predict. (You may have to first define the HOLD-FIELD keyword in Predict using Keyword Maintenance.) Natural Construct observes the following priorities when defaulting a hold field name for a file:

- 1 If the HOLD-FIELD keyword is attached to a field that meets the format criteria for a hold field, Natural Construct uses this field as the logical hold field.
- 2 If a field name contains any of the following strings:
 - HOLDFIELD
 - HOLD-FIELD
 - HOLD_FIELD
 - TIMESTAMP
 - TIME-STAMP
 - TIME_STAMP
 - LOGCOUNTER
 - LOG-COUNTER
 - LOG_COUNTER

and the field meets the format criteria for a hold field, Natural Construct uses this field as the logical hold field.

Default Business Object Description

To specify a default business object description, assign a name to the file's Literal Name attribute in Predict. This name is defaulted as the business object description when using a super model. Additionally, this name is displayed when the file is referenced in error messages.

Descriptive Browse Fields

When the user invokes a browse dialog attached to a field on a maintenance form, it is referred to as a foreign field browse. When invoked, a foreign field browse displays only the foreign field values unless you designate other fields in the foreign file as descriptive. For example, suppose you know that the warehouse number field in a warehouse file will be referenced as a foreign field browse on a number of maintenance dialogs or pages. To help users select the correct warehouse when browsing, you can designate another field, such as the Warehouse Name field, as descriptive. When users browse for a warehouse number, the descriptive value (in this case, a warehouse name) is displayed, along with the warehouse number.

A descriptive field is designated in Predict by associating a special keyword with the field. You can indicate that certain fields are descriptive in all situations, while others are descriptive depending on the form or page that contains the foreign field.

For information about descriptive fields, see **Displaying Descriptions for a Foreign Field**, page 289, *Construct Spectrum SDK for Client/Server Applications*, or **Creating and Customizing an HTML Template**, page 91, *Construct Spectrum SDK for Web Applications*.

File Volume Information in Client/Server Applications

You can supply information related to the size and stability of your files in Predict. These values are used to determine the default behavior of a standalone browse dialog and browse dialogs linked to a maintenance dialog. For more information about linking browse and maintenance functions, see **Integrating Browse and Maintenance Functions**, page 275, *Construct Spectrum SDK for Client/Server Applications*.

Creating a Domain and Setting Up Security

The application environment includes users, application libraries, business objects and their associated modules. Users are combined into larger entities known as “groups”. Application libraries, business objects and their associated modules are combined into larger entities known as “domains”. Before creating an application with Construct Spectrum, you must define a domain for the application. Before users can access the application, you must grant access to the business objects and object methods within the domain.

- To create a domain and set up security:
 - ❑ **Step 1: Define the Steplib Chain**, page 43
 - ❑ **Step 2: Define the Domain**, page 45
 - ❑ **Step 3: Define Security for the Domain**, page 47

The following sections describe each of these steps in detail.

Step 1: Define the Steplib Chain

The first step in setting up a domain is to define its steplib chain. A steplib chain identifies where your application libraries reside on the server. To locate and execute application modules, you must set up a steplib chain and link it to your application domain.

When defining your steplib chain, keep the following tips in mind:

- Before adding a steplib entry, determine the database ID (DBID) and file number (FNR) of the FUSER system file you are using.
- The library in which the dispatch service is executing is checked before libraries in the steplib chain; you do not have to add this library to your steplib chain.
- If you intend to use the default DBID and FNR values for the current FUSER system file at runtime, you do not have to specify a DBID and FNR value for a library.
- Ensure that you add your FUSER file in the SYSTEM library to the steplib chain. Most generated applications use the server framework components supplied with Construct in this file (prefixed with “CD” or “CC”).
- Any components required by your generated methods, such as subprograms, copycode, or data areas, must be available in your application library or one of its steplibs.
- Both the FUSER and FNAT system libraries are automatically added to your steplib chain; you do not have to add these libraries to your steplib chain.

Tip: If you are new to Construct Spectrum, set up a sample environment. For example, set up a sample application library and link it to your sample steplib chain. Use the same name to identify your application library, steplib chain, and domain.

- To access the Maintain Steplib Table panel:
 - 1 Log onto the SYSSPEC library and enter "MENU" at the Next prompt.
The Construct Spectrum Administration subsystem main menu is displayed.
 - 2 Enter "AA" in the Function field.
The Application Administration main menu is displayed.
 - 3 Enter "MM" in the Function field.
The Application Administration Maintenance menu is displayed.
 - 4 Enter "ST" in the Function field.
The Maintain Steplib Table panel is displayed:

```

BSSD_MP ***** Construct Spectrum Administration Subsystem ***** BSSD_11
Aug 31 - Maintain Steplib Table - 10:55 AM

*Action (A,B,C,D,M,N,P) _

Steplib Name.....: _____
+-----+
| Library | DB | FNR |
+-----+
| 1 | _____ | _____ |
| 2 | _____ | _____ |
| 3 | _____ | _____ |
| 4 | _____ | _____ |
| 5 | _____ | _____ |
| 6 | _____ | _____ |
| 7 | _____ | _____ |
| 8 | _____ | _____ |
+-----+

Direct Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---
confm help retrn quit flip pref main

```

Maintain Steplib Table Panel

- 5 Add up to eight application libraries to the steplib chain.
- 6 Record the name of the steplib chain.
You will add the steplib chain to the application domain described in the following section.

Step 2: Define the Domain

Domains are used to group related business objects and services. You can set up the same business object in multiple domains. The services assigned to the object can be different for each domain. For example, if you have a Customer object that is used in two applications, an Accounts Receivable and a Sales application, the Customer object in the Accounts Receivable application probably requires different services than a Customer object in a Sales application. Consider setting up two domains, one for each application. Assign services to the Customer object based on the business requirements addressed by each application.

The following steps describe how to set up a domain and link it to the steplib chain described in the previous section, **Step 1: Define the Steplib Chain**, page 43. By default, all business objects in the domain are accessed using the same steplib chain. You can, however, override the steplib chain for each business object and object method. For more information, see **Override the Steplib Chain for the Domain**, page 115.

Tip: Specify a steplib chain as high in the application architecture hierarchy as possible. This prevents you from having to specify the steplib chain in many places. If the steplib chain applies to an entire application, place it in the appropriate domain. If the steplib chain applies to one object only, identify it in the header portion of the application service definition. In this way, only exceptions need be specified.

- To access the Maintain Domains Table panel:
- 1 Log onto the SYSSPEC library and enter “MENU” at the Next prompt.
The Construct Spectrum Administration subsystem main menu is displayed.
 - 2 Enter “AA” in the Function field.
The Application Administration main menu is displayed.
 - 3 Enter “MM” in the Function field.
The Application Administration Maintenance menu is displayed.

- 4 Enter “DO” in the Function field.
The Maintain Domains Table panel is displayed:

```

BSDO__MP          Construct Spectrum Administration Subsystem      BSDO__11
Jun 27           Maintain Domains Table                          4:11 PM

Action (A,B,C,D,M,N,P)  A

Domain Name.....: SAMPLE__
Description.....: Sample Domain _____
Steplibs.....: SAMPLE_____ *

Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
confm help retrn quit      flip pref                                main
Domain SAMPLE added successfully

```

Maintain Domains Table Panel

- 5 Type “A” in the Action field.
- 6 Type the name of your domain in the Domain Name field.
- 7 Type a brief description of the domain in the Description field.
- 8 Type the name of your steplib chain in the Steplibs field.
- 9 Press Enter to add the domain.
Next, you will link the domain to user groups described in the following section.

Note: Specifying a steplib chain is optional. If no steplib is specified, the Spectrum dispatch service attempts to locate the business object from the current execution library and then from the FNAT SYSTEM library.

Step 3: Define Security for the Domain

To make your application available to users, you must grant them security privileges. To set up security, assign users to groups. Groups identify users who require similar access privileges to your application. You can then grant groups security privileges to your application domain. Granting access to a domain enables users to access the objects and methods within the domain.

Tip: You can postpone this task until after you have created and tested your application. At that time, you can better determine what security privileges should be granted to each group.

For each group granted access to a domain, you can further define security privileges by granting access to selected objects and methods. For example, assume you have an application domain called “Payroll” containing all of the objects and methods required for your organization’s payroll application. Two types of users require access to the payroll application: managers and data entry personnel. Managers require access to the entire application, while data entry personnel require access only to input hours, vacation time, sick time, and so on. You can set up one group for the managers and one for the data entry personnel. The Manager group is given access to all objects and methods in the Payroll domain and the Data Entry group is given access only to those objects and methods required to do their job.

For information about defining users and groups, see **Defining Groups and Users**, page 75, *Construct Spectrum Administration*. For information about defining security for groups and domains, see **Setting Construct Spectrum Security Options**, page 95, *Construct Spectrum Administration*.

FEATURES OF THE WIZARDS

This chapter introduces you to the Construct Spectrum Add-In tools used to build, customize, and support Spectrum web and ABO projects. These tools include the Configuration editor, which allows you to customize environmental settings, and the Regenerate Multiple function, which you can use to regenerate many modules. This chapter also describes how to use implied user exits and the `cst:PRESERVE` tag to protect and preserve custom code as you generate and regenerate modules. The remainder of this chapter explains the Report and Cache Viewer functions.

The following topics are covered:

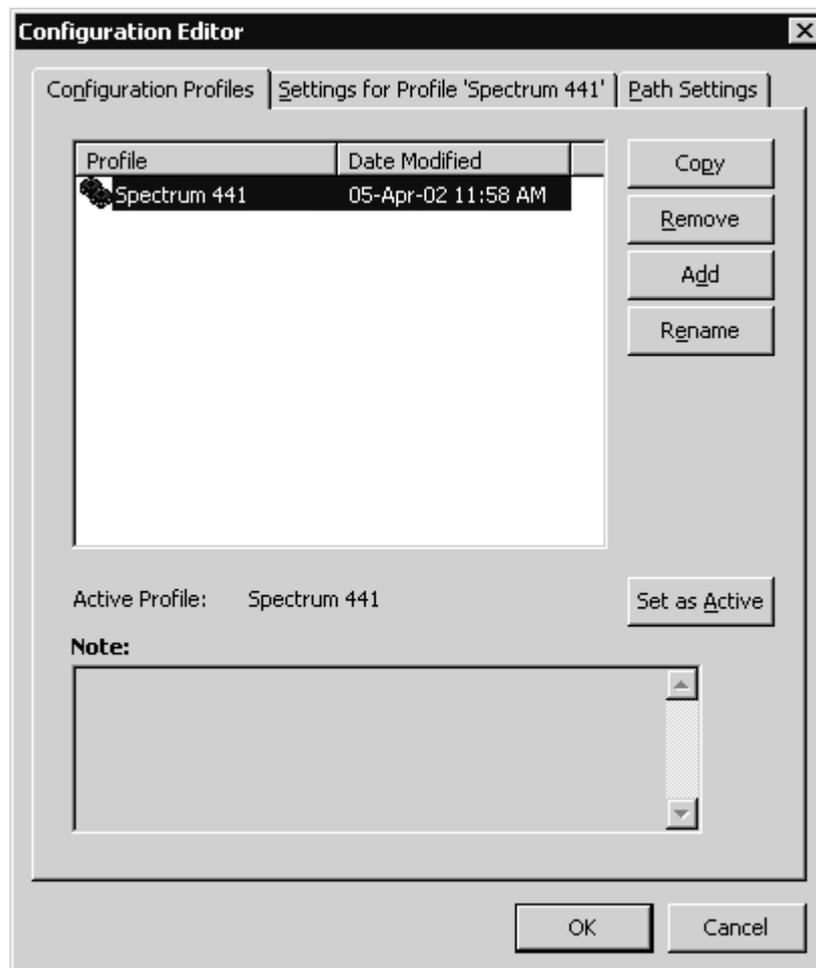
- **Using the Configuration Editor**, page 50
- **Working with Code**, page 58
- **Regenerating Modules**, page 59
- **Editing Modules**, page 62
- **Generating and Reviewing Reports**, page 63
- **Using The Spectrum Cache**, page 68

Using the Configuration Editor

The Configuration editor maintains the configuration profiles used in Construct Spectrum. Configuration profiles specify global settings, such as Spectrum services, and generate parameters in your development environment. You can set up a separate configuration profile for each environment you want to access.

Invoke the Configuration Editor

- To invoke the Configuration editor:
 - 1 Select Configuration from the Spectrum menu.
The Configuration editor is displayed, showing the Configuration Profiles tab:



Configuration Editor — Configuration Profiles Tab

This tab displays all available configuration profiles and the dates they were last modified. In the example, Spectrum 441 is the active profile.

The following options are available on the Configuration Profiles tab:

Option	Description
Copy	Makes a replica of the current profile and adds it to the Profiles list. You can then rename the profile and modify the default settings as desired. For information, see Create a New Configuration Profile , page 55.
Remove	Removes the selected profile. When you select a profile and click this button, a confirmation window is displayed. Tip: You cannot remove the active profile. You can, however, designate another profile as the active profile and then remove it. To change the active profile, select another profile and click Set as Active.
Add	Adds an unnamed profile to the Profiles list. You can then rename the profile and modify the default settings as desired. For information, see Modify the Profile Settings , page 52, and Modify the Path Settings , page 57.
Rename	Renames a profile. When you select a profile and click this button, the profile is highlighted for you to type the new name.
Set as Active	Designates the active (default) profile. When you select a profile and click this button, the selected profile becomes the active profile.

In addition to the Configuration Profiles tab, the following tabs are available:

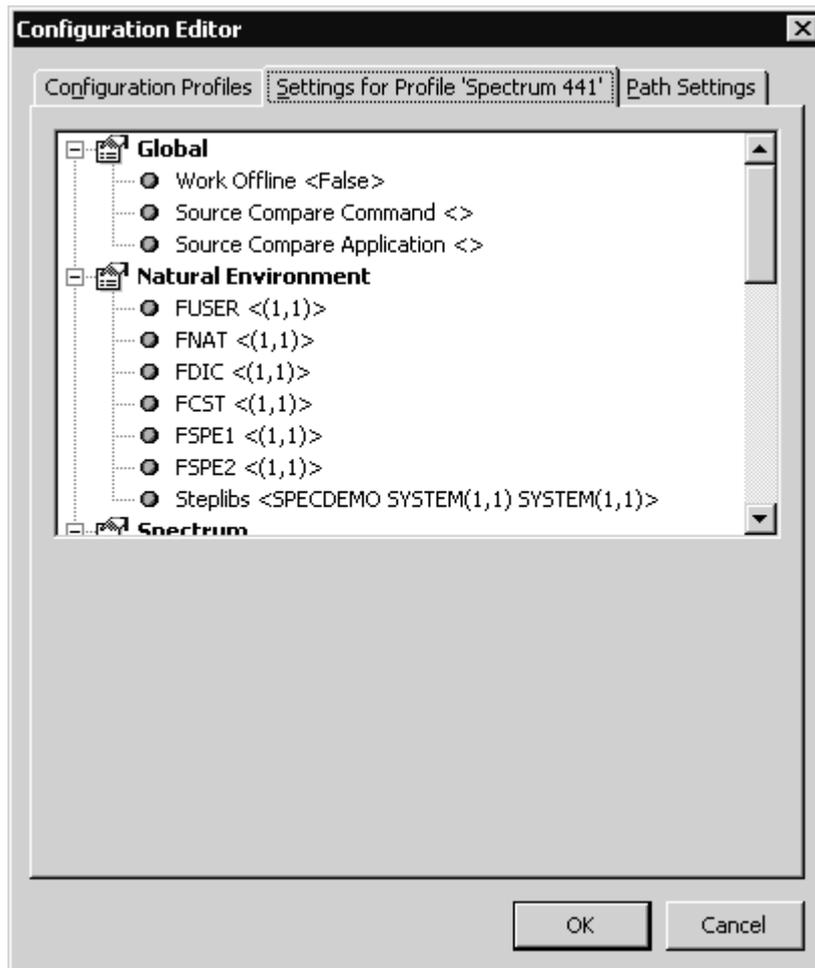
- Settings for Profile '*profile name*' tab
Select this tab to display the configuration settings for the selected profile. For information, see **Modify the Profile Settings**, page 52.
- Path Settings tab
Select this tab to display the path settings. For information, see **Modify the Path Settings**, page 57.

Modify the Profile Settings

The Settings for Profile '*profile name*' tab in the Configuration editor allows you to view and modify a variety of settings for your Spectrum ABO or web projects.

➤ To modify profile settings:

- 1 Select the profile you want to change on the Configuration Profiles tab.
- 2 Select the Settings for Profile tab.
The settings for the specified profile are displayed:



Configuration Editor — Settings for Profile 'Spectrum 441' Tab

- 3 Select the setting you want to change.
The options for that setting are displayed below the settings. For example, if you select FUSER, the current DBID and FNR values are displayed for you to modify.

- 4 Change the settings as desired.
The following settings are available:

Setting	Description
Global	
Work Offline	Indicates whether to connect to the mainframe: <ul style="list-style-type: none"> • Select True to work offline • Select False to allow calls to the server
Source Compare Command	Indicates the command used to invoke the source comparison utility.
Source Compare Application	Indicates the name of the code comparison application used. This name is displayed on the status bar of the Generation Status window when the utility is running.
Natural Environment	
FUSER FNAT FDIC FCST FSPE1 FSPE2	Lists the libraries in which Natural modules are stored. To access modules in another file, specify the DBID (database identification) and FNR (file number) for the FUSER, FNAT, Predict, Spectrum secured, and Spectrum unsecured files.
Steplib	Lists the libraries in the steplib chain. Use the direction buttons to reorder the libraries.
Spectrum	
User ID	Displays your user ID. To change your user ID, type a new ID.
Dispatcher Service	Displays the dispatch service currently used to access the mainframe server. You can: <ul style="list-style-type: none"> • Use the drop-down list to change the dispatch service. • Click Service Manager to open the Spectrum Service Manager and copy, edit, add, delete, or ping services.
Conversation Factory Service	Displays the conversation factory currently used to access the mainframe server. You can: <ul style="list-style-type: none"> • Use the drop-down list to change the service. • Click Service Manager to open the Spectrum Service Manager and copy, edit, add, delete, or ping services.
Note:	For more information, see Spectrum Service Manager , page 113, <i>Construct Spectrum Administration</i> .

Setting	Description (continued)
Wizards	
ABO Wizard	
LIF Definitions Module	Displays the default settings for the library image file (LIF) module.
Default Arbitrary Class Name	Displays the default name for generated arbitrary subprogram ABO classes.
Default Browse Class Name	Displays the default name for the generated browse ABO classes.
Default Maint. Class Name	Displays the default name for the generated maintenance ABO classes.
Code Frame	Displays the default code frame settings.
HTML Wizard	
Default Browse File	Displays the default name for the generated browse HTML templates.
Default Maint File	Displays the default name for the generated maintenance HTML templates.
Code Frame	Displays the default code frame settings.
Page Handler Wizard	
Default Class Name	Displays the default name for the generated page handler classes.
Code Frame	Displays the default code frame settings.
Object Factory	
Code Frame	Displays the default code frame settings.
Misc	Displays any notes for the selected profile.

- 5 Click OK to save the modified profile settings and close the Configuration editor.

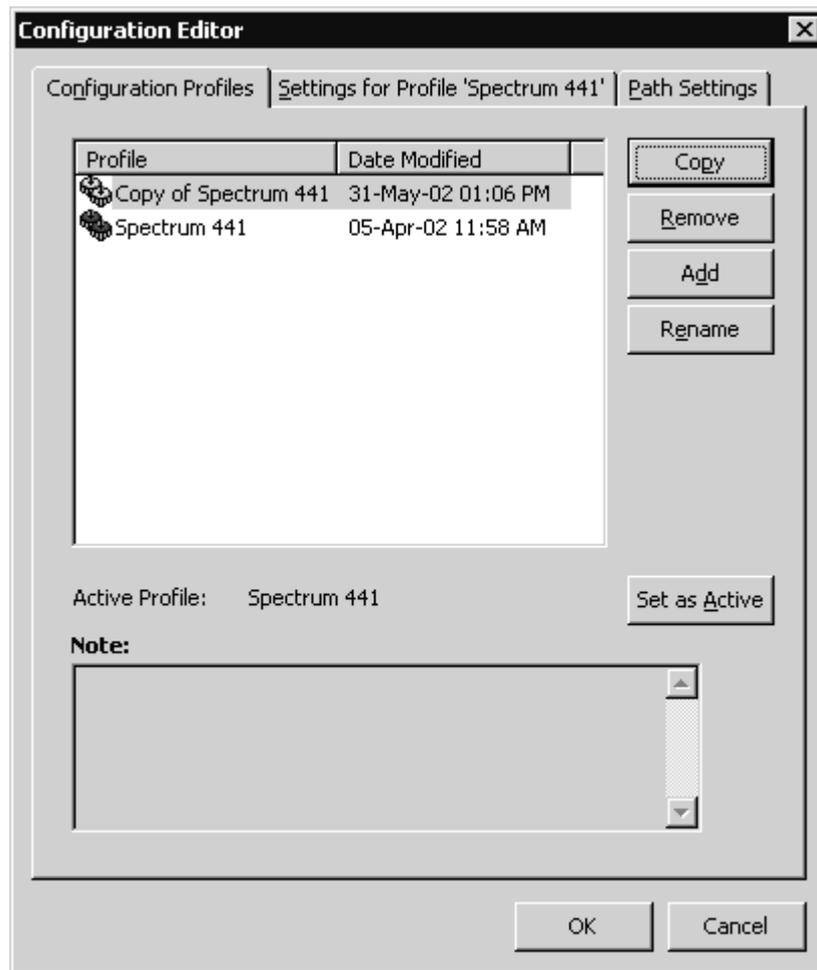
Create a New Configuration Profile

If you are creating multiple applications with different environmental settings, you may want to have multiple profiles. One method of creating a new profile is to copy an existing profile and then modify the settings.

➤ To copy a profile:

- 1 Select the profile you want to copy from the Configuration Profiles tab.
- 2 Click Copy.

A copy of the profile is added to the Configuration editor:



Configuration Editor — Copy a Profile

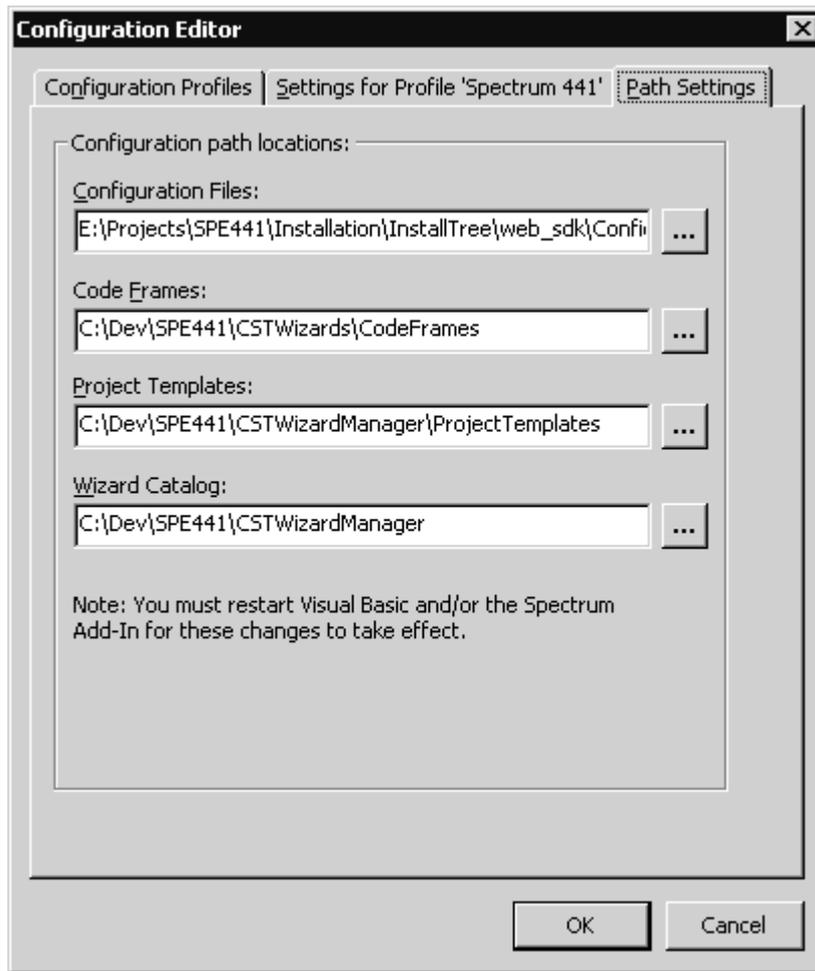
- 3 Select the profile copy and enter a new name.

- 4 Select the Settings for Profile '*profile name*' tab or the Path Settings tab to specify configuration settings for the new profile.
For information, see **Modify the Profile Settings**, page 52, and **Modify the Path Settings**, page 57.
- 5 Click OK to save the profile.

Modify the Path Settings

The Configuration editor allows you to view and modify the path settings for your Spectrum web or ABO projects.

- To modify path settings:
 - 1 Select the profile you want to change on the Configuration Profiles tab.
 - 2 Select the Path Settings tab.
The path settings for the specified profile are displayed:



Configuration Editor — Path Settings Tab

- 3 Use the browse buttons to change the path settings for the Configuration Files, Code Frames, Project Templates, or Wizard Catalog locations.

For changes to take effect, restart Visual Basic or the Spectrum Add-In.

Working with Code

The following sections describe how to use implied user exits and the `cst:PRESERVE` tag to protect and preserve custom code when regenerating modules.

Implied User Exits

Implied user exits act as placeholders for user exits coded after generating a module, ensuring the user exits are placed in the source code in exactly the same order as before generation. However, implied user exits are not added to the generated code unless you have coded them first. Consequently, you must add an exit line to preserve any hand-coded changes during regeneration.

Implied user exits are easily recognized because they use a standard structure and naming convention. Their names are prefixed with the name of the function, subroutine or property you are coding, and suffixed with a location (start or end). For example, you can add two user exits to a function called `Initialize`: `Initialize.Start` and `Initialize.End`. The property procedures also indicate the type of property in the name. For example, a `Property Get` exit for `CustomerNumber` is:

```
'CustomerNumber.Get.Start'
```

All subroutines have implied user exits at the beginning and end of the routine. For example:

```
Private Sub PerformAction()  
    '<cst:EXIT Name='PerformAction.Start' Implied=True>  
    Dim sval as String  
  
    sval = LookupAction()  
  
    '<cst:EXIT Name='PerformAction.End' Implied=True>  
End Sub
```

Preserve Customizations to Generated Code

Use the `cst:PRESERVE` tag to protect your custom code during a regenerate. Place the tag before and after whole subroutines, entity groups, or between individual variables. For example, to preserve code in the `Class_Initialize` subroutine, add the following code:

```
'<cst:PRESERVE>  
Private Class_Initialize()  
    Set m_ABO = CreateObject(PROG_ID)  
End Sub  
'</cst:PRESERVE>
```

Regenerating Modules

This section describes how to regenerate individual and multiple modules. The regeneration process is performed in the background (without your input).

Regenerate Individual Modules

There are two ways to regenerate individual modules.

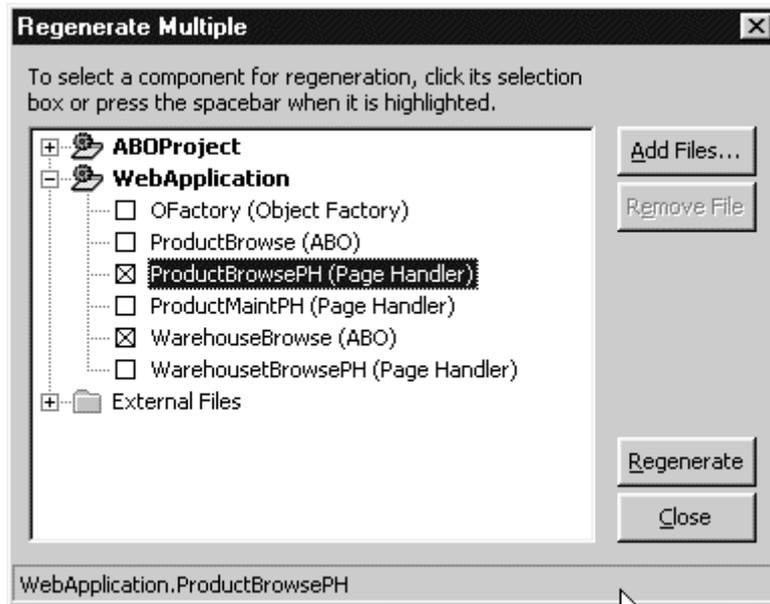
- To regenerate a single module:
 - 1 Do one of the following:
 - Right-click the module in the Project Explorer and select Regenerate from the short-cut menu.
 - Select the module in the Project Explorer and select Regenerate <module name> from the Spectrum menu.

Note: You can regenerate individual modules by clicking Edit <module name> with <Wizard> from the Spectrum menu. This invokes the wizard used to generate a module, allowing you to edit the specifications and regenerate the module.

For information about editing modules, see **Editing Modules**, page 62.

Regenerate Multiple Modules

- To regenerate several modules simultaneously:
 - 1 Open your project in Visual Basic.
 - 2 Select Spectrum > Regenerate Multiple.
The Regenerate Multiple window is displayed:



Regenerate Multiple Window

This window contains both your project modules and any external files you added to your projects. If you do not have any modules in a project or folder, an icon is displayed.

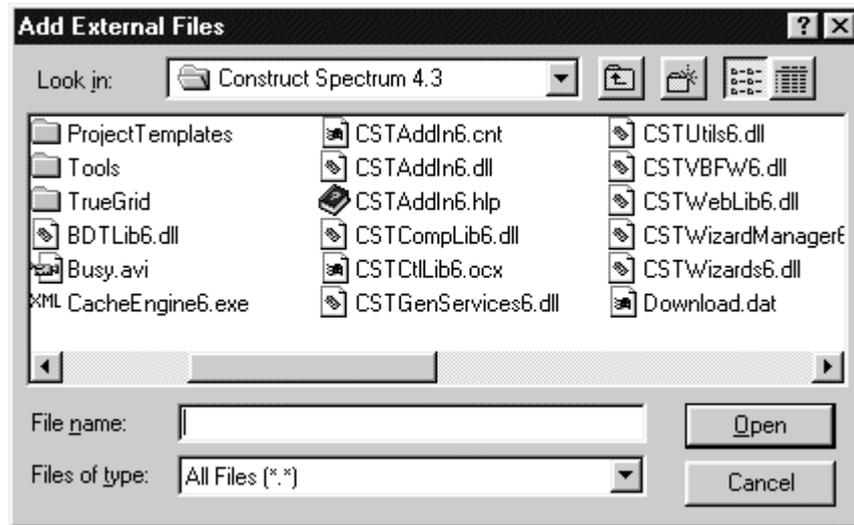
In this window, you can:

- Click Add Files to move external files into your project for regeneration. For information, see **Regenerate External Files**, page 61.
 - Click Remove Files to move files you do not want to regenerate from this window.
 - Expand the tree to display the individual project modules you want and any external files you want to regenerate.
- 3 Select the modules you want to regenerate and click Regenerate.
A message indicates when the regeneration is complete.

Regenerate External Files

You can also use the Regenerate Multiple window to add external files you want to regenerate.

- To add external files:
 - 1 Click Add Files in the Regenerate Multiple window.
The Add External Files window is displayed:



Add External Files Window

- 2 Select the file(s) you want to regenerate and click Open.
The files are added to the Regenerate Multiple window.

Editing Modules

- To edit a module, either:
- 1 Select the module you want to modify in the Project Explorer.
 - 2 Select Edit <module name> with <Wizard> from the Spectrum menu.
The wizard used to generate the module opens.
 - 3 Modify the model specifications and regenerate it.
- or
- 1 Right-click the module in the Project Explorer.
 - 2 Select Edit with <Wizard> from the shortcut menu.
The wizard used to generate the module opens.
 - 3 Modify the model specifications and regenerate it.

For more information about the ABO wizard, see **Using the ABO Wizard**, page 92.

For more information about the HTML Template wizard, see **Creating and Customizing an HTML Template**, page 91, *Construct Spectrum SDK for Web Applications*.

For more information about the Object Factory wizard, see **Updating and Customizing the Object Factory**, page 137, *Construct Spectrum SDK for Web Applications*.

For more information about the Page Handler wizard, see **Creating and Customizing a Page Handler**, page 75, *Construct Spectrum SDK for Web Applications*.

Generating and Reviewing Reports

This section describes how to generate reports as you generate modules, and how you can review these stored reports as you edit and regenerate modules. It also describes how to use a code comparison tool to further determine the differences between initial and regenerated code.

A report is generated every time you use a wizard to generate an ABO or web component. These reports are also stored, allowing you to review the generation process while editing and regenerating modules. If you have a code comparison utility configured to work with Construct Spectrum, you can invoke it from the Report window to examine code differences between initial and regenerated modules. For more information, see **Use Reports with a Code Comparison Tool**, page 67.

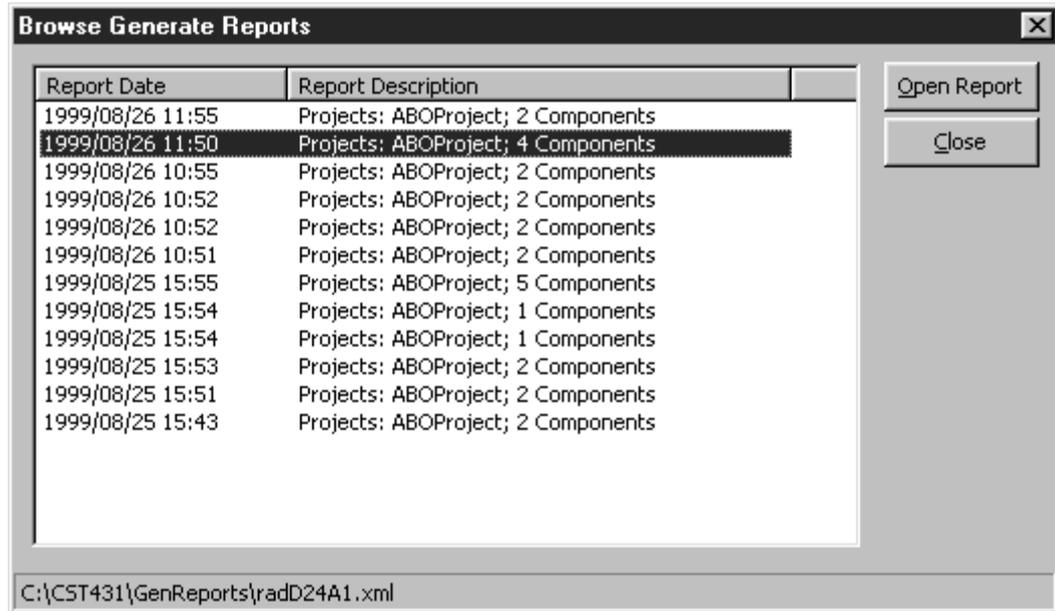
Access Reports

There are two ways to access reports:

- **As you generate.**
Clicking **Generate** in the last window of a wizard generates both the module and a report that details the specific actions that occurred during the generation process. This report is automatically stored, allowing you to review it as you modify and regenerate modules before saving them in your Visual Basic project. For information, see **Review a Stored Report**, page 64.
- **By reviewing stored reports.**
It may be useful to review stored reports as you edit and regenerate your application components using the **Generate Report** window. You can also invoke your code comparison tool from this window to determine the differences between initial and regenerated code. For information, see **Use Reports with a Code Comparison Tool**, page 67.

Review a Stored Report

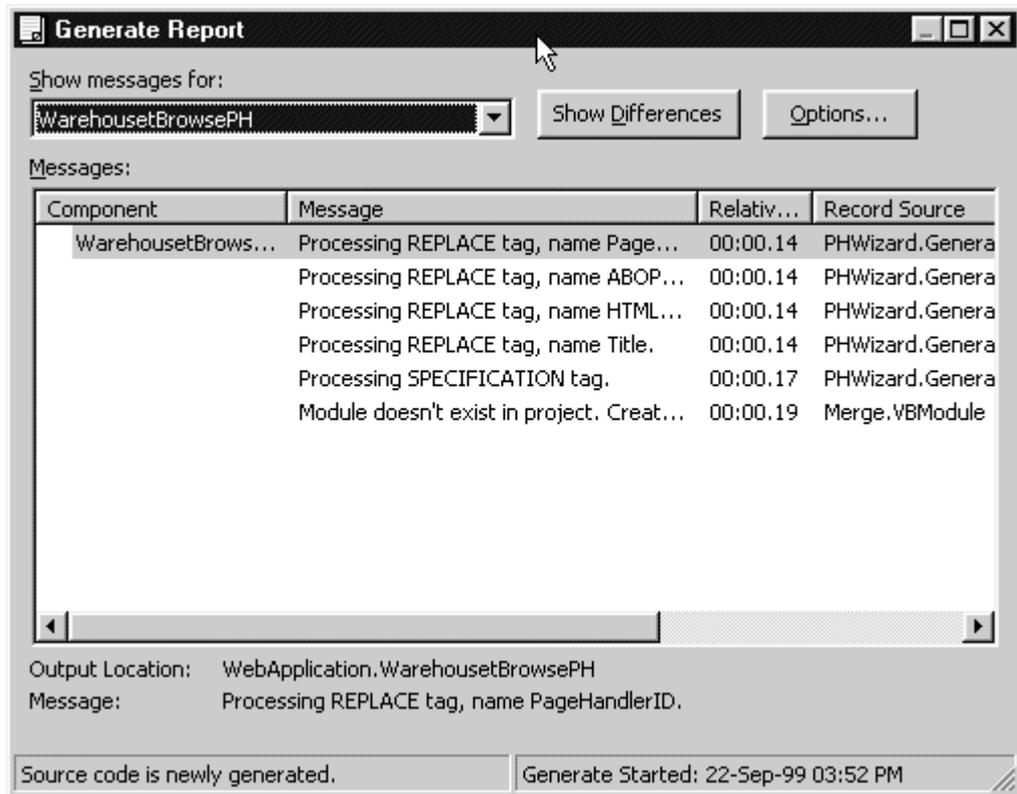
- To review a stored report:
- 1 Select Reports from the Spectrum menu.
The Browse Generated Reports window is displayed:



Browse Generated Reports Window

This window provides the generation date and the description of saved reports. This includes listing the names of projects in the report and the number of components that were included in the report.

- 2 Select the report you want to open and click Open Report.
The Generate Report Window is displayed:



Generate Report Window

This window displays any items that were added, removed, or changed, not only in the current module, but also in any other modules affected by the generation, such as LIF definitions. It also displays the location of the component, the time the generate was initiated, and any messages.

Use the Show messages for drop-down list to select other components. The following table outlines the components for which you can view messages:

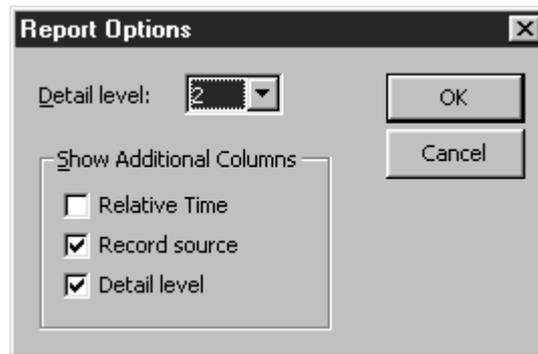
Component	Displays
All components	Messages for the generation status of all module components.
System	Generic system process messages that are unrelated to a specific component.
<Specific component>	Messages for the generation of a specific component.

Note: All components may not be displayed as they may not be included in all generated modules.

Specify Report Detail

You can specify the level of detail you want to see in your report and select other report options and requirements.

- To specify report options:
- 1 Click Options in the Generate Report window.
The Report Options window is displayed:



Report Options Window

- 2 Select the options you want.
For example, you can:
 - Use the Detail level drop-down list to select the level of detail you want the report to display. Level 1 provides a very high level summary, whereas Level 4 provides a highly detailed report.
 - Select Relative Time to display exactly when various stages of the generation process occurred.
 - Select Record source to view the source of each message.
- 3 Click OK to return to the Generate Report window.
The window contains information in response to all of the options you specified.

Use Reports with a Code Comparison Tool

If you have a code comparison tool that is configured to work with Construct Spectrum, you can click Show Differences in the Generate Report window to view differences between an original and regenerated file. If you do not have a code comparison tool installed or it is not properly configured to work with Construct Spectrum, the View Differences button is disabled.

Use the Configuration editor to configure your code comparison tool with Construct Spectrum.

- To configure the code comparison tool:
 - 1 Open the Configuration editor.
 - 2 Select the Settings for Profile '*profile name*' tab.
 - 3 Supply a command in Source Compare Command.

For example:

```
"C:\Program Files\BeyondCompare\beyond32.exe" "%1" "%2" /noedit
```

- 4 Click OK.
To launch the comparison utility, click Show Differences in the Reports Generate window.

Using The Spectrum Cache

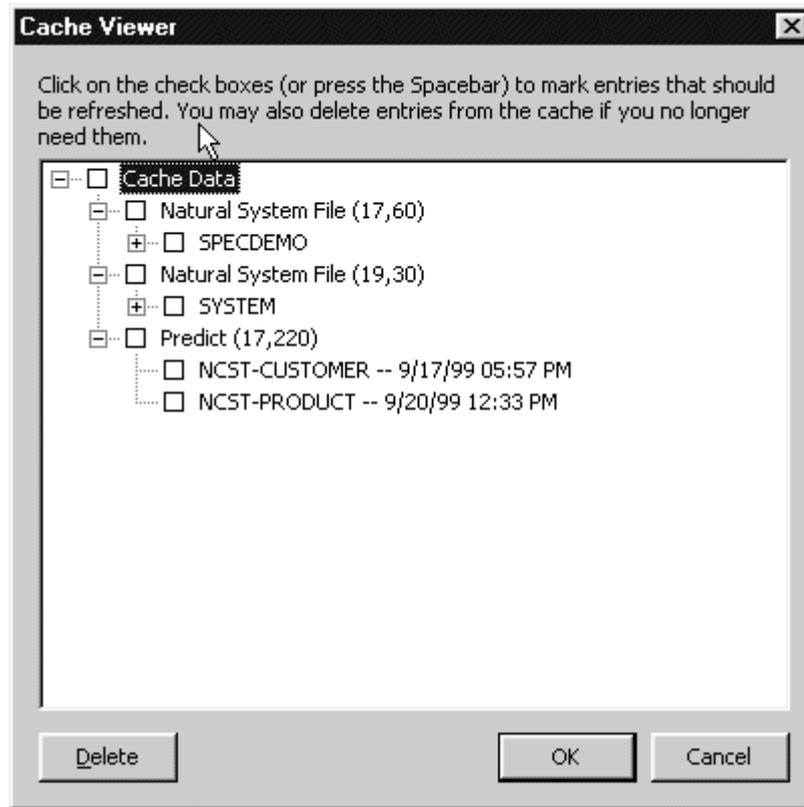
The Spectrum Cache is a dynamic, hierarchical data structure that stores data returned from the server. The cache allows you to quickly store and access values that are used frequently but that take a long time to retrieve or derive. This section describes how to use the Cache Viewer.

Overview

The hierarchical tree structure of the Spectrum cache means it can store complex values such as Predict file definitions. The cache contains all the data used by the wizards during the generate process. It also contains information extracted from FUSER modules and information about the generation environment. Use the Cache Viewer to display data in the cache, mark nodes to be refreshed, or remove nodes to clean up the cache.

- To invoke the Cache Viewer:
 - 1 Do one of the following:
 - Select View Cache from the Spectrum menu.
 - Click the Cache Viewer icon on any wizard.

The Cache Viewer window is displayed:



Cache Viewer Window

The Cache Viewer displays a hierarchical structure of the system files, libraries, nodes, and Predict views in your application. The lowest nodes are followed by the date they were last refreshed.

Mark Nodes to be Refreshed

Use the Cache Viewer to select the nodes you want refreshed.

- To mark nodes to be refreshed:
 - 1 Invoke the Cache Viewer.
 - 2 Expand the tree to view individual nodes.
 - 3 Select the node(s) you want to have refreshed.
 - 4 Click OK.

When you mark a node, you also mark all of its children. When the wizards fetch data from the cache, they recognize the nodes you specified to be refreshed and make the appropriate call to the server. If the server call fails, existing data in the cache is used.

Remove Nodes From the Spectrum Cache

You can remove nodes that are no longer needed to clean up the cache.

- To remove nodes:
 - 1 Invoke the Cache Viewer.
 - 2 Expand the node tree and select the node(s) you want to delete.
 - 3 Click Delete.
The nodes are removed from the cache.

USING THE BUSINESS-OBJECT-SUPER-MODEL

This chapter describes how to use the Business-Object-Super-Model to generate multiple Natural components of a Construct Spectrum web or client/server application — without using the Construct Spectrum client framework components.

The following topics are covered:

- **Overview**, page 72
- **Before You Begin**, page 73
- **Generating Packages**, page 76
- **Troubleshooting**, page 83

Overview

The Business-Object-Super-Model uses a single, high-level model specification to generate all the required Natural components (modules) of a web or other distributable application. This model generates sets of modules, called “packages”, for all the business objects in an application, such as the object maintenance and browse subprograms, proxies, and parameter data areas (PDA) for a Customer business object.

Typically, you will use the Business-Object-Super-Model to generate the first iteration of your application. To generate these components, the super model executes the individual model for each module. As you refine the application, you will likely regenerate certain modules separately using the individual models. The following table lists each module in a typical package and the model used to generate it:

Module	Model Name	Description
Object maintenance subprogram, Object PDA, Restricted PDA	Object-Maint-Subp	Subprogram used to maintain a business object. This model also generates the parameter data area and restricted PDA for the object.
Object maintenance subprogram proxy	Subprogram-Proxy	Proxy used to communicate information between the Spectrum dispatch service and an object maintenance subprogram.
Object browse subprogram, Key PDA, Row PDA, Restricted PDA	Object-Browse-Subp	Subprogram used to encapsulate access to data on the server and return records in rows and columns, and the PDAs that communicate information to and from the subprogram.
Object browse subprogram proxy	Subprogram-Proxy	Proxy used to communicate information between the Spectrum dispatch service and an object browse subprogram.

You can also generate modules that allow users to browse business objects within a package or linked through a foreign field relationship.

Tip: Although the super model does not support user exits, you can specify a user exit by regenerating the Natural module using its individual model.

Before You Begin

Before generating application packages for a Construct Spectrum application, there are several prerequisite tasks you must perform. Before completing any of these tasks, ensure that all required software has been installed and configured on both the server and the client.

Before you begin:

- 1 **Check the Model Defaults**, page 73
- 2 **Set up Default Values in Predict**, page 73
- 3 **Establish a Naming Convention**, page 74
- 4 **Set Up the Application Environment**, page 75

These tasks are described in the following sections.

Check the Model Defaults

When the super model invokes individual models, it uses the default values specified for each model. Review and change (if necessary) the current defaults for these models. To review the values, invoke each model used by the Business-Object-Super-Model and note the default values.

Model	Defaults
Object-Browse-Subp	Uses the first four characters of the module name to suffix the object, key, and restricted PDAs.
Object-Maint-Subp	Uses the first four characters of the module name to suffix the object and restricted PDAs.

Set up Default Values in Predict

The Business-Object-Super-Model generates specifications for all of the models used to generate an application from a small set of input parameters. To accomplish this, it relies heavily on parameter defaulting. You can add keywords to your file and field definitions in Predict to default various parameters. Customize parameter defaults by linking Predict keywords and verification rules to your fields, files, and relationships.

For more information about Predict defaults and definitions, see **Setting Up Predict Definitions**, page 39.

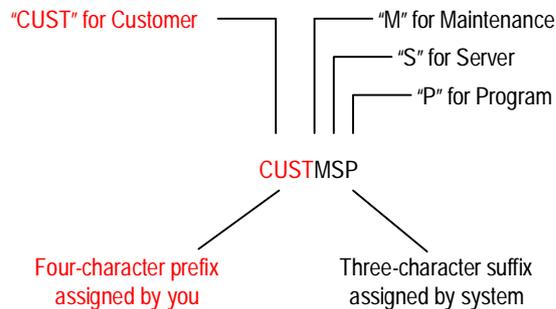
Establish a Naming Convention

Because the Business-Object-Super-Model generates multiple modules, it is important to establish a naming convention. Locating the modules is easier when your naming convention clearly identifies them.

When using the Business-Object-Super-Model, you must supply a four-character prefix to be used for all modules within a package. (If you specify a prefix that is less than four characters, it is padded with dashes.) The super model defaults the suffix, which identifies the module type, as follows:

Suffix	Module
MSO	Object maintenance subprogram
MSA	object PDA
MSR	restricted PDA
MSP	Proxy for the object maintenance subprogram
BSO	Object browse subprogram
BROW	row PDA
BKEY	key PDA
BPRI	restricted PDA
BSP	Proxy for the object browse subprogram

The following example illustrates the naming conventions for a generated module:



Naming Conventions for a Generated Module

Set Up the Application Environment

Before creating a Construct Spectrum application, you must set up and configure the mainframe environment for your application as follows:

- 1 Define the steplib chain.
- 2 Define the domain.
- 3 Define security for the domain.

For more information, see **Setting up the Mainframe Environment**, page 37.

Generating Packages

You can use the Business-Object-Super-Model in either the Construct Windows interface or the Construct Generation subsystem. The parameter information you are asked to specify is the same in both interfaces and there are the same number of input specification steps. In the Generation subsystem, there are three specification panels: Standard Parameters, Package Parameters, and Specific Package Parameters. Similarly, in the Construct Windows interface, there are three steps in which you can specify standard parameters, packages parameters, and new package parameters.

The following sections describe how to use the Business-Object-Super-Model to create application packages. Examples are from the Construct Windows interface.

- To generate application packages using the Business-Object-Super-Model:
 - ❑ **Step 1: Define the Standard Parameters**, page 77
 - ❑ **Step 2: Define the General Package Parameters**, page 78
 - ❑ **Step 3: Define the Specific Package Parameters**, page 79
 - ❑ **Step 4: Create Another Package (Optional)**, page 81
 - ❑ **Step 5: Generate the Modules**, page 81

Note: For information about invoking the super model, see **Using the Generation Subsystem**, page 59, *Natural Construct Generation*, and **Generating with the Super Model**, page 86, *Construct Spectrum SDK for Client/Server Applications*.

Step 1: Define the Standard Parameters

The following example shows the standard parameters for the Business-Object-Super-Model:

The screenshot shows the 'BUSINESS-OBJECT-SUPER-MODEL Wizard' dialog box. On the left, a vertical navigation pane has five steps: 'Start', 'Standard Parameters' (which is selected and highlighted in black), 'Packages', 'New package', and 'Finish'. The main area is titled 'Standard Parameters' and contains the following fields and controls:

- Module:** A text box containing 'EHCU' with a tooltip '(SPECDEMO on 1000,1002)'.
- System:** A text box containing 'SPECDEMO'.
- Title:** A text box containing 'Super Spec for my module'.
- Description:** A text area containing 'Specification'.
- Message numbers:** An unchecked checkbox.

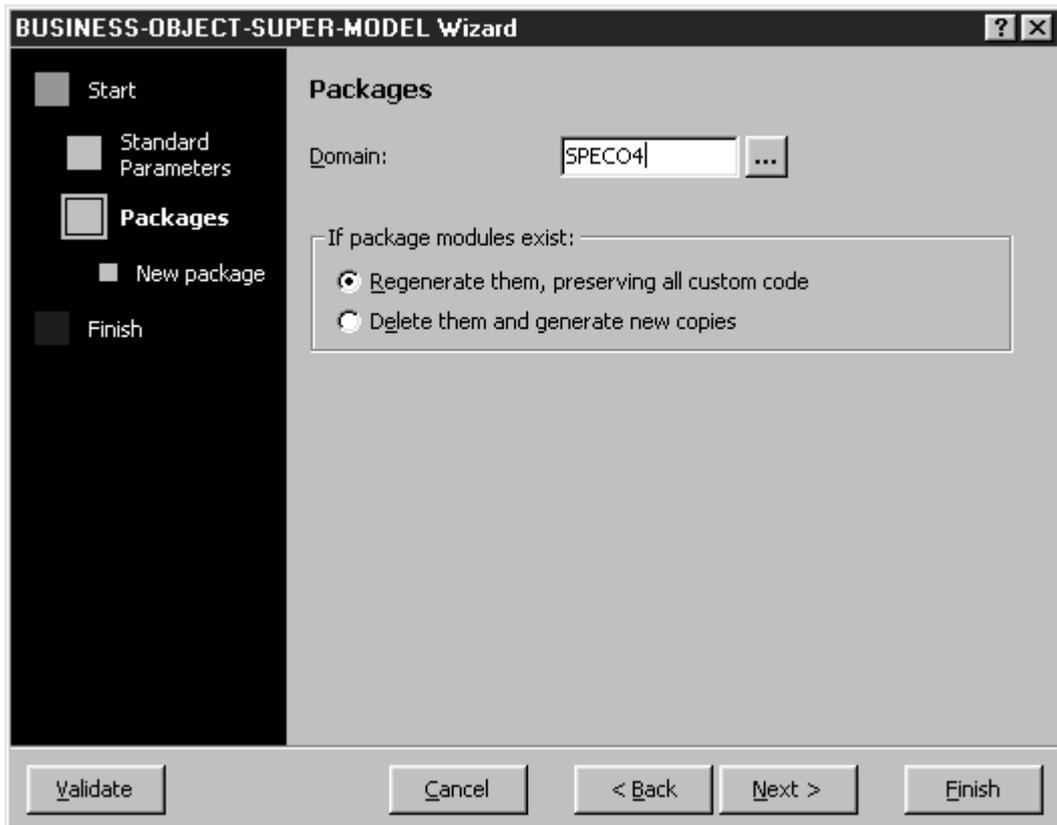
At the bottom of the dialog, there are five buttons: 'Validate', 'Cancel', '< Back', 'Next >', and 'Finish'.

Business-Object-Super-Model Wizard — Standard Parameters

- To define standard parameters for the package:
 - 1 Type a name for the super model specifications in Module.
This name identifies the package you are about to create. The name should be descriptive so that you can easily identify the package later.
 - 2 Type the name or identification number of the library where you want to generate the modules in System.
By default the name of the current library is displayed.
 - 3 Type a brief title for the package in Title.
 - 4 Type a brief description of the package in Description.
 - 5 Click Next.
The general package parameters are displayed.

Step 2: Define the General Package Parameters

Next, specify the name of the package for which you want to generate modules, as well as some basic package criteria that affect the entire application:



Business-Object-Super-Model Wizard — General Package Parameters

- To define general package parameters:
 - 1 Provide the domain name for this application in Domain.
To display a list of domains for selection, click the Browse button.
 - 2 Do one of the following:
 - Mark Regenerate them, preserving all custom code (the default) to regenerate existing modules and save all custom code. Any modified parameters in the specification are not used. However, the super model will keep user exits and apply updates from Predict (such as a new field or BDT keyword) and from the model code frames.
 - Mark Delete it and generate a new copy to replace all existing modules.
 - 3 Click Next.
The specific package parameters are displayed.

Step 3: Define the Specific Package Parameters

All packages in your application are displayed on the navigation bar. To navigate between packages, click Next, Back, or select a package from the navigation bar:

BUSINESS-OBJECT-SUPER-MODEL Wizard

Start
Standard Parameters
Packages
 NCST-CUSTOMER
Finish

Package prefix: EHCU

PREDICT view: NCST-CUSTOMER ... Defaults

Primary key: ...

Hold field: ...

Description: Add Delete

Package modules:

Module	Gen.	Model	G/R/O	Library
EHCUMSO	<input type="checkbox"/>	Object Maintenance Subprogram		? ?
EHCUMSP	<input type="checkbox"/>	Spectrum Maintenance Proxy		? ?
EHCUBSO	<input type="checkbox"/>	Object Browse Subprogram		? ?
EHCUBSP	<input type="checkbox"/>	Spectrum Browse Proxy		? ?

Check

Validate Cancel < Back Next > Finish

Business-Object-Super-Model Wizard — Specific Package Parameters

- To define package parameters:
 - 1 Specify a prefix for the package in Package prefix.
For more information, see **Establish a Naming Convention**, page 74.
 - 2 Specify the view used by the browse and maintenance subprograms in Predict view.
This view determines which business object will be used. Click Defaults to retrieve the defaults for the object.
 - 3 Specify the primary key for the specified view in Primary key.
The key can be a descriptor, superdescriptor, or subdescriptor. If the key does not exist in the corresponding Predict file, an error message is displayed upon validation. This value cannot be the same as that in the Hold field.
 - 4 Specify the name of the field used to logically protect the record against intervening Update or Delete actions in Hold.

- 5 Type a brief description of your package file in Description.

Tip: Based on how the file is defined in Predict, the super model attempts to provide default field values. You can override the defaults using Predict keywords. Rather than typing the values directly, set up your Predict file definition to default the required values. For information, see **Setting Up Predict Definitions**, page 39.

- 6 Select the package modules you want to generate from Package modules. To select all the modules, right-click and select Select All Modules from the shortcut menu. The following information is displayed in Package modules:
 - Module lists all modules that can be generated by the super model. Each module is identified by the package prefix, followed by the standard suffix for the module type. For more information, see **Establish a Naming Convention**, page 74.
 - Gen indicates which modules will be generated.
 - Model indicates the names of the individual models that the super model invokes to generate the package modules.
 - G/R/O indicates one of the following:
 - G Module does not currently exist in source form and will be generated and saved in the current library.
 - R Module currently exists in source form and will be regenerated and saved in the current library. This status occurs when you select Regenerate it, preserving custom code in Step 2.
 - O Module currently exists in source form and will be overwritten and saved in the current library. This status occurs when you select Delete it, and generate a new copy in Step 2.
 - Library indicates one of the following for each module:
 - ? Click Check to determine if there is existing source or object code.
Blank indicates that a check was made, but there is no existing code.
 - S Indicates that source code exists. If the S is black, the source code is in the current library. If the S is red, the source code is in another library. To view the location of the source code, place the mouse pointer over the S.
 - C Indicates that compiled (object) code exists. If the C is black, the source code is in the current library. If the C is red, the source code is in another library. To view the location of the source code, place the mouse pointer over the C.

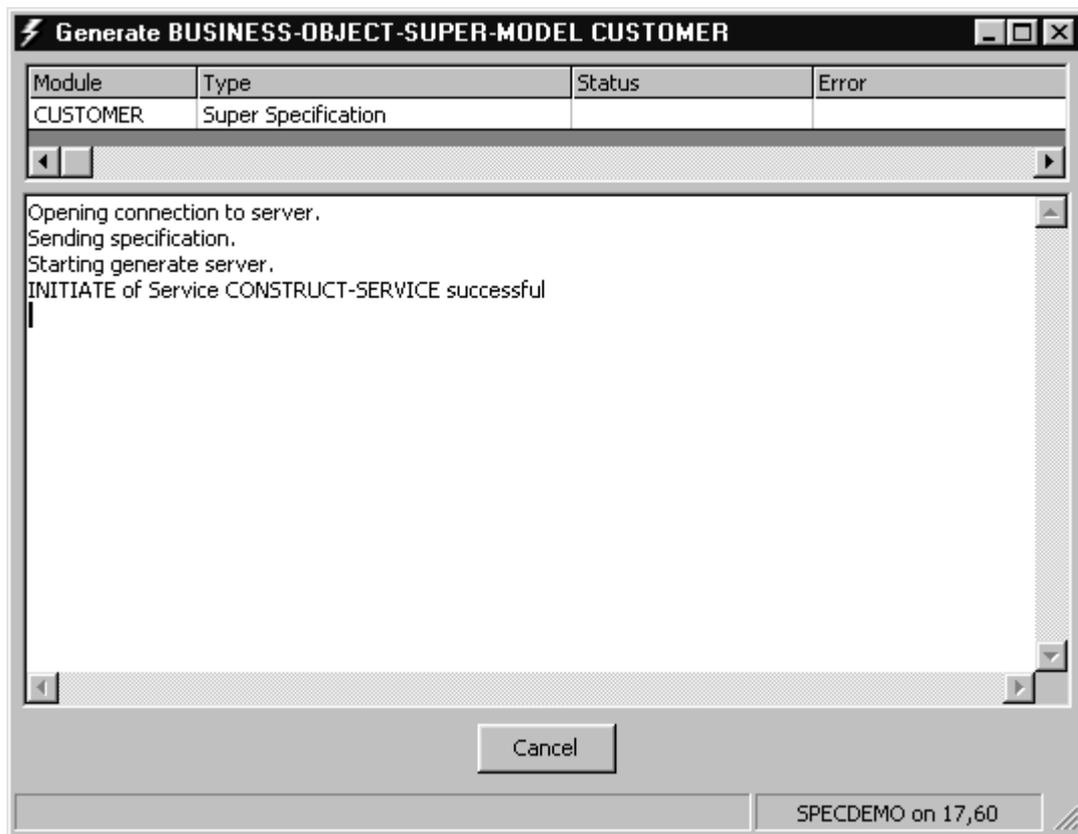
Step 4: Create Another Package (Optional)

You can define the parameters for up to 12 packages.

- To create another package:
 - 1 Click Next or Add.
 - 2 Complete each additional package as described in previous steps.

Step 5: Generate the Modules

- To generate the modules:
 - 1 Click Finish.
The Code window is displayed.
 - 2 Select File > Generate or click the Generate icon on the toolbar.
The Generate window is displayed, showing the generation process:



Generate Window

The module status pane displays the names of the modules as they are generated and stowed by the Business-Object-Super-Model. The messages pane provides a status report of the generation process, including any error messages that may occur. When all modules have been generated and stowed, a confirmation message is displayed.

Note: Click Cancel to terminate the generation process at any time.

Generation Subsystem

In the Generation subsystem, you can either generate in batch or generate from the main menu. (Generation is automatically done in batch in the Construct Windows interface.)

Tip: If you are generating multiple modules, generate in batch to avoid tying up system resources.

- To generate the modules from the Generation main menu:

Note: If the super model specification is not currently in the Natural Construct edit buffer, read it into Natural Construct and proceed.

- 1 Enter “G” in the Function field.
The Business-Object-Super-Model specification is saved and the specifications for the individual modules are created and saved.
- To generate modules in batch:
- 1 Save the super model specification on the Natural Construct Generation main menu.
 - 2 Invoke the NCSTBGEN utility.
For information about this utility, see **Multiple Generation Utility**, page 753, *Natural Construct Generation*.
 - 3 Specify the name under which you saved your super model specification and the model name: Business-Object-Super-Model.
 - 4 Generate the modules.

Troubleshooting

After using the Business-Object-Super-Model to generate the modules in a package, review the generation status report to reconcile any errors that may have occurred. The following table lists possible errors and solutions:

Error	Solution
A module was generated, but not stowed because of a missing DDM.	Correct the error and regenerate the module using its model specification.
A generation error occurred because of a missing dependent module.	Correct the error and regenerate the module using its model specification.
Generation errors affected several modules.	Correct the errors and regenerate the modules as follows: <ol style="list-style-type: none">1 Re-read the super model specification into Construct Spectrum.2 Mark the modules that require regeneration.3 Repeat the generation steps until all modules have been successfully generated and stowed.
Compilation errors in the super model-generated code caused cycling.	Ensure that the SYNERR parameter is set to ON in your user profile NATPARM.

USING ACTIVEX BUSINESS OBJECTS

This chapter describes ActiveX business objects (ABOs). It contains step-by-step instructions to use the ABO Project wizard and ABO wizard. It also describes how to customize your generated ABOs.

The following topics are covered:

- **Overview**, page 86
- **Using the ABO Project Wizard**, page 87
- **Using the ABO Wizard**, page 92
- **Customizing the ABO**, page 98

Overview

An ABO is a Visual Basic class. This class wraps the Spectrum calls required to communicate with a Natural subprogram exposed by a subprogram proxy. It exposes a set of interfaces that provide a consistent and familiar interface to Natural components and make the subprogram easy to use. You must generate an ABO for each of the Natural subprograms used in your application.

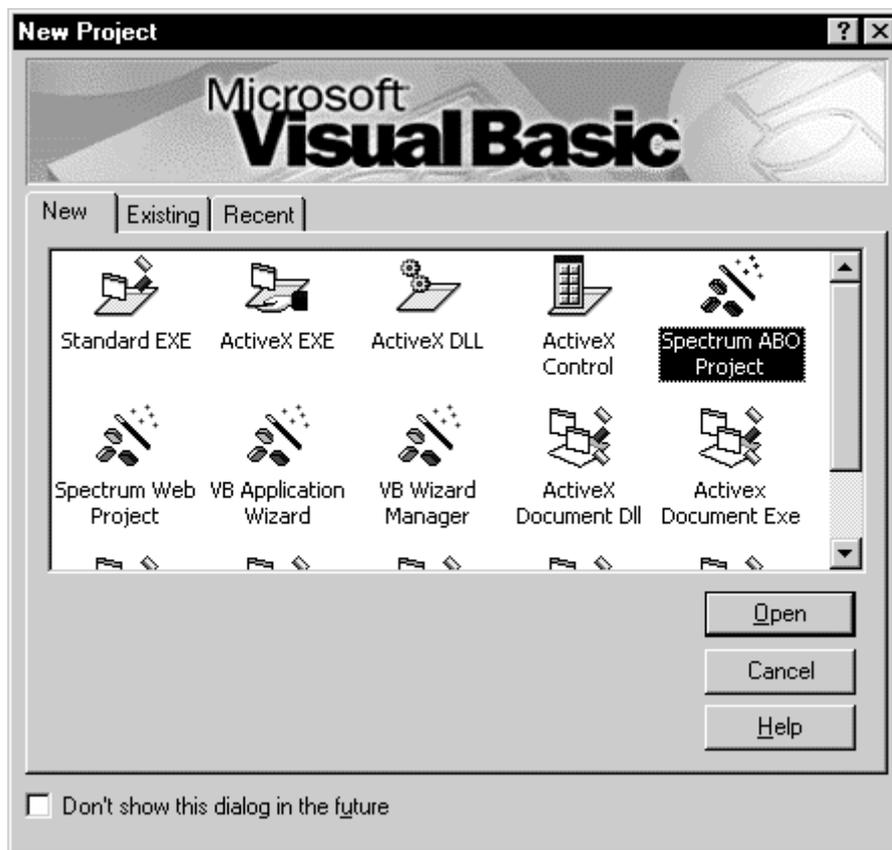
An ABO project contains the ABOs used by your application, as well as the framework components supplied by Construct Spectrum. Use the ABO Project wizard to generate your ABO project and then use the ActiveX Business Object wizard to generate the ABOs.

Using the ABO Project Wizard

This section describes how to use the ABO Project wizard to create an ABO project, as well as the framework components Construct Spectrum adds to the project.

Create the ABO Project

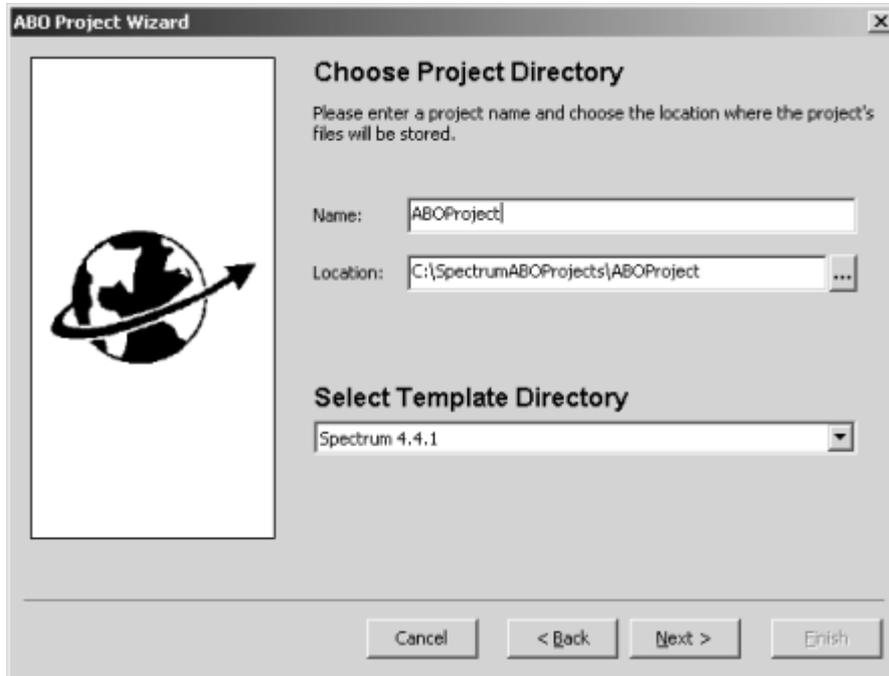
- To create the ABO project:
 - 1 Start Visual Basic.
The New Project window is displayed:



New Project Window

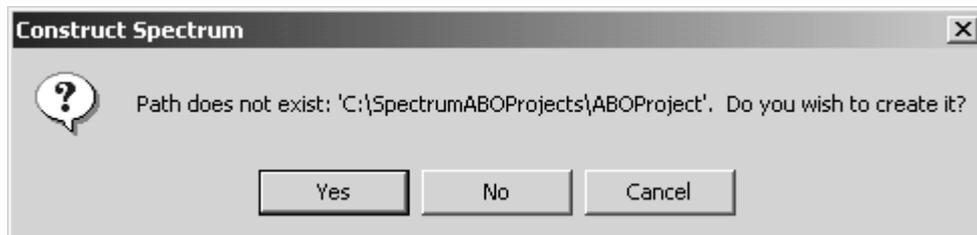
- 2 Select Spectrum ABO Project.
- 3 Click Open.
The ABO Project wizard is displayed.

- Click Next.
The Choose Project Directory window is displayed:



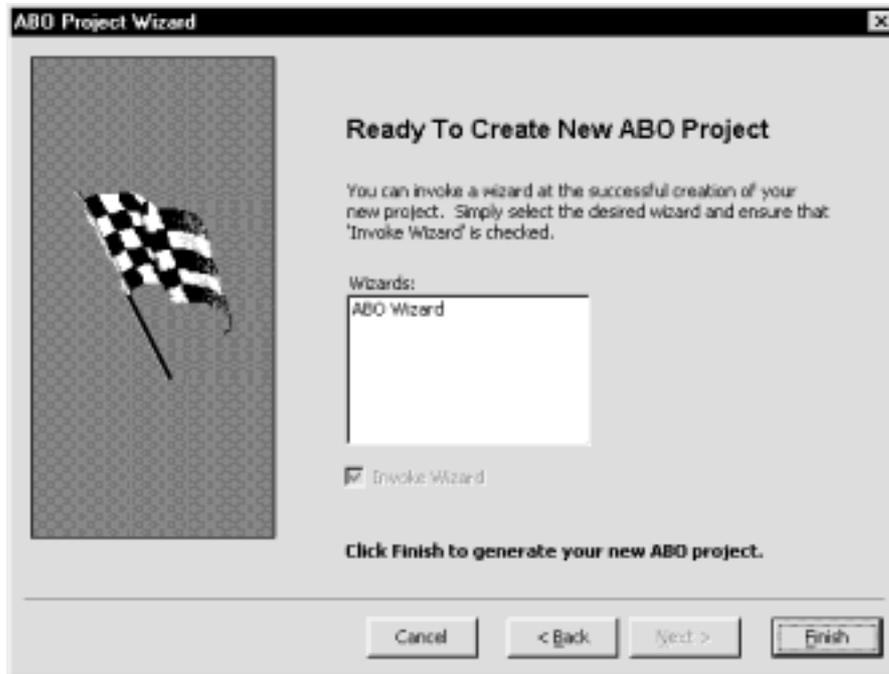
ABO Project Wizard — Choose Project Directory

- Enter your project name and the location to store your project.
Store the project in the same directory in which your web applications are stored.
- Click Next.
If you keep the default directory or specify a directory that does not exist, the following window is displayed:



Create New Directory

- 7 Click Yes to create the new directory.
The Ready to Create New ABO Project window is displayed:

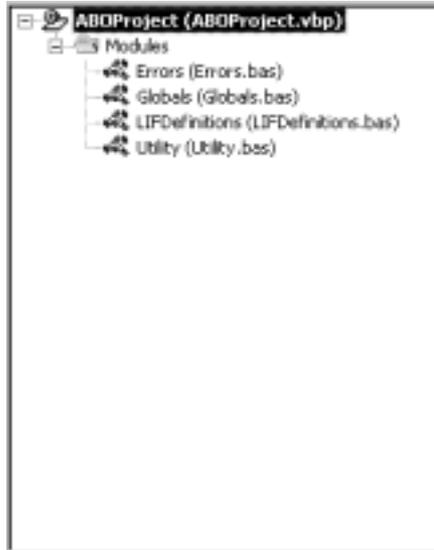


ABO Project Wizard — Ready To Create New ABO Project

- 8 Select the wizard.
- 9 Select Invoke Wizard.
This option launches the ABO wizard after the project is created.

10 Click Finish.

The generated ABO project is displayed in Project Explorer:



ABO Project in Project Explorer

Framework Components for the ABO Project

The following table describes the Construct Spectrum framework components that are included in the generated ABO project:

Component	Description
Errors.bas	Provides error-raising capabilities for the ActiveX component.
Globals.bas	Contains definitions, variables, and help routines used by the generated ABOs.
LIFDefinitions.bas	Is empty in a new ABO project. It becomes populated with Natural data area definitions when ABOs are added to the project using the ABO wizard.
Utility.bas	Contains procedures that can be used in any type of Visual Basic application. For example, Subst performs substitutions into a string and is useful for developing international applications: <pre>'This is the message that contains substitution placeholders. 'This message might come from a resource file in the case of a localized app. smsg = "Value must be in the range %1 to %2." MsgBox Subst(smsg, 100, 10000), vbExclamation</pre>

Using the ABO Wizard

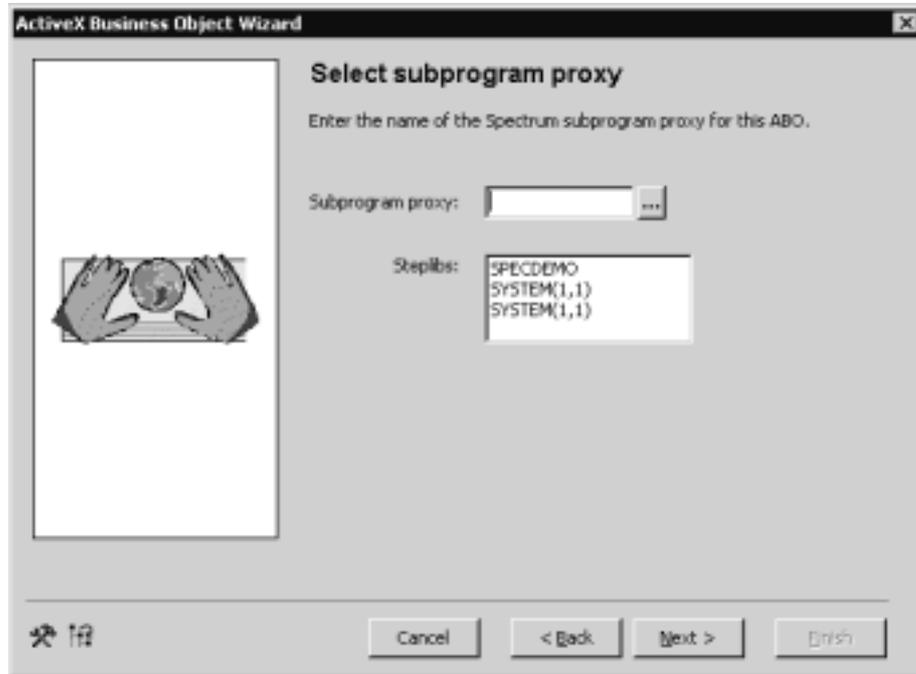
After creating the ABO project, use the ABO (ActiveX Business Object) wizard to generate an ABO for each Natural subprogram used in your application. This wizard is installed as a Visual Basic Add-In. In addition to generating a new ABO, you can use the ActiveX Business Object wizard to regenerate an existing ABO or display an existing ABO to examine its specifications.

- To generate an ABO and add it to the project:
 - 1 Select Wizards > ActiveX Business Object from the Spectrum menu. The ActiveX Business Object Wizard is displayed:



ActiveX Business Object Wizard

- 2 Click Next.
The Select Subprogram Proxy window is displayed:

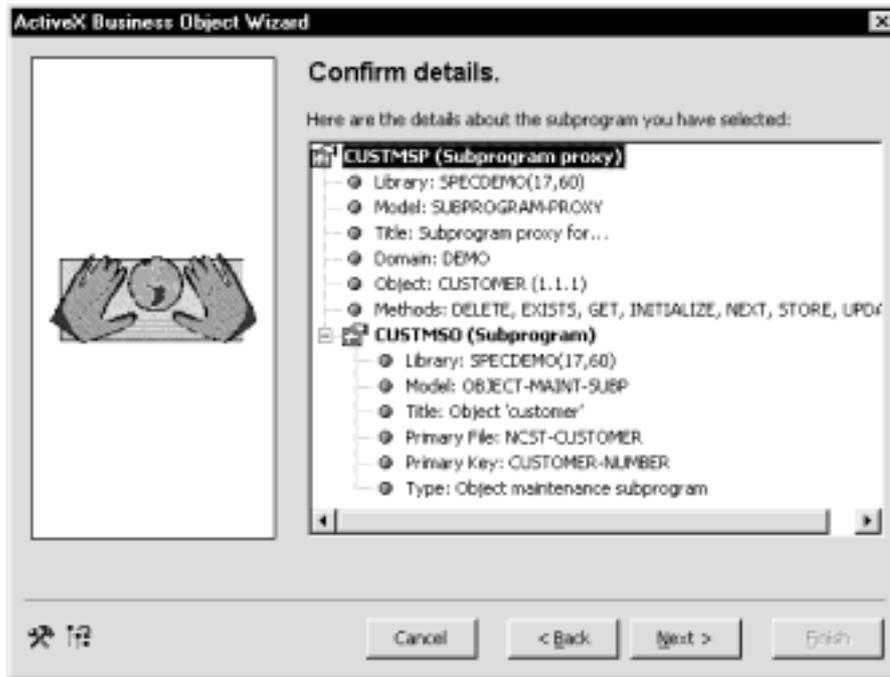


ActiveX Business Object Wizard — Select Subprogram Proxy

Note: For information about using the Spectrum Cache viewer or Configuration editor, see **Features of the Wizards**, page 49.

- 3 Enter the name of the subprogram proxy for the ABO.
You can also specify the name of a steplib.
- 4 Click Next.
The wizard performs the following processing:
 - Reads the ****SAG** lines of the subprogram to extract the model name
 - Reads the logical key names (for an object browse subprogram)
 - Reads the ****SAG** lines of the subprogram proxy to extract the domain, object, version, subprogram name, and 1:V overrides
 - Accesses the Spectrum files to retrieve the method names linked to this subprogram proxy

The Confirm Details window is displayed:



ActiveX Business Object Wizard — Confirm Details

This window shows the library, model name, title, domain, object, and methods for the subprogram proxy, as well as information about the proxy's subprogram.

- 5 Confirm that everything is correct.
To select a different proxy, click Back.

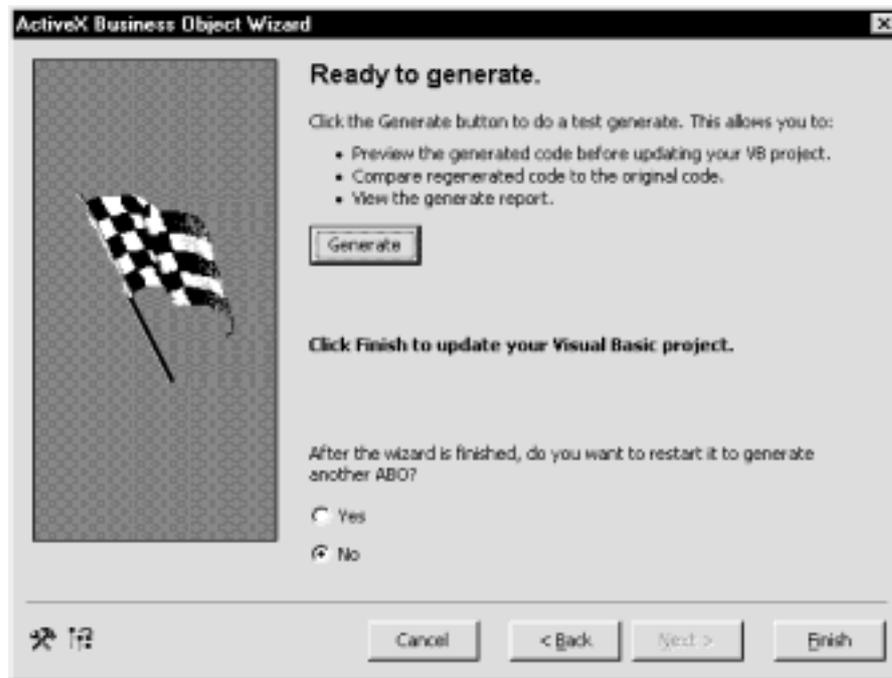
- 6 Click Next.
The Customize the ABO's Interface window is displayed:



ActiveX Business Object Wizard — Customize the ABO's Interface

- 7 Verify the default name supplied for the ABO and change it if desired.
- 8 Verify that the ABO will be generated into the correct project.
To change the project, click Change and select a different project.
- 9 Click Customize Properties to customize the ABO's properties.
For more information, see **Customizing the ABO**, page 98.
- 10 Verify the status of the ICSTPersist option.
This option allows the ABO to save its instance data at runtime and restore it later.

- 11 Check the status of the ICSTPropertyInfo option.
This option provides extended information about the properties exposed by the ABO. This information can be accessed at runtime. It includes:
- property name
 - VB data type
 - number of decimal places for numeric property
 - length of the data
 - logical format
 - read-only or not read-only
 - number of dimensions in an array
 - number of occurrences in each dimension
- 12 Click Next.
The Ready to Generate window is displayed:



ActiveX Business Object Wizard — Ready to Generate

- 13 Do one of the following:
 - Click Generate to view the generation report.
If you have a code comparison utility installed and configured for use with Construct Spectrum, you can also compare the newly generated code with code from an earlier generation of the module. For information about using a code comparison utility, see **Use Reports with a Code Comparison Tool**, page 67. For information about the generation report, see **Generating and Reviewing Reports**, page 63.
 - Click Finish to complete the generation.
When generation is complete, a message window informs you of the success or failure of the operation. If there were problems with the generation, the window prompts you to view the generation report.
- 14 Generate an ABO for each Natural subprogram used in your application.

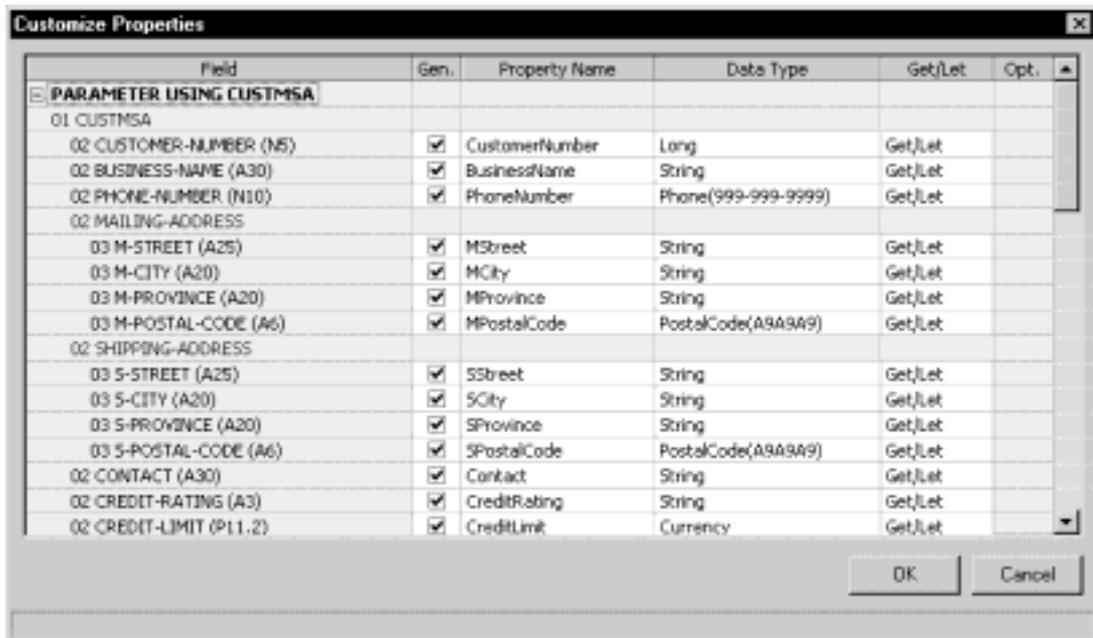
Customizing the ABO

You can customize the ABO's properties in the Customize Properties window or in the supplied user exits. These options are described in the following sections.

Customize Properties Generated for the ABO

You can customize or view the ABO interface before generating the ABO.

- To customize or view the ABO interface:
 - 1 Click Customize Properties in the Customize ABOs Properties window. The Customize Properties window is displayed:



Customize Properties Window

By default, the wizard generates properties for the fields in the object PDA for an object maintenance subprogram and in the key and row PDA for an object browse subprogram. For all other subprograms, the wizard generates properties for the entire PDA.

If the subprogram is an object maintenance subprogram, the wizard displays the method names and their generated derivations, of which the method names can be customized. If the subprogram is a browse subprogram, the wizard displays the logical key names.

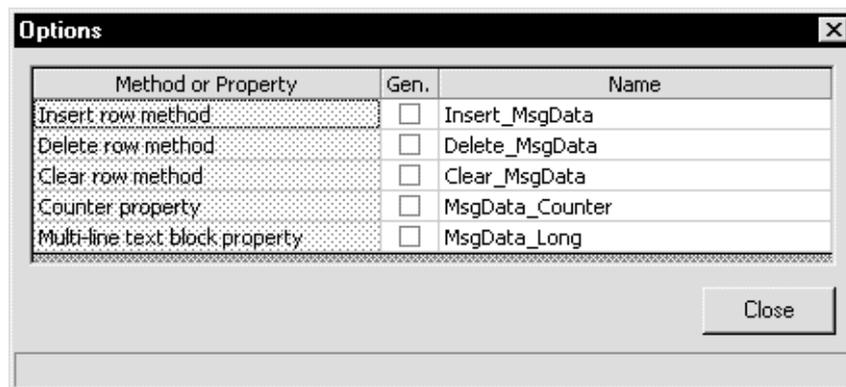
You can customize the derived property names, data types, and get/let (read/write) status. Using the check boxes in the Generate column, specify which properties should be generated. Any fields that have been changed from the default are highlighted.

The options in the Customize Properties window are:

Column	Description
Field	Name of the field in the Natural data area.
Gen	Indicates whether default properties are generated. Deselect the properties you do not want to generate.
Property Name	Name of the property generated for the Natural field.
Data Type	Native Visual Basic data type the property is declared as.
Get/Let	Get returns the value of a property; Let sets a property value.
Opt	Indicates whether added methods or properties are generated for the array. This option is applicable to MUs and PEs only. For more information, see the following section.

Opt Column

- To view additional properties for MUs and PEs:
 - 1 Click the Opt cell for the field.
The Browse button is displayed.
 - 2 Click Browse.
The Options window is displayed:



Options Window

Extra properties are generated for the ABO class by default. You have the option of manually renaming each method or property name. The Multi-line text block property is only available for alphanumeric MUs. This allows users to edit all the elements of an array at one time, in one continuous text string. For more information, see **Customizing HTML Before Generation**, page 103, *Construct Spectrum SDK for Web Applications*.

Customize the ABO within User Exits

You can also customize the ABO within user exits. The following user exits are supplied in the generated ABO.

GetAppService_.SetMethodAndBlocks

Use this exit to override the default method names and block numbers in the GetAppService_ procedure.

ICSTBrowseObject_LogicalKeyInfo.Extra

This exit resides in the Property Get LogicalKeyInfo procedure in the ICSTBrowseObject interface. The procedure provides information at runtime about the logical keys supported by the ABO. Use this exit to define additional logical keys that you have added to the ABO manually.

Note: This exit is only available in browse ABOs.

ICSTPersist_InstanceData.Get.Extra

Use this exit to persist additional module-level variables.

Note: This exit is only available if you generate the ABO with the ICSTPersist interface.

ICSTPersist_InstanceData.Let.Extra

Use this exit to restore the additional module-level variables that were persisted in the ICSTPersist_InstanceData.Get.Extra user exit.

Note: This exit is only available if you generate the ABO with the ICSTPersist interface.

ICSTPropertyInfo_PropertyInfo.Get.Extra

This exit resides in the Property Get PropertyInfo procedure in the ICSTPropertyInfo interface. This procedure provides information at runtime about the properties in the ABO's class. Use this exit to define additional properties that you added to the ABO class manually.

Note: This exit is only available if you generate the ABO with the ICSTPropertyInfo interface.

<CounterPropertyName>.Get.NullList

This is a dynamically generated user exit. Every array counter property procedure that is generated will have this user exit.

Array counter property procedures contain code that determines the number of array occurrences that are used. This code examines each occurrence of the array and checks whether certain fields are empty. If one of these fields is not empty, the code considers the array occurrence to be used.

Use this exit to specify the fields that should be checked and the values that the fields should have to be considered empty. The wizard always generates sample code into this exit consisting of the field names and the empty values. You can change the sample code after generating.

Tip: Because coded user exits are always preserved when regenerating, delete the existing exit if you want the wizard to regenerate the sample.

USING THE SUBPROGRAM-PROXY MODEL

This chapter describes the subprogram proxy, how to generate proxies using the Subprogram-Proxy model, and how to customize the proxy. It also contains information about adding a method to an application service definition, overriding block handling, versioning, and debugging.

The following topics are covered:

- **Overview**, page 104
- **Generating a Subprogram Proxy**, page 105
- **Generating Methods**, page 111
- **Overriding Block Handling**, page 116
- **Versioning Support**, page 120
- **Debugging Support**, page 120

Overview

Typically, you will use the Subprogram-Proxy model when tailoring an existing application. The major functions of this model are to generate:

- A subprogram proxy that interacts with the Spectrum dispatch service and the target Natural subprogram.
- The application service definition entry needed in the Construct Spectrum Administration subsystem.

The subprogram proxy acts as a bridge between the Spectrum dispatch service and a specific subprogram. When a request is initiated from a dialog or web page to the server (for example, when a user updates a customer record), information is sent from the client to the subprogram proxy on the server. The subprogram proxy then calls the appropriate object subprogram to fulfill the request.

The subprogram proxy is also responsible for converting data between the network transfer format and the native Natural data format used in the subprogram's PDA. It also provides optimized data block handling and creates application service definitions.

You must generate a subprogram proxy for each subprogram included in your application. You can create subprogram proxies using the VB-Client-Server-Super-Model, the Subprogram-Proxy model, or the Business-Object-Super-Model.

If you are creating a new application or have performed extensive changes to your application file relationships, use the super models to generate your application. The Business-Object super model generates object maintenance and browse subprograms, their PDAs, and subprogram proxies. For more information, see **Using the Business-Object-Super-Model**, page 71.

The VB-Client-Server-Super-Model generates the same Natural modules as the Business-Object super model, as well as the Visual Basic modules needed by client/server applications.

Accessing System Files

To generate a subprogram proxy, the Subprogram-Proxy model requires access to the unsecured data in the Construct Spectrum Administration subsystem files. It uses this data to provide an active help listing for the Domain field in Standard Parameters. Additionally, this model creates or updates the application service definition for the specified object subprogram.

The subsystem file containing the unsecured data must be available either through an LFILE designation in your Natural startup or through the Natural nucleus used in the session in which you are generating (NT-FILE parameter must be specified).

Generating a Subprogram Proxy

This section describes how to generate a subprogram proxy and considerations to be aware of when generating the proxy.

- To generate a subprogram proxy:
 - ❑ **Step 1: Specify Standard Parameters**, page 106.
 - ❑ **Step 2: Specify the Number of Occurrences Returned**, page 108.
 - ❑ **Step 3: Add User Exits**, page 109.
 - ❑ **Step 4: Generate the Subprogram Proxy**, page 110.

Before using the Subprogram-Proxy model, consider the following:

- **Maintain one application service definition for each business object**
An application service definition specifies the methods and the subprogram from which each method is executed for a business object. The application service definition is created or updated when you generate a subprogram proxy.

To maintain one application service definition for each business object, ensure that the domain name, object name, and version number are the same when you generate each subprogram proxy for the business object. For example, if you have a Customer business object that has both a maintenance and a browse function, generate one subprogram proxy for the maintenance function and one for the browse function. To ensure that only one application service definition is created for both the maintenance and browse functions, specify the same domain, object, and version when you specify the model parameters for each subprogram proxy.

For more information, see **Generating Methods**, page 111.

- **Define 1:V Variables**
When generating a subprogram proxy, pay special attention to subprograms that have 1:V variables (such as object browse subprograms). Subprograms use the Natural 1:V notation to define the row parameter that allows an arbitrary number of records to be returned to the client. To minimize the number of calls to the server, you normally want as many records as possible returned for each server request. However, the more records requested, the longer it takes to satisfy each request. Also ensure you do not specify more occurrences than will fit within the maximum 32K communication area available for each request.

Step 1: Specify Standard Parameters

- To specify standard parameters for the subprogram proxy:
 - 1 Invoke the Subprogram-Proxy wizard.
 - 2 Select Standard Parameters:

The screenshot shows the 'SUBPROGRAM-PROXY Wizard' dialog box, specifically the 'Standard Parameters' step. The wizard has three steps: 'Start', 'Standard Parameters' (which is currently selected and highlighted in black), and 'Finish'. The 'Standard Parameters' section contains the following fields and options:

- Module:** CUSTMSP (SPECDEMO on 17,60)
- System:** DEMO
- Title:** Subprogram proxy for Cust
- Description:** This subprogram proxy supports the customer order maintenance system
- Subprogram:** CUSTMSO (with a browse button '...')
- Domain:** DEMO (with a browse button '...')
- Object name:** CUSTOMER
- Version:** 1.1.1
- Generate trace code
- Compress network data
- Encrypt network data

At the bottom of the dialog, there are five buttons: 'Validate', 'Cancel', '< Back', 'Next >', and 'Finish'. An 'Edit 1:V Overrides' button is also present next to the Subprogram and Domain fields.

Subprogram-Proxy Wizard — Standard Parameters

This window is similar for all model wizards. For information about the parameters and options, see the online help.

Note: The parameters and options available in the Construct Windows interface and the Generation subsystem are identical. For information, see **Using the Subprogram-Proxy Model**, page 103, *Natural Construct Generation*.

Tip: Follow the Construct Spectrum naming conventions and use “MSP” for the last three characters of a maintenance subprogram proxy or “BSP” for the last three characters of a browse subprogram proxy. This will make it easier to identify the subprogram proxy when listing modules.

- 3 Specify the following parameters for the subprogram proxy:

Parameter	Description
Subprogram	Name of the subprogram for which the proxy is being generated. For example, to generate a subprogram proxy for the ORDMSO Customer Order subprogram, enter “ORDMSO”.
Domain	Name of the domain. To set up security for your applications, link selected groups of users to each domain.
Object name	Name of the business object. For example, a customer information business object can be called “Customer”. For more information, see Versioning Support , page 120.
Version	Version level of your package. This number consists of three parts: version, release, and SM level.
	Note: One application service definition is created for both a maintenance and a browse dialog if the domain, object, and version values are identical on this panel for their respective subprogram proxies.
Generate trace code	Use this option if you are developing an early iteration of your application or if runtime errors are occurring in the application. This option adds code to the proxy that can help you determine the cause of a parameter format error. If your application is stable, do not generate trace code. This improves the performance of your subprogram proxy and reduces the amount of generated code.
Compress network data	Use this option if the proxy transmits large volumes of data to the client. If you are generating a browse subprogram proxy and a large volume of data is being sent to the client, select data compression.
Encrypt network data	Use this option if the proxy transmits sensitive data to the client.

Note: The Compression and Encryption parameters apply only to data sent to the client. If you are creating a client/server application, you can enable compression and encryption for data sent from the client to the server. Mark the Compress network data and Encrypt network data check boxes in the Standard Parameters window of the VB-Maint-Object model or VB-Browse-Object model (depending on the type of dialog you are creating).

Step 2: Specify the Number of Occurrences Returned

Next, specify the maximum number of 1:V arrays that can be returned to the client for each request. A 1:V array can consist of either one-dimensional data, such as a list of repeating values, or two-dimensional data, such as a row of record data.

For maximum efficiency, specify 20 occurrences for each subprogram structure (PDA).

- To specify the maximum number of occurrences to return for each request:
 - 1 Click Edit 1:V Overrides in the Standard Parameters window.
If no fields in the target subprogram use the 1:V notation, a message is presented indicating this. Otherwise, the model determines these values and displays a window listing their names. For example:



Edit 1:V Overrides Window

Note: If you are using the Subprogram-Proxy model in the Generation subsystem to generate your subprogram proxy, press PF5 (1:V) on the Standard Parameters panel to access the 1:V Overrides panel.

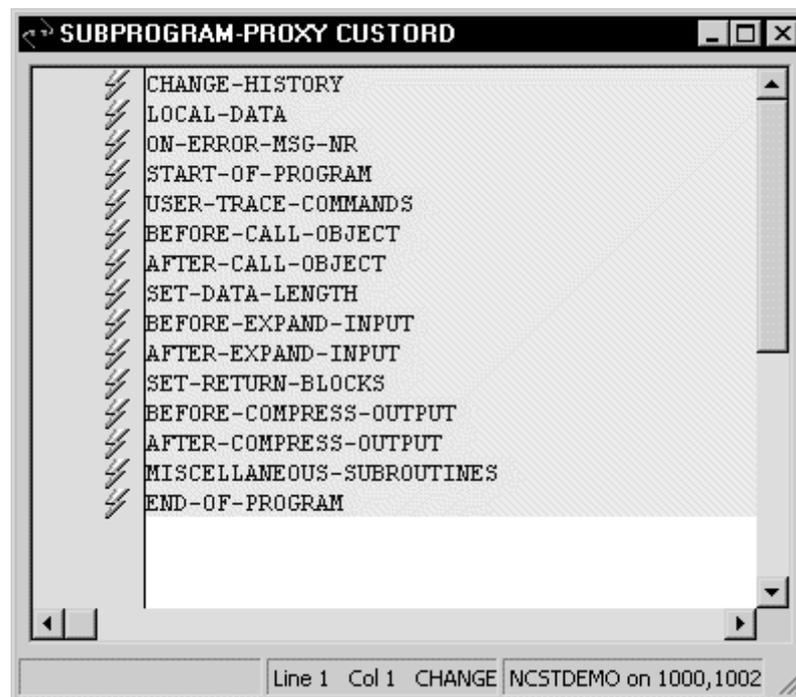
- 2 Specify the maximum number of occurrences that can be returned to the client with each call to the server.
Click Refresh to update the information by making another call to the server.
- 3 Click OK to return to the Standard Parameters window.

Step 3: Add User Exits

After supplying model parameters, you can customize the generation results by creating user exit code for the module. For example, you can use user exits to modify block handling or add block handling for new methods.

➤ To add user exits:

- 1 Click Finish in the Standard Parameters window.
The user exits available for the Subprogram-Proxy model are displayed:



User Exits for the Subprogram-Proxy Model

The icon on the left indicates whether sample code is generated for the user exit.

- 2 Right-click the user exit and select Generate Sample from the shortcut menu.
- 3 Modify the code as required.

You can also generate sample code from the user exit list by selecting User Exit List from the View menu or clicking the View button.

➤ To generate sample code:

- 1 Select the user exit.
- 2 Click Generate sample.

Note: You can also add new user exits and write code for them. For information, see **Invoke User Exit Editor Function**, page 69, *Natural Construct Generation*.

Step 4: Generate the Subprogram Proxy

➤ To generate the subprogram proxy:

- 1 Select Generate from the File menu or click the Generate button on the toolbar. The Generate window is displayed, showing module and status information.
- 2 When generation completes successfully, select Stow <Module name> from the File menu.

Once generation has completed, the following two items exist:

- The generated subprogram proxy.
- The application service definition in the Construct Spectrum Administration subsystem.

Generating Methods

The subprogram proxy generates a method for each of the actions supported by an object subprogram. The application service definition includes the following methods:

Object	Method
Maintenance	Delete Exists Get Initialize Next Store Update
Browse	Browse
Any other type	Default

If a subprogram proxy is generated using the same domain, business object, and version as another subprogram proxy, the new methods are also added to the application service definition. This allows a single application service definition to access both the maintenance and browse functions of a business object.

Access the Application Service Definitions

- To view application service definition records:
 - 1 Invoke the Construct Spectrum Administration main menu.
 - 2 Enter “AA” in the Function field.
The Application Administration main menu is displayed.
 - 3 Enter “MM” in the Function field.
The Application Administration Maintenance menu is displayed.
 - 4 Enter “AS” in the Function field.
The Maintain Application Service Definitions panel is displayed:

```

BSIF__MP          Construct Spectrum Administration Subsystem          BSIF__11
Jan 30            Maintain Application Service Definitions              3:15 PM

Action (A,B,C,D,M,N,P)  _

Domain.....: DEMO_____ *
Object.....: PRODUCT_____
Version.....: 01 / 01 / 01
Description.....: PRODUCT_____
Default subprogram proxy: PRODMSP_
Steplibs.....: _____ *

01          Method Name          Subprogram
          -----          -----
          BROWSE_____          PRODBSP_
          DELETE_____          _____
          EXISTS_____          _____
          GET_____          _____
          INITIALIZE_____          _____
Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
confm help retrn quit          flip pref bkwrd frwr          main
Appl Srv Definition DEMO-PRODUC displayed successfully

```

Maintain Application Service Definitions Panel

Use this panel to add a method.

Add a Method

You can add custom methods to a maintenance or browse object. For example, if your maintenance object requires special processing that is not provided by one of the supplied methods, you can add a new method to implement the processing.

➤ To add a new method:

- 1 **Step 1: Create the Method**, page 113.
- 2 **Step 2: Update the Application Service Definition**, page 113.
- 3 **Step 3: Update the Library Image File**, page 114.

Optionally, you can transmit only the data required for the custom method when the method is invoked. For more information about optimizing the handling of data for custom method, see **Overriding Block Handling**, page 116.

Step 1: Create the Method

➤ To create the method:

- 1 Define the method in the USER-DEFINED-FUNCTIONS user exit for the subprogram and save your changes.
- 2 If the subprogram does not currently include this user exit, regenerate it using the Object-Maint-Subp model and select the USER-DEFINED-FUNCTIONS user exit.
For information about using the Object-Maint-Subp model and user exits, see **Object-Maint Models**, page 345, *Natural Construct Generation*.

Step 2: Update the Application Service Definition

➤ To update the application service definition:

- 1 Type “M” in Action.
- 2 Type the domain, object, and version of the application service definition you are updating in the appropriate fields.
- 3 Type the name of the method in Method Name.
Use the name that was specified when the method was created and added to the maintenance subprogram user exit.
- 4 If the subprogram proxy for this business object’s method is different from the default subprogram proxy specified for the business object, type the new subprogram proxy name in Subprogram Proxy; otherwise, leave the field blank.
- 5 If the steplib for this business object’s method is different from the default steplib specified for the domain, provide the new steplib name in Steplibs.
- 6 Press Enter.
The method is added to the application service definition.

Step 3: Update the Library Image File

The library image file (LIF) resides on your client and must be updated with the valid methods for a business object. To update the LIF, download the subprogram proxy to the Visual Basic project for the application and Construct Spectrum automatically adds the new method.

- To update the library image file with the method:
- 1 Open the project for your application in Visual Basic.
 - 2 Select Download Generated Modules from the Construct Spectrum Add-In menu. For more information, see **Downloading the Generated Modules**, page 107, *Construct Spectrum SDK for Client/Server Applications*.
 - 3 Download the subprogram proxy definition to your project. A maintenance subprogram proxy has the suffix “MSP” and a browse subprogram proxy has the suffix “BSP”.
 - 4 Save your changes and run the project. The new method is available for use in your application.

For web applications, you must regenerate the ABO, page handler and HTML template. For more information, see **Creating and Customizing a Page Handler**, page 75, and **Creating and Customizing an HTML Template**, page 91, *Construct Spectrum SDK for Web Applications*.

Override the Steplib Chain for the Domain

All business objects in an application service definition share the same domain. All business objects within a domain are accessed using the domain's steplib chain. You can, however, override the steplib chain for each business object or method defined in your application service definition.

- To override the steplib chain for the domain:
 - 1 Access the Maintain Application Service Definitions panel:

```

BSIF__MP          Construct Spectrum Administration Subsystem      BSIF__11
Jan 30            Maintain Application Service Definitions          3:15 PM

Action (A,B,C,D,M,N,P)  _

Domain.....: DEMO_____ *
Object.....: PRODUCT_____
Version.....: 01 / 01 / 01
Description.....: PRODUCT_____
Default subprogram proxy: PRODMSP_
Steplibs.....: _____ *

01          Method Name          Subprogram
          -----          Proxy          Steplibs *
-----
1 BROWSE_____          PRODBSP_          _____
2 DELETE_____          _____          _____
3 EXISTS_____          _____          _____
4 GET_____          _____          _____
5 INITIALIZE_____          _____          _____

Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10---PF11---PF12---
confm help retrn quit          flip pref bkwrd frwr          main
Appl Srvc Definition DEMO-PRODUC displayed successfully

```

Maintain Application Service Definitions Panel

For more information about this panel, see **Access the Application Service Definitions**, page 112.

- 2 Display the application service definition you want to modify.
- 3 Type "M" in Action.
- 4 For each method or business object that requires a special steplib, specify the steplib name in Steplibs.
- 5 Press Enter to update the application service definition.

Overriding Block Handling

The subprogram proxy optimizes level 1 parameter block handling for the default methods provided with your object maintenance and object browse subprograms. This optimization ensures that only the required data for a particular method is sent from the server to the client. This section describes the default block handling provided with the subprogram proxy and how to override this block handling, if necessary.

Default Block Handling

The following tables define which level 1 blocks are sent for each default method in your maintenance and browse subprograms.

Maintenance Subprogram Blocks Sent to Server

Function	Business Object Data	Business Object Key	Restricted Data	CDAOBJ2 (function)	CDPDA-M (message)
DELETE		X	X	X	
EXISTS		X		X	
GET		X		X	
INITIALIZE				X	
NEXT		X		X	
STORE	X			X	
UPDATE	X		X	X	

Maintenance Subprogram Blocks Returned to Client

Function and Flags	Business Object Data	Business Object Key	Restricted Data	CDAOBJ2 (function)	CDPDA-M (message)
DELETE and (Error = True or Return Object = False)				X	X
DELETE and (Clear After = False)			X	X	X
GET and Exists = False				X	X
EXISTS				X	X
NEXT and Exists = False				X	X
STORE and (Error = True or Return Object = False)				X	X
STORE and (Clear After = False and Derived Data = False)			X	X	X
UPDATE and (Error = True or Return Object = False)				X	X
UPDATE and (Clear After = False and Derived Data = False)			X	X	X
All Other Combinations	X		X	X	X

Browse Subprogram Blocks Sent to Server

Function	Key Data	Row Data	Restricted Data	CDBRPDA (function)	CDPDA-M (message)
BROWSE	X		X	X	

Browse Subprogram Blocks Returned to Client

Function	Key Data	Row Data	Restricted Data	CDBRPDA (function)	CDPDA-M (message)
BROWSE	X	X	X	X	X

Specify Overrides

You can override the default block handling rules listed in the previous tables and provide your own rules. For example, if you add a new method, you can specify which blocks are sent to the client. By default, custom methods transmit all data blocks.

- To override the default block handling:
 - 1 Define the custom block handling on the server.
 - 2 Define the custom block handling on the client.

Step 1: Define Block Handling On Server

You can set block handling overrides for every level 1 data block in a subprogram's parameter data. Define these overrides in the SET-RETURN-BLOCKS user exit for the subprogram proxy and regenerate the proxy. For information about regenerating, see **Generating a Subprogram Proxy**, page 105.

Disable a Block Unconditionally

- To disable a block unconditionally so that it is never sent to the client:
 - 1 Select the SET-RETURN-BLOCKS user exit for the subprogram proxy.
 - 2 Reset any block indicator that is not to be sent to the client.
Block indicators identify a data block and are named #PDA.#RB-*blockname*, where *blockname* is the name of the level 1 variable that defines the block.

Note: Code the statements in this user exit as part of a DECIDE FOR statement.

- 3 Add the following code:

```
WHEN #SPC-TRUE
RESET #PDA.#RB-BLOCKNAME/* Unconditional assignment
```

Send Blocks to the Client Conditionally

- To conditionally send blocks to the client:

In the SET-RETURN-BLOCKS user exit of the subprogram proxy, add a DECIDE clause that resets certain block selectors based on a condition. For example:

```
**SAG DEFINE EXIT SET-RETURN-BLOCKS
/* Do not return restricted data on a delete
WHEN CDAOBJ2.#FUNCTION = CDLMETH.DELETE
    RESET #PDA.#RB-CUSTMSR
/* Do not return object or restricted data on existence check
WHEN CDAOBJ2.#FUNCTION = CDLMETH.EXISTS
    RESET #PDA.#RB-CUSTMSA
        #PDA.#RB-CUSTMSR
**SAG END-EXIT
```

Adhere to the following guidelines when assigning the blocks:

- Know the name of each block you are assigning. The format is #PDA.#RB-*blockname*, where *blockname* is the name of the level 1 field.
- Reset only those blocks that are not to be returned to the client.

Step 2: Define Block Handling On Client

For information about defining block handling on the client, see **Step 3: Update the Library Image File**, page 114.

Versioning Support

You can create new versions of a subprogram proxy without affecting older versions. The version number specified when entering the model input parameters is part of the key used to store the associated application service definition. Versioning allows you to maintain a system without affecting existing applications. Each request issued from the client includes its required version number.

Note: When creating a new version of a subprogram proxy, use a new name. Otherwise, the existing version is overwritten.

Security Implications

Security definitions do not include the version number. This means that if the only thing about the subprogram proxy that changes is the version number, it will automatically be included in existing security definitions for the domain and business object name specified. If it requires a new security definition, the subprogram proxy domain or business object name should be changed to force the creation of a new application service definition. This new application service definition can then be secured as necessary.

Debugging Support

Subprogram proxies automatically support the DATASIZES and INITIALIZE trace options. These options return the size of the data blocks and their initialized values and are useful when debugging an application. You can add additional trace options in the USER-TRACE-COMMANDS user exit for the subprogram proxy.

For more information, see **Debugging Your Client/Server Application**, page 161.

USING BUSINESS DATA TYPES (BDTS)

This chapter describes business data types (BDTs) as they relate to client/server and web applications. It describes the composition of BDTs and how to create and use them.

The following topics are covered:

- **Overview**, page 122
- **Understanding and Using BDTs**, page 123
- **Creating and Customizing BDTs**, page 141

Overview

The first section of this chapter is of particular interest to users of BDTs. It discusses the concept of BDTs in general terms and gives you a good understanding of the benefits of using BDTs and how they work. The second section is of interest to authors of BDTs. It discusses how to create and customize BDTs in both the client/server and web framework components.

BDTs provide a way to present data to the user in a format that is consistent and based on business conventions rather than on programming language conventions. For example, a BDT can format a phone number with dashes (-) or some other delimiter value so that it is easily recognized by the user as a phone number.

To accomplish this, BDTs convert data values between simple internal Visual Basic data types (such as String, Long, Currency, Date, and Boolean) and values that are displayed to the user in a browse or maintenance dialog.

Construct Spectrum also uses BDTs to create sample strings to calculate the length of GUI controls.

Understanding and Using BDTs

There is some commonality between BDTs that are used in client/server applications and web applications. The following sections discuss BDTs as they relate to both the client/server and web framework components.

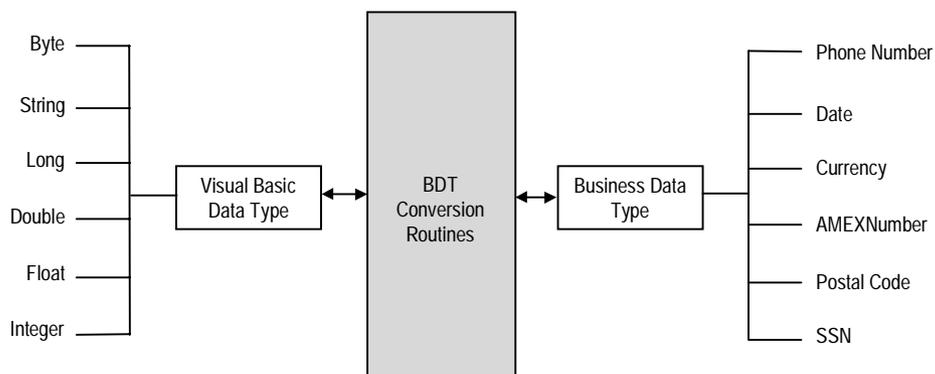
Benefits of Using BDTs

Using business data types offers three primary benefits:

- **Consistency**
BDTs ensure that a specific data type is displayed in the same format throughout the application.
- **Flexibility**
BDTs recognize a variety of input formats which makes using the application easier.
- **Accuracy**
BDTs centralize the validation code for a data type and provide a consistent mechanism for returning validation error messages.

Relationship With Visual Basic Data Types

The relationship between Visual Basic data types and business data types is many-to-many. That is, a Visual Basic Double variable can represent more than one BDT, such as Phone Number, AMEX Number, or Currency. Conversely, a Phone Number BDT could be mapped to Visual Basic String, Double, or Float variables. The Visual Basic data types to which a BDT can be applied depend on the considerations written into the BDT's conversion routine.



Relationship Between Visual Basic Data Types And BDTs

Construct Spectrum includes a set of standard BDTs. You can use these BDTs as they are or you can customize them. You can also write your own BDTs. If there is a piece of information whose format you are constantly validating, consider writing a BDT to handle it. Once a BDT has been created, you can use it in other applications.

Composition of a BDT

A BDT is composed of a name, a conversion routine, and the list of modifiers it can use.

Name

Applications need only the name of the appropriate BDT to perform the conversion to and from a display value.

Conversion Routine

The conversion routine converts data between an internal Visual Basic data type and a displayable format.

The `BDTConversion` object is used internally by BDT conversion routines. When the application calls one of the BDT controller's conversion methods, the controller creates a `BDTConversion` object and initializes it with details about the conversion requested. For example, the BDT controller will supply the BDT name, any modifiers associated with it, and any Natural format provided. The BDT controller then calls the conversion routine for the specified BDT, passing the `BDTConversion` object as a parameter.

The conversion routine uses the properties of the `BDTConversion` object to determine what type of conversion to perform (convert to display, convert from display, or create sample string), to get information about the modifiers used, the Natural format specified, and to return the converted value.

Modifiers

Use modifiers to override the default conversions that are performed by a BDT's conversion routine.

Elements of a BDT

Each time an application uses a business data type, it involves a number of elements. The following sections describe these elements and how they relate to BDTs.

BDT Controller

The BDT controller knows about all the BDTs that the application uses. This is because the application registers all of its BDTs with the BDT controller when the application is started. Whenever an application uses a BDT, it relies on the BDT controller to locate and call the associated conversion routine as follows:

- 1 The application calls the BDT controller and passes it all the necessary parameters, including the name of the BDT and the value to be converted.
- 2 The BDT controller locates the conversion routine for the BDT.
- 3 The BDT controller calls the conversion routine, forwarding the parameters from the application.
- 4 The conversion routine does the conversion and returns the result to the BDT controller.
- 5 The BDT controller returns the result to the application.

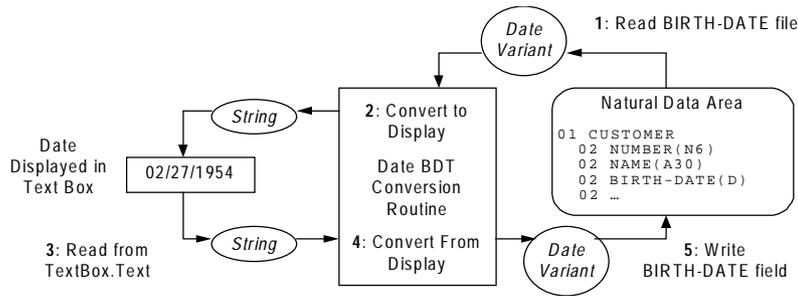
The application needs only the name of the BDT to accomplish the required conversion. The BDT controller is declared in the client/server framework as follows:

```
Public BDT As New BDTController
```

In the web framework, the BDTController object is a global multi-use object, meaning that you can invoke properties and methods of this class as if they were global functions. You do not have to create an instance of this class first because one will automatically be created.

How the Client Framework Uses BDTs

The client framework use BDTs to display values read from a NaturalDataArea object on the client. The values are then displayed in GUI controls. The following diagram shows the process of reading a value from a NaturalDataArea object on the client, displaying it in a GUI control where the user can modify the value, and copying the new value back to NaturalDataArea on the client. Once the value is copied back to the client, it can be sent to the server. The BDT in the following diagram is named DATE and is applied to the BIRTH-DATE field:



Processing Date BDT Applied to BIRTH-DATE Field

- 1 Reading the field value from the Natural data area returns a Visual Basic Variant data type.
- 2 This value is formatted for display by the Date BDT's conversion routine through a call to ConvertToDisplay. The result is a String value.
- 3 The string is displayed on a form by assigning it to a GUI control. The value displayed in the text box can be edited by the user.
- 4 When the user is finished editing, the edited value is read from the Text property for the TextBox control.
- 5 This string is converted back to a variant by the Date BDT's conversion routine through a call to ConvertFromDisplay. If the string does not contain a valid date, or the conversion routine cannot interpret the user's value correctly, an error is returned.
- 6 The new value is assigned back to the field in the Natural data area, which can then be sent to the server, for example, in the case of an update to the server database.

In a web application, BDTs are implemented internally. To change the BDT used by a field, you can define a user exit. For information, see **Customizing a Page Handler**, page 84, *Construct Spectrum SDK for Web Applications*.

Conversion Routines

When an application uses a BDT, the BDT controller calls the conversion routine. The conversion routine offers three services that affect the appearance of data:

- Converts the value in a Visual Basic data type to display in business format.
- Converts the value from its business format display to a Visual Basic data type.
- Creates a sample display value that is representative of the display values produced by the BDT.

In applying the second service, the conversion routine returns an error message if an inappropriate value is passed to it.

ConvertToDisplay Method

The ConvertToDisplay method converts a value from a Visual Basic data type to a display format. This method takes the value, and either the name of a BDT or a Natural format, and returns a string that is formatted for display. The syntax is:

```
Function ConvertToDisplay(RawData As Variant, _  
                        Optional BDTName As String, _  
                        Optional NatFormatLength As String _  
                        ) As String
```

For example, in a client/server application:

```
txtBirthDate.Text = BDT.ConvertToDisplay(custpda("BIRTH-DATE"), _  
                                       "Date")
```

You can specify a BDT name, a Natural format, or both. If you do not specify a BDT name, the BDT controller uses the Natural format (for example, N6) to choose an appropriate BDT first, and then calls that BDT's conversion routine.

If you do specify a BDT name, that BDT's conversion routine can use the optional Natural format to further refine how it performs the conversion or interprets the data. For example, the Numeric BDT uses the Natural format to determine how many decimal places to display. The Date BDT uses the Natural format as follows:

- If the Natural format is D, interpret the variant data as a date.
- If the Natural format is N6, P6, or A6, interpret the variant data as a date in the format YYMMDD.
- If the Natural format is N8, P8, or A8, interpret the variant data as a date in the format YYYYMMDD.

ConvertFromDisplay Method

The ConvertFromDisplay method converts a value from a display format to a Visual Basic data type. This method takes the display value, and either the name of a BDT or a Natural format, and returns a variant value that can be manipulated further by the application. The syntax is:

```
Function ConvertFromDisplay(FormattedData As String, _
    Optional BDTName As String, _
    Optional NatFormatLength As String _
) As Variant
```

For example, in a client/server application:

```
custpda("BIRTH-DATE") = BDT.ConvertFromDisplay(txtBirthDate.Text, _
    "Date")
```

You can specify a BDT name, a Natural format, or both. Using these optional parameters has the same result as in ConvertToDisplay.

ConvertInPlace Method

The ConvertInPlace method allows you to validate and reformat a value in a GUI control, such as in a LostFocus event. This method takes a formatted value by reference, internally calls ConvertFromDisplay, and then passes the result back to ConvertToDisplay which returns the new formatted result. For example, in a client/server application:

```
Private Sub txtBirthDate_LostFocus()

    Dim stemp As String

    stemp = txtBirthDate.Text
    BDT.ConvertInPlace stemp, "Date"
    txtBirthDate.Text = stemp

End Sub
```

When the user moves out of the field, the field value is validated and, if valid, is reformatted according to the date format used by the BDT. For example, if a user enters “feb 5”, the input is reformatted to the date format chosen, such as 2/5/1999, when the user leaves the field. The syntax is:

```
Function ConvertInPlace(ByRef FormattedData As String, _
    Optional BDTName As String, _
    Optional NatFormatLength As String _
) As Variant
```

You can specify a BDT name, a Natural format, or both. Using these optional parameters has the same result as in ConvertToDisplay.

The returned value is the value returned by the internal call to ConvertFromDisplay, so you can perform additional processing on the value entered by the user.

CreateSampleString Method

The `CreateSampleString` method creates a sample displayable value for each BDT. This sample value can be used as a template to determine the dimensions of the associated control or to determine how wide a column in a browse dialog must be to display the BDT value. The syntax is:

```
Function CreateSampleString(Optional BDTName As String, _  
                           Optional NatFormatLength As String _  
                           ) As String
```

You can specify a BDT name, a Natural format, or both. Using optional parameters allows you to further refine how the BDT performs the conversion or interprets the data.

You can use the returned value to calculate the required width of a `ListView` control column or a `TextBox` control used to display this business data type.

Modifiers

The processing performed by a BDT can be refined using special modifiers. Each business data type defines its own set of modifiers to provide the flexibility it needs.

Individual modifiers are separated by commas, and each modifier must be introduced by a name. Modifiers have names such as `TRIM`, `CASE`, `DEC`, and `ROUND`.

In calls to the conversion routines, use the format `name=value`, where `name` is the modifier you want to use and `value` controls the behavior of the conversion routine for the given modifier. Append modifiers to the BDT name parameter with commas. The following code invokes the `Numeric` BDT's conversion routine and uses the `DEC` modifier to specify that two decimal places should be displayed in the value and the `ZERO` modifier to suppress display of the value when it is 0.

For example, in a client/server application:

```
txtHours.Text = BDT.ConvertToDisplay(dblHours, _  
                                     "Numeric,DEC=2,ZERO=OFF")
```

For more information about the modifiers supported by each BDT, see **BDTs Supplied With Construct Spectrum**, page 133.

Natural Formats

When you omit the BDT name in calls to the `ConvertToDisplay`, `ConvertFromDisplay`, `ConvertInPlace`, or `CreateSampleString` method, you must provide the Natural format. The BDT controller uses this format to choose which BDT to use for the conversion. It does this by calling a Natural-to-BDT mapper function. This function provides the most appropriate BDT to use for each Natural format.

The mapper function must be registered with the BDT controller. In the client/server framework, the mapper is implemented as a method of the `StandardBDTs` class and is registered in its `SelfRegister` method. For more information, see **Register a BDT**, page 145.

For information about registering BDTs in the web environment, see **Register BDTs in the Web Framework**, page 151.

Handling Errors Returned from a BDT Conversion Routine

The BDT controller has four properties that return error information from the conversion routines. These properties can be examined after a call to `ConvertFromDisplay` or `ConvertInPlace`. The BDT conversion routines place information in these properties if a conversion error occurs. The application can then examine the properties on return from the call and display the error to the user:

Property	Contents
<code>ErrorCode</code>	Numeric error code. Each BDT can define its own error codes. The application makes program flow decisions based on this value.
<code>ErrorMsg</code>	Error message. This message should provide useful information.
<code>ErrorPos</code>	Position of the first invalid character.
<code>ErrorLen</code>	Number of invalid characters.

To show the user where invalid characters are, an application can use the `ErrorPos` and `ErrorLen` properties to set the `SelStart` and `SelLength` properties of a `TextBox` control.

Example code using error information properties in a client/server application

```
Private Sub txtBirthDate_LostFocus ()  
  
    Dim sdate As String  
  
    sdate = txtBirthDate.Text  
    BDT.ConvertInPlace sdate, "Date"  
    If BDT.ErrorCode Then  
        txtBirthDate.SelStart = BDT.ErrorPos  
        txtBirthDate.SelLength = BDT.ErrorLen  
        MsgBox BDT.ErrorMsg, vbExclamation  
        txtBirthDate.SetFocus  
    Else  
        txtBirthDate.Text = sdate  
    End If  
  
End Sub
```

Warning:

A conversion routine may set `ErrorPos`, but not `ErrorLen`. In the sample code above, it will not cause problems.

How Web Applications Use BDTs

Construct Spectrum web applications use BDTs as a way to format and validate user input for display on web pages. No formatting or validation is done in the web browser, instead the work is done on the web server inside of the Spectrum web application component, ABOInterface. To determine what BDTs to use, the page handler queries the ABO at runtime for the logical format each property provides. These logical formats are translated into BDT names. You can override the translation and logical formats of properties in the BDT.Overrides user exit.

For example, in a page handler:

```
Private Sub ICSTPageHandler_Initialize(...)

    With m_ABOInterface
        Set .ABOObject = m_ABO

        .Init ERR_SESSION_KEY, m_RequestData.Session, m_RequestData.Request
        '<cst:EXIT Name="BDT.Overrides">
            ' For the CustomerPhoneNumber property use a phone BDT.
            .BDT("CustomerPhoneNumber") = "Phone"
            ' Use an alpha BDT for the logical format PostalCode.
            .LogicalFormatBDT("PostalCode") = "Alpha"
        '</cst:EXIT>
    End With
End Sub
```

When a maintenance or browse HTML template is parsed and FIELD tags are detected, the value to be displayed is formatted using the correct BDT.

When a user submits a web page that includes properties on an HTML form to be updated, the ABOInterface component is used to update these properties. Part of the process includes validating the data included on the form before updating the property. If a BDT validation error occurs, the property is flagged for an error and the user's action is cancelled. When the web page is returned to the user, the properties in error are highlighted in red (Internet Explorer) or an error graphic is displayed next to the field (Netscape Navigator) and the error messages are displayed at the bottom of the page.

For more information, see **Customizing a Page Handler**, page 84, *Construct Spectrum SDK for Web Applications*.

BDTs Supplied With Construct Spectrum

This section describes the standard BDTs supplied with Construct Spectrum. The following sections describes each BDT, lists the modifiers it supports, and describes what each modifier does.

Alpha

Apply the Alpha BDT to alphanumeric data.

Modifier	Description
TRIM=L T LT	Trims leading spaces (L), trailing spaces (T), or leading and trailing spaces (LT). Default is no trimming. This affects ConvertToDisplay and ConvertFromDisplay behavior.
CASE=U L	Forces the text into uppercase (U) or lowercase (L). Default is to not change the case. This affects ConvertToDisplay and ConvertFromDisplay behavior.

Boolean

Apply the Boolean BDT to data that can have a value of either False or True.

Modifier	Description
EM=<False> <True>	Displays the <False> string for False and the <True> string for True. Default is EM=False True. ConvertFromDisplay compares the formatted data to the <False> and <True> strings and recognizes a match if the value matches unambiguously to the beginning of either string. This is not case-sensitive.

The following examples show various types of edit mask values, user input, and each result.

EM Value	Formatted Value	Raw Value
EM=False True	T	True
	t	True
	tr	True
	TRU	True
	F	False
	false	False
	yes	Error: Invalid
	<blank>	Error: Invalid

Modifier	Description (continued)		
	EM=True False	true F	False True
	EM=Off On	off on o	False True Error: Ambiguous
	EM= X	x <blank> xx	True False Error: Invalid

Time

Apply the Time BDT to any time value. The Time BDT supports the following Natural formats:

Natural Format	Visual Basic Data Type	Description
T	Date, Variant	If the value is Null, ConvertToDisplay returns an empty string.
N7 or P7	Long, Single, Double, or Currency	Numeric value is interpreted as HHMMSST.
A7	String	Alpha value is interpreted as HHMMSST.

Numeric

Apply the Numeric BDT to any numeric data.

Modifier	Description
DEC= <i>n</i>	Forces the display of <i>n</i> decimal places. Default is to display as many decimal places as there are significant decimal digits when the Natural format is not provided, or to use a fixed number of decimal places if the Natural format is provided. In this latter case, use DEC=-1 to ignore the Natural format and display significant decimal digits only.
ROUND= <i>n</i>	Rounds the value to <i>n</i> decimal places. If <i>n</i> is negative, it rounds to the left of the decimal place. Default is no rounding.
GS=OFF ON	Used to suppress (OFF) or display (ON) group separators (thousands separators). Default is GS=OFF.

Modifier	Description (continued)
ZERO=OFF ON	Suppresses (OFF) or displays (ON) zero values. Default is ZERO=OFF.
SIGN=OFF ON	Suppresses (OFF) or displays (ON) the sign for positive numbers. Default is SIGN=OFF.
MULT= <i>n</i>	ConvertToDisplay multiplies the raw value by <i>n</i> . ConvertFromDisplay divides the value by <i>n</i> before returning the raw value. <i>n</i> can be any positive or negative numeric value except zero. Default is MULT=1.
SCIENTIFIC=OFF ON	Displays the value in normal (OFF) or scientific notation (ON). Default is SCIENTIFIC=OFF.
EM= <i>xxx</i>	Any format string understood by the Visual Basic Format function. ConvertToDisplay uses the Format function to format the value according to that format string.
STRICT=OFF ON	Used by ConvertFromDisplay to determine how to deal with non-numeric characters in the formatted value. OFF quietly discards non-numeric characters and ON generates an error if the value contains non-numeric characters. The default is STRICT=ON. Has no effect on ConvertToDisplay.

Currency

Apply the Currency BDT to any currency values.

Modifier	Description
ZERO=OFF ON	Suppresses (OFF) or displays (ON) zero values. Default is ZERO=ON.

Date

Apply the Date BDT to any date value. The Date BDT supports the following Natural formats.

Natural Format	Visual Basic Data Type	Description
D and T	Date, Variant	If the value is Null, ConvertToDisplay returns an empty string.
N6 or P6	Long, Single, Double, or Currency	Numeric value is interpreted as YYMMDD.
N8 or P8	Long, Single, Double, or Currency	Numeric value is interpreted as YYYYMMDD.
A6	String	Alpha value is interpreted as YYMMDD.
A8	String	Alpha value is interpreted as YYYYMMDD.

Referencing BDTs in Predict

You can attach a BDT name to a field in Predict by adding a keyword with the same name as the BDT and prefix it with 'BDT_'. For example, to cause an N8 field to be treated as a date value when displayed on a browse or maintenance dialog, add the BDT_DATE keyword to the field.

The following table lists the BDT keywords loaded into Predict during installation:

BDT	Predict Keyword
Alpha	BDT_ALPHA
Boolean	BDT_BOOLEAN
Currency	BDT_CURRENCY
Date	BDT_DATE
Numeric	BDT_NUMERIC
Phone	BDT_PHONE
PostalCode	BDT_POSTALCODE
Time	BDT_TIME
ZipCode	BDT_ZIPCODE

Defining BDTs

One of the most powerful things about BDTs is that you can customize existing BDTs or create your own. If there is information whose format you are constantly validating, consider writing a BDT to handle it. A perfect case for a customized BDT might be an organization-specific account number.

To define a BDT, you must provide the following:

- Name for the BDT
- List of modifiers it will support
- Display format it will use
- Natural formats it will support
- Variant data types it will support

Tip: To maintain consistency, follow the naming convention used in the Construct Spectrum client framework: use short names consisting of one or two words and mixed case (capitalize the first letter of each word).

Name

A BDT name can be any consecutive string of characters except commas. Leading and trailing spaces are ignored, and uppercase and lowercase are considered identical.

Modifiers

Individual modifiers are separated by commas and each modifier is introduced by a name. Modifiers have names such as TRIM, CASE, DEC, and ROUND. Modifier names can be any consecutive string of characters except commas or equal signs. Leading and trailing spaces are ignored; uppercase and lowercase are considered identical.

Natural Formats

In addition to modifiers, all BDT handlers can be passed the format and length of the Natural variable that will receive the contents of the converted strings. For example, the BDT handlers can use this information to apply truncation rules or insert defaults.

When you omit the BDT name in calls to `ConvertToDisplay`, `ConvertFromDisplay`, `ConvertInPlace`, and `CreateSampleString`, you must provide the Natural format. The BDT controller uses the Natural format to choose which BDT to use for the conversion. It does this by calling a Natural-to-BDT mapper function supplied in the Construct Spectrum client framework. The mapper function must also be registered with the BDT controller.

Variant Data Types

When converting from formatted data to raw data, decide what type of variant to use for the raw data. Using a phone number BDT, for example, you can return the phone number as a Visual Basic Double, String, Currency, or an array. As all of these data types have enough precision to store all digits of a phone number, choose a data type that is convenient for an application programmer.

The returned data type may also depend on the Natural format passed to the conversion routine. For example, a seven-digit telephone number with area code can be stored in an A10 field or an N10 or P10 field. The conversion routine can return a String variant if the Natural format is A and a Double variant if the Natural format is N or P.

Returning Conversion Error Information

Conversion routines return conversion error information to the BDT controller in the error properties of the `BDTConversion` object. The BDT controller copies these properties to its own properties having the same names. The client application examines the error properties of the BDT controller to determine if an error occurred.

When returning error information, the most important property to set in the conversion routine is `ErrorCode`. If this property is not set, the BDT controller and the client application do not know that an error has occurred because they make program flow decisions based on `ErrorCode`.

If you set `ErrorCode`, also set `ErrorMsg`, giving the client application a message to display to the user. To provide the most information to the client application, set `ErrorPos` and then optionally set `ErrorLen`.

When converting from formatted data to raw data, your conversion routine can range from very forgiving in the input allowed to very strict. For example, a forgiving conversion routine may throw away any non-numeric characters in a numeric BDT without returning an error, while a strict conversion routine might require the input to match a rigid format to be converted without error.

A forgiving conversion routine is easier to code because it contains comparatively few validations. Coding a strict conversion takes more time and may be more difficult to if the routine must examine the input character-by-character to determine if it is valid. However, your error messages can be more informative.

Handling Runtime Errors

Your conversion routine should use Visual Basic runtime error handling to trap any runtime errors that may occur. If they are not trapped by the conversion routine, Visual Basic transfers the error up the call chain to the first enabled error handler. The BDT controller that called the conversion routine has an enabled error handler and converts the error into &H80040206 — An unhandled runtime error occurred when calling the method %1 in object %2:Error %3, %4.

The client application is typically not prepared to handle a runtime error that occurs in a conversion routine that it called indirectly. Therefore, it is imperative that Visual Basic runtime errors are trapped in the conversion routines and translated into BDT-specific errors that are documented and returned in the error properties of BDTConversion.

Creating and Customizing BDTs

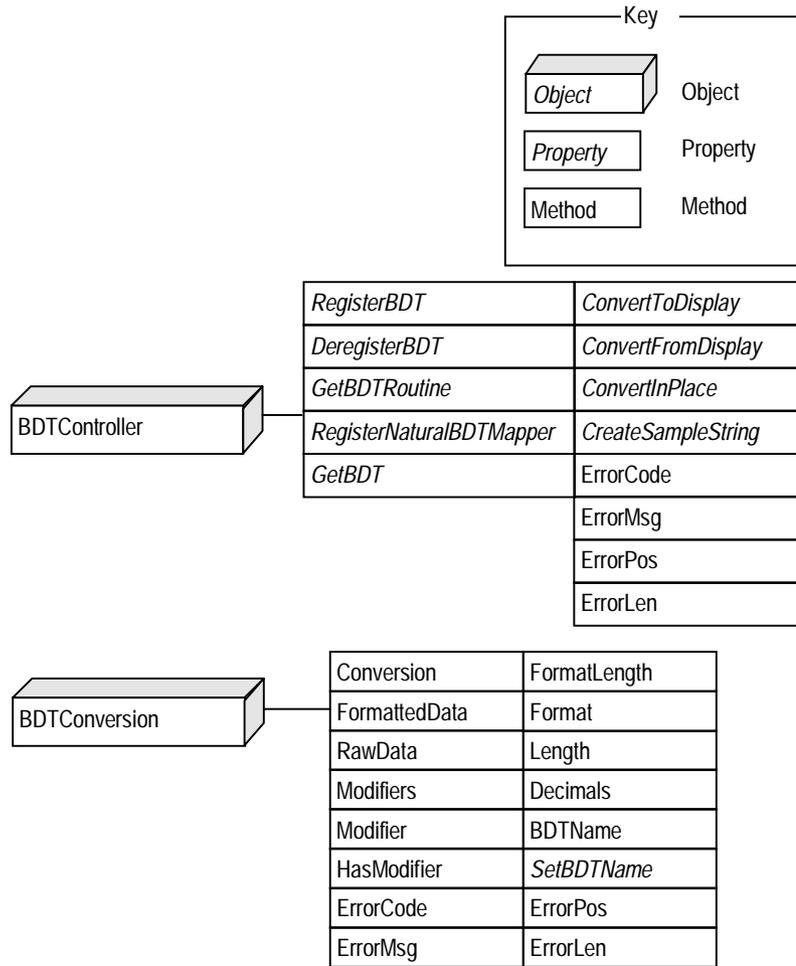
This section discusses how to create and customize BDTs. The client/server and web frameworks use an open architecture that allows you to add business data types tailored to your application specifications.

BDTs and the Client/Server Framework

This section discusses how the client/server framework uses BDTs. For information about creating BDTs for the web framework, see **BDTs and the Web Framework**, page 150.

Understanding the BDT Objects

The Construct Spectrum client framework has two objects that support BDTs: BDT-Controller and BDTConversion. The properties and methods of these objects are shown in the following diagram:



Properties and Methods of the BDT Objects

The BDTController object is used by the application to register its BDTs (the startup code in the Construct Spectrum client framework does this for you) and to call BDT conversion routines.

Each of the ConvertToDisplay, ConvertFromDisplay, ConvertInPlace, and CreateSampleString methods and the ErrorCode, ErrorMsg, ErrorPos, and ErrorLen error properties are discussed in separate sections in this chapter. The remaining methods are related to registering BDTs, which is described in **Register a BDT**, page 145.

Create BDT Conversion Routines

BDT conversion routines must be implemented as public methods of an OLE automation object. This object can reside in an in-process server, an out-of-process server, or as a class in the Visual Basic project.

All BDT conversion routines must have the following syntax:

```
Public Sub xxx(BDTC As BDTConversion)
```

where *xxx* can be any name.

The following table describes all of the properties and methods of the BDTConversion object for the client/server framework. For the examples, assume the following call was made:

```
strHours = BDT.ConvertToDisplay(dblHours, _
                               "Numeric,ZERO=OFF,ROUND=2,STRICT=ON", _
                               "N3.2")
```

Property or Method	Description
Conversion	Tells the conversion routine what type of conversion to perform. Can be one of the following constants: bdtConvertToDisplay bdtConvertFromDisplay bdtCreateSampleString
FormattedData and RawData	Is one of the following: <ul style="list-style-type: none"> • When Conversion = bdtConvertToDisplay, the conversion routine reads the value in RawData, formats it for display, and assigns the formatted string value to FormattedData. • When Conversion = bdtCreateSampleString, the conversion routine assigns a sample string to FormattedData. For example: <pre>With BDTC Select Case .Conversion Case bdtConvertFromDisplay .RawData = cvtToRaw(.FormattedData) Case bdtConvertToDisplay .FormattedData = cvtToDisp(.RawData) Case bdtCreateSampleString .FormattedData = createSample() End Select End With</pre>
Modifiers	Returns the number of modifiers specified by the caller. In the example, Modifiers returns 3.

Property or Method	Description (continued)
Modifier	<p>Returns the value of a specific modifier or can be used to enumerate the modifiers used. In the example:</p> <pre>With BDTC Print .Modifier("ZERO") ' Prints "OFF" Print .Modifier(1) ' Prints "ZERO" Print .Modifier(2) ' Prints "ROUND" Print .Modifier(.Modifier(1)) ' Prints "OFF" End With</pre> <p>Note: Modifier names, such as ZERO or ROUND, are passed to the BDT conversion routine in uppercase. You do not have to use case-sensitive string comparisons when checking which modifiers were used.</p>
HasModifier	<p>Returns True if a specified modifier was used and False if not. In the example:</p> <pre>With BDTC Print .HasModifier("ZERO") ' Prints True Print .HasModifier("ROUND") ' Prints True Print .HasModifier("DEC") ' Prints False Print .HasModifier("#\$%^&") ' Prints False End With</pre>
FormatLength	<p>Returns the Natural format string used in the call. In the example, FormatLength returns N3.2.</p>
Format, Length, and Decimals	<p>Returns the format, length, and decimal portions, respectively, of the Natural format string used in the call. In the example, Format contains N, Length contains 3, and Decimals contains 2.</p>
BDTName	<p>Returns the name of the BDT from the call. In the example, BDTName contains Numeric.</p> <p>Note: BDT names are passed to the BDT conversion routine with the capitalization used when the BDT was registered. For example, if RegisterBDT was called to register the BDT mIxEdCase, and then ConvertToDisplay was called for the BDT MixedCase, BDTName contains mIxEdCase.</p>
SetBDTString	<p>Changes the BDT name and modifiers in the BDTConversion object.</p>
ErrorCode, ErrorMsg, ErrorPos, and ErrorLen	<p>Contain error information. The conversion routine should assign values to these properties if a conversion error occurred.</p>

To see how these properties and methods are used in BDTs, examine the conversion routines in the StandardBDTs.cls and CustomBDTs.cls modules.

Register a BDT

To make a BDT available to an application, the BDT controller needs to know about the BDT. This is done by registering the BDT with the BDT controller. To register the BDT, tell the BDT controller the name of the BDT and provide a pointer to the conversion routine. Conversion routines must be implemented as methods of an OLE automation object. To invoke a method, you must have a reference to the object in an object variable. The pointer to the BDT conversion routine consists of a reference to an object and the name of a public method in that object.

The registration process is shown in this example from the Construct Spectrum client framework. This code creates the BDT controller and instantiates the objects that contain the BDT conversion routines:

```
Public BDT As New BDTController

Private Sub InitializeBDTs()

    Dim StandardBDTs As New StandardBDTs
    Dim CustomBDTs As New CustomBDTs

    StandardBDTs.SelfRegister BDT
    CustomBDTs.SelfRegister BDT

End Sub
```

The registration actually occurs in the SelfRegister methods. The following example shows registration within the StandardBDTs class:

```
Public Sub SelfRegister(BDT As BDTController)
    BDT.RegisterBDT "Alpha", Me, "Convert_Alpha"
    BDT.RegisterBDT "Boolean", Me, "Convert_Boolean"
    BDT.RegisterBDT "Numeric", Me, "Convert_Numeric"
    BDT.RegisterBDT "Currency", Me, "Convert_Currency"
    BDT.RegisterBDT "DateTime", Me, "Convert_DateTime"
End Sub
```

In the RegisterBDT method, the first parameter is the name of the BDT, the second parameter is the object reference, and the third parameter is the name of a conversion routine in the object (a public method).

The BDT controller maintains a list of all BDT names internally along with the object reference and method to call for each.

Deregister a BDT

To deregister one or more BDTs, call the `DeregisterBDT` method as follows:

```
BDT.DeregisterBDT           ' Deregisters all BDTs.
BDT.DeregisterBDT Me       ' Deregisters only the BDTs in the
                             ' specified object.
BDT.DeregisterBDT Me, "Numeric" ' Deregisters only the Numeric BDT in
                             ' the specified object.
```

Deregistering BDTs is useful if you need to release all references to an object so that the object can be destroyed. You can then recreate the object and re-register all BDTs it implements.

Locate the Conversion Routine for a BDT

To locate the conversion routine for a given BDT, use the `GetBDTRoutine` to return the object reference and method name of the conversion routine. The syntax is:

```
Sub GetBDTRoutine(BDTName As String, _
                  ByRef Handler As Object, _
                  ByRef ProcName As String)
```

If the BDT name has not been registered, `Handler` will contain `Nothing` and `ProcName` will contain an empty string on return.

Create a Natural-to-BDT Mapper

The BDT controller calls a Natural to BDT mapper function when the application uses a conversion function and a Natural format is provided, instead of the name of a BDT. The mapper provides the most appropriate BDT to use for each Natural format.

A mapper function must be registered with the BDT controller just as BDTs are registered. In Construct Spectrum, the mapper is implemented as a method of the `StandardBDTs` class and is registered in its `SelfRegister` method as follows:

```
Public Sub SelfRegister(BDT As BDTController)
    ...
    BDT.RegisterNaturalBDTMapper Me, "NaturalBDTMapper"
End Sub

Public Function NaturalBDTMapper(Format As String, _
    Length As Long, _
    Decimals As Integer) As String

    Dim sbdtstring As String

    ' BDT name was not provided. Pick a default BDT name based on the
    ' Natural format.
    Select Case Format
    Case "A": sbdtstring = BDT_ALPHA & ",TRIM=LT"
    Case "B": sbdtstring = BDT_ALPHA
    Case "D": sbdtstring = BDT_DATE
    Case "F": sbdtstring = BDT_NUMERIC
    Case "I": sbdtstring = BDT_NUMERIC
    Case "L": sbdtstring = BDT_BOOLEAN
    Case "N": sbdtstring = BDT_NUMERIC
    Case "P": sbdtstring = BDT_NUMERIC
    Case "T": sbdtstring = BDT_TIME
    End Select

    NaturalBDTMapper = sbdtstring
End Function
```

The `GetBDT` method of the BDT controller returns the name of the BDT used for the given Natural format. Using the mapper in the previous example:

```
Print BDT.GetBDT("D")      ' Prints "Date"
Print BDT.GetBDT("L")      ' Prints "Boolean"
Print BDT.GetBDT("N6")     ' Prints "Numeric"
Print BDT.GetBDT("N6.2")  ' Prints "Numeric"
```

Other Considerations

The following sections contain other considerations when creating BDTs.

Use One Conversion Routine with Multiple BDTs

When you register a BDT, you can use the same function pointer for multiple BDTs. For example:

```
BDT.RegisterBDT "AccountNumber", Me, "Convert_Numbers"  
BDT.RegisterBDT "DeptNumber", Me, "Convert_Numbers"  
BDT.RegisterBDT "GroupNumber", Me, "Convert_Numbers"  
BDT.RegisterBDT "FileNumber", Me, "Convert_Numbers"
```

When the application uses the BDT, the conversion routine checks the BDT name to determine what conversion to perform:

```
Public Sub Convert_Numbers (BDTC As BDTConversion)  
    Select Case BDTC.BDTName  
        Case "AccountNumber"  
            ...  
        Case "DeptNumber", "GroupNumber"  
            ...  
        Case "FileNumber"  
            ...  
    End Select  
End Sub
```

Placement of the Conversion Routine

When you create a new BDT conversion routine, you can add it to an existing class or you can create a new class. Using the client framework, adding the conversion routine to the existing StandardBDTs or CustomBDTs module requires the fewest changes to the code. You need only add a method and then change the SelfRegister method to register the new BDT.

Warning:

When you save the updated version of the class, ensure that you do not overwrite the version in the Construct Spectrum client framework directory, unless you want the update to affect all existing and new projects that point to that class.

If you create a new class, change the InitializeBDTs procedure in the Startup module to instantiate the class and call its SelfRegister method.

Override a Supplied BDT

When the same BDT name is registered with the BDT controller more than once, the last one registered is used. This feature can be used if you want to replace a supplied BDT (conversion routine) with your own. As long as you register your BDT conversion routine last, it will be called instead of the supplied one.

If you are replacing a supplied BDT routine with your own, you can use the `GetBDTRoutine` method to get the pointer to the current BDT routine before registering your own and call the original BDT routine in certain cases:

```
Private m_OldHandler As Object
Private m_OldProcName As String

Public Sub SelfRegister(BDT As BDTController)

    ' Save the pointer to the old routine.
    BDT.GetBDTRoutine "Currency", m_OldHandler, m_OldProcName
    BDT.RegisterBDT "Currency", Me, "Convert_Currency"

End Sub

Public Sub Convert_Currency(BDTC As BDTConversion)

    With BDTC
        Select Case .Conversion
            Case bdtConvertToDisplay
                ' Custom conversion.
                ...
            Case Else
                ' Call the old routine.
                InvokeMethod m_OldHandler, m_OldProcName, Array(BDTC)
            End Select
        End With

    End Sub
```

This example uses the `InvokeMethod` procedure in the Construct Spectrum client framework. The `InvokeMethod` procedure can call any public method of any object by passing in a reference to the object and the name of the method in a string.

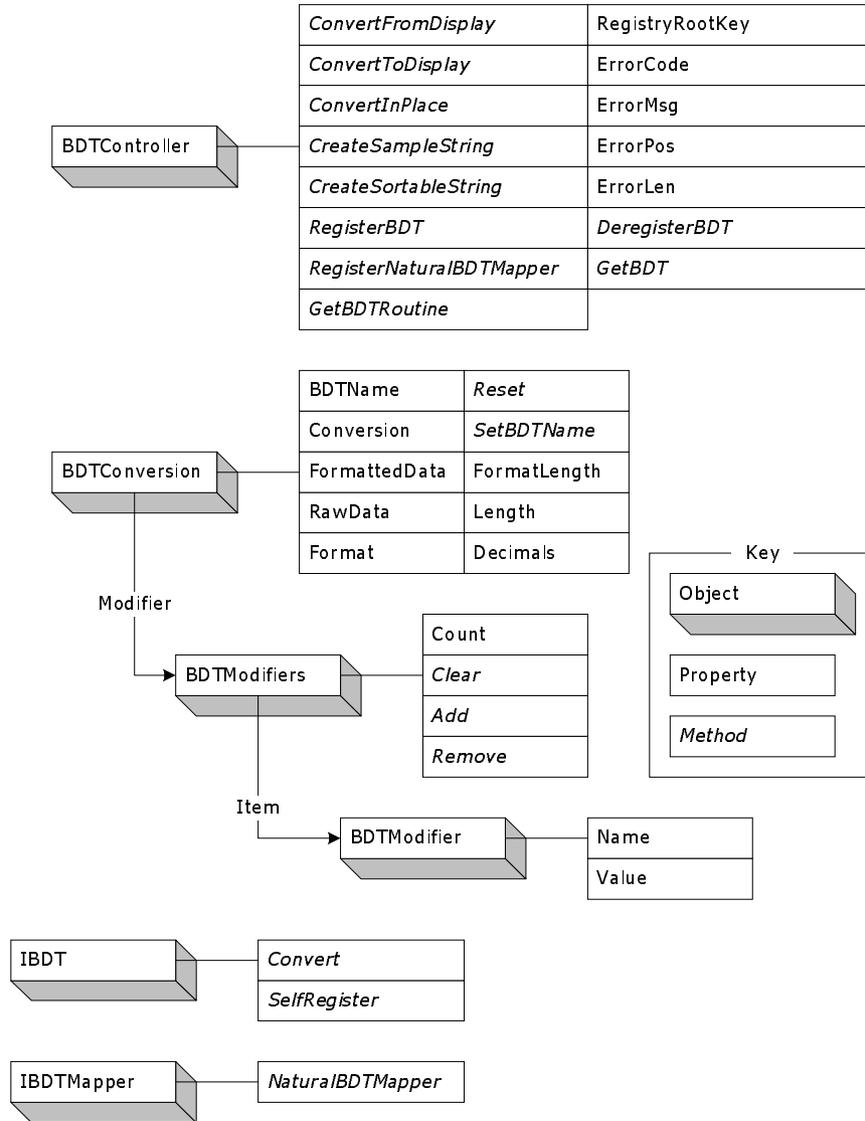
Reference BDTs in Your Application

Each BDT in the `StandardBDTs` and `CustomBDTs` classes of the client framework has an associated named constant in `BDTSupport.bas`. The name of the constant is the same as the name of the BDT except it is in uppercase and prefixed with “BDT_”. Instead of using the BDT name directly through the application, use the named constant. This allows the Visual Basic compiler to check that the BDT is defined in the framework.

When you create your own BDTs, ensure that you add the named constants to `BDTSupport.bas`.

BDTs and the Web Framework

The Construct Spectrum web framework uses objects in the BDTLib6 object library to support BDTs. The following diagram shows these objects, as well as their properties and methods:



Methods and Properties for the Web Framework Objects

Implement BDTs in the Web Framework

In the Construct Spectrum web framework, BDT conversion routines reside inside a Visual Basic class module that implements the IBDT interface. This type of Visual Basic class is called a BDT class, which:

- Implements the IBDT interface
- Contains a BDT conversion routine

The IBDT interface has two methods:

Method	Description
Convert	The BDT controller calls this method to perform a conversion. It passes in a BDTConversion object that contains details about the conversion to be performed.
SelfRegister	In this method, the BDT class must tell the BDT controller the names of the BDTs that it implements.

The following sections describe the steps to implement a BDT in a web framework.

Register BDTs in the Web Framework

When an application needs to use a BDT, it calls the BDT controller and specifies the name of the BDT it wants to use (such as Boolean). The BDT controller knows how to locate and call the BDT conversion routine by registering the BDT class with the BDT controller.

This allows the BDT controller to associate the name of a BDT with a BDT class. When the BDT controller needs to call the conversion routine, it creates an instance of the BDT class, gets a reference to the class' IBDT interface, and finally calls the Convert method.

Use one of the following techniques to register BDT classes with the BDT controller:

- 1 Register BDT classes using the Windows Registry
- 2 Explicitly register BDT classes

The following sections describe each of these options.

Register BDT Classes Using the Windows Registry

The first technique for registering a BDT class is to use the Windows Registry to list all of the BDT classes installed on the PC. For the standard BDTs supplied with the web framework, the following excerpt from the Registry shows how this is done:

```
HKEY_LOCAL_MACHINE
  Software
    Software AG
      Business Data Types
        Alpha
          ProgID = StandardBDTs6.BDTAlpha
        AlphaMultiline
          ProgID = StandardBDTs6.BDTAlphaMultiline
        Boolean
          ProgID = StandardBDTs6.BDTBoolean
        ...
```

Notice the names of the BDTs under the Business Data Types key. Each BDT key contains a ProgID string value that tells the BDT controller the programmatic ID (progID) of the class that implements the BDT conversion routine. Knowing the progID allows the BDT controller to create an instance of the BDT class.

The following example shows how the BDT controller locates and calls the BDT conversion routine for the Boolean BDT:

- 1 The BDT controller looks up the ProgID value under the Boolean key and finds the name "StandardBDTs6.BDTBoolean".
- 2 It uses the Visual Basic CreateObject function to create an instance of this class. By using the facilities of COM, CreateObject loads the ActiveX DLL that implements the BDT class (StandardBDTs6.dll) and creates an instance of the BDT class (BDTBoolean).
- 3 The BDT controller calls the SelfRegister method in the IBDT interface implemented by the BDT class. For example:

```
Private Sub IBDT_SelfRegister(BDT As BDTController)
    BDT.RegisterBDT "Boolean", Me
End Sub
```

The BDT class calls the BDT controller's RegisterBDT method, passing in the name of the BDT and an object reference to itself.

- 4 The RegisterBDT method in the BDT controller stores the BDT name and the object reference in an internal table. The BDT controller uses this table as a cache to store object references so it doesn't have to create a new instance of the BDT class each time the application uses the BDT.
- 5 The BDT controller now has a reference to an instance of the BDT class, and calls the Convert method in the class' IBDT interface.

The next time the application uses the Boolean BDT, the BDT controller looks at its internal table first and finds the BDT name and object reference. It can then call the Convert method immediately without having to perform the previous steps.

Placing BDT classes in an ActiveX DLL and using the Windows Registry to list them has the following advantages:

- BDT classes can be shared by many applications. By implementing BDT classes in an ActiveX DLL, they can be developed, tested, and enhanced separate from any application.
- An application that needs to use BDTs does not have to explicitly register all of the BDTs that it will use. The BDT controller simply locates and loads the BDT class containing the conversion routine dynamically at runtime whenever it is needed.
- BDTs can be added to a PC by installing the ActiveX DLL that contains them and then registering the new BDTs to the BDT controller by adding Registry keys under HKEY_LOCAL_MACHINE\Software\Software AG\Business Data Types.

Explicitly Register BDT Classes

The second technique for registering a BDT class is to explicitly call the BDT controller's RegisterBDT method in your application's startup code. For example:

```
Public Sub Main

    ...

    ' Register the BDTs used by the application.
    RegisterBDT "Phone", New BDTPhone
    RegisterBDT "ZipCode", New BDTZipCode
    RegisterBDT "UPC", New BDTUPC

    ...

    ' Show the application's main form.
    frmMain.Show

End Sub
```

In this example, the RegisterBDT method is called. For each BDT, the name of the BDT and a reference to an instance of the BDT class that implements the BDT conversion routine is passed. The BDT controller stores the BDT name and reference in its internal table in a similar way as it does with the Windows Registry.

Because the BDT controller is a global, multi-use object, it can be invoked with its properties and methods (such as RegisterBDT) as if they were global functions.

Explicitly registering BDT classes in your application has the following advantages:

- You can create private BDTs that are available only inside your application. They can be developed, tested, and enhanced with your application.
- Extra keys do not need to be added to the Windows Registry to tell the BDT controller about the BDTs.

BDT Conversion Object

In the Construct Spectrum web framework, BDT conversion routines reside inside BDT classes. The following table describes the properties and methods of the BDT Conversion object. In the examples used in the table, assume the following call was made:

```
strHours = ConvertToDisplay(dblHours, _
                           "Numeric,ZERO=OFF,ROUND=2,STRICT=ON", _
                           "N3.2")
```

Property or Method	Description
Conversion	<p>Tells the conversion routine what type of conversion to perform. Can be one of the following constants:</p> <pre>bdtConvertToDisplay bdtConvertFromDisplay bdtCreateSampleString</pre>
FormattedData and RawData	<p>Should be one of the following:</p> <ul style="list-style-type: none"> • When Conversion = bdtConvertFromDisplay, the conversion routine reads the value in FormattedData, converts it into a Visual Basic data type, and assigns the new value to RawData. • When Conversion = bdtConvertToDisplay, the conversion routine reads the value in RawData, formats it for display, and assigns the formatted string value to FormattedData. • When Conversion = bdtCreateSampleString, the conversion routine assigns a sample string to FormattedData. For example: <pre>With BDTC Select Case .Conversion Case bdtConvertFromDisplay .RawData = cvtToRaw(.FormattedData) Case bdtConvertToDisplay .FormattedData = cvtToDisp(.RawData) Case bdtCreateSampleString .FormattedData = createSample() End Select End With</pre>
Modifier.Count	<p>Returns the number of modifiers specified by the caller. In the example, Modifier.Count returns 3.</p>
Modifier	<p>Returns the value of a specific modifier or can be used to enumerate the modifiers used. In the example:</p> <pre>With BDTC Print .Modifier("ZERO") ' Prints "OFF" Print .Modifier(1) ' Prints "ZERO" Print .Modifier(2) ' Prints "ROUND" Print .Modifier(.Modifier(1)) ' Prints "OFF" End With</pre>

Property or Method	Description (continued)
Note:	Modifier names, such as ZERO or ROUND, are passed to the BDT conversion routine in uppercase. You do not have to use case-sensitive string comparisons when checking which modifiers were used.
FormatLength	Returns the Natural format string used in the call. In the example, FormatLength returns N3.2.
Format, Length, and Decimals	Returns the format, length, and decimal portions, respectively, of the Natural format string used in the call. In the example, Format contains N, Length contains 3, and Decimals contains 2.
BDTName	Returns the name of the BDT from the call. In the example, BDTName contains Numeric.
Note:	BDT names are passed to the BDT conversion routine with the capitalization used when the BDT was registered. For example, if RegisterBDT was called to register the BDT mIxEdCase, and then ConvertToDisplay was called for the BDT MixedCase, BDTC.BDTName contains mIxEdCase.
SetBDTString	Changes the BDT name and modifiers in the BDTConversion object.
ErrorCode, ErrorMsg, ErrorPos, and ErrorLen	Contain error information. The conversion routine should assign values to these properties if a conversion error occurred.

Create the BDT Class

A different class module is usually used for each BDT. You can also group a set of related BDTs in a class module to share conversion routines or code. For example, a BDT called PartNumber might be implemented by a class called BDTPartNumber. You can name your class as desired.

The basic structure of a BDT class when implemented in Visual Basic is:

```
Option Explicit

Implements IBDT

Private Sub IBDT_Convert(BDTC As BDTConversion)

    Select Case BDTC.Conversion
    Case bdtConvertToDisplay
        BDTC.FormattedData = ...
    Case bdtConvertFromDisplay
        BDTC.RawData = ...
    Case bdtCreateSampleString
        BDTC.FormattedData = ...
    Case bdtCreateSortableString
        BDTC.FormattedData = ...
    End Select

End Sub

Private Sub IBDT_SelfRegister(BDT As BDTController)
    BDT.RegisterBDT "<BDT name>", Me
End Sub
```

The BDT conversion routine is implemented in the `IBDT_Convert` procedure. It uses the properties of the `BDTConversion` object to determine what type of conversion to perform (convert to display, convert from display, create sample string, or create sortable string), to get information about the modifiers used, the Natural format specified, and to return the converted value.

You can examine the BDT classes in the StandardBDTs sample project to see how these properties and methods are used in BDTs.

Other BDT Controller Methods

To deregister one or more BDTs, call the DeregisterBDT method as follows:

```
DeregisterBDT           ' Deregisters all BDTs.
DeregisterBDT Me       ' Deregisters only the BDTs in the
                        ' specified object.
DeregisterBDT Me, "Numeric" ' Deregisters only the Numeric BDT in
                        ' the specified object.
```

Deregistering BDTs is useful if you need to release all references to an object so that the object can be destroyed. The object can then be recreated and re-registered with all the BDTs it implements.

If the conversion routine for a given BDT needs to be located, use the GetBDTRoutine to return the object reference of the BDT class that implements the conversion routine. The syntax is:

```
Sub GetBDTRoutine(ByVal BDTName As String, ByRef Handler As IBDT)
```

If the BDT name has not been registered, Handler contains Nothing on return.

Create a Natural-to-BDT Mapper

The BDT controller calls the Natural to BDT mapper function when the application uses a conversion function and provides the Natural format, instead of the name of a BDT. The mapper provides the most appropriate BDT to use for each Natural format.

A mapper function must be registered with the BDT controller as BDTs are registered. Using the Windows Registry technique, use the following Registry key:

```
HKEY_LOCAL_MACHINE
  Software
    Software AG
      Business Data Types
        NaturalBDTMapper
          ProgID=StandardBDTs6.NaturalBDTMapper
```

You can also explicitly register the mapper function using the RegisterNaturalBDTMapper function:

```
Public Sub Main
    ...

    RegisterNaturalBDTMapper New NaturalBDTMapper
    ...

    ' Show the application's main form.
    frmMain.Show
End Sub
```

This class must implement the `IBDTMapper` interface. The following example shows how the mapper function is implemented in the `StandardBDTs` sample project:

```
Option Explicit

Implements IBDTMapper

Private Function IBDTMapper_NaturalBDTMapper(Format As String, _
                                             Length As Long, _
                                             Decimals As Integer) _
                                             As String

    Select Case Format
    Case "A"
        IBDTMapper_NaturalBDTMapper = "Alpha,TRIM=T"
    Case "B", "C"
        IBDTMapper_NaturalBDTMapper = "HexBytes"
    Case "D"
        IBDTMapper_NaturalBDTMapper = "Date"
    Case "F", "I", "N", "P"
        IBDTMapper_NaturalBDTMapper = "Numeric"
    Case "L"
        IBDTMapper_NaturalBDTMapper = "Boolean"
    Case "T"
        IBDTMapper_NaturalBDTMapper = "DateTime"
    End Select

End Function
```

The `NaturalBDTMapper` method must return the most appropriate BDT to use for the given Natural format/length specified.

Once a mapper function has been registered, the `GetBDT` method of the BDT controller returns the name of the BDT used for the Natural format. Using the previous mapper:

```
Print GetBDT("D")      ' Prints "Date"
Print GetBDT("L")      ' Prints "Boolean"
Print GetBDT("N6")     ' Prints "Numeric"
```

Create One BDT Class with Multiple BDTs

One BDT class can implement BDT conversion routines for multiple BDTs. For example:

```
Private Sub IBDT_SelfRegister(BDT As BDTController)
    RegisterBDT "AccountNumber", Me
    RegisterBDT "DeptNumber", Me
    RegisterBDT "GroupNumber", Me
    RegisterBDT "FileNumber", Me
End Sub
```

When the application uses the BDT, the conversion routine checks the BDT name as follows to determine what conversion to perform:

```
Public Sub IBDT_Convert(BDTC As BDTConversion)

    Select Case BDTC.BDTName
    Case "AccountNumber"
        Select Case BDTC.Conversion
            ...

    Case "DeptNumber", "GroupNumber"
        Select Case BDTC.Conversion
            ...

    Case "FileNumber"
        Select Case BDTC.Conversion
            ...

    End Select
End Sub
```

DEBUGGING YOUR CLIENT/SERVER APPLICATION

This chapter describes how to debug client/server applications created using Construct Spectrum.

The following topics are covered:

- **Overview**, page 162
- **Types of Errors**, page 165
- **Generating Debug Data**, page 167
- **Running Spectrum Dispatch Services Online**, page 174
- **Using Natural Debugging Tools**, page 175
- **Debugging Tools on the Client and Server**, page 177
- **Troubleshooting**, page 183

For related information, see:

- *Construct Spectrum Messages* for a list of each Construct Spectrum error with possible causes and solutions.
- Natural documentation
Refer to the Natural documentation for information on the Natural Debugging facility.
- *Microsoft Visual Basic Programmer's Guide*
Refer to the Debugging chapter for information on the debugging environment for Visual Basic applications, including the kinds of errors, different modes, and the debugging tools available.

Overview

Client/server applications are more complex than traditional, single-platform applications. Multiple computers are connected together, requiring a communication layer that opens the door for new types of errors. In client/server applications, errors can occur in more than one place. Server components must be developed as callable routines without a user interface. Data values have different internal representations on the client and on the server. All of these distributed computing issues for client/server applications allow more room for errors.

Because it is not always apparent where the errors occur, debugging client/server applications can be more difficult than debugging single-platform applications. Errors may occur within the client software, the server software, the network layer, or a combination of these. To simplify the debugging process, the client framework provides tools and procedures you can use to debug your applications.

Communication Errors

Communication errors occur during a remote call from the client application to the subprogram. Each remote call involves many individual software components and data files. Some software components run on the client, while others run on the server. With so many components and different platforms involved in every call, the potential for error is greater than in non-client/server applications. A high-level list of the components involved in a remote call includes:

- application service definitions
- client application
- EntireX Broker
- EntireX Broker stub
- Entire Net-Work
- library image files
- Spectrum Dispatch Client
- Spectrum dispatch service
- Spectrum security service
- subprogram proxy
- subprogram

Communication Error Handling

Because the client application initiates every remote call, it is also necessary to transfer back to the client application any error that does occur. The client application takes corrective action or it displays the error message to the user.

Error messages return to the client application in all but the most severe error situations. The Spectrum Dispatch Client makes the error details available to the client application through its error properties `ErrorSource`, `ErrorNumber`, `ErrorMessage`, and `ErrorValue`. If `DisplayErrors` is set to `True`, the Spectrum Dispatch Client will also display the error message in a message box.

Severe error situations that prevent the error message from being returned to the client application include:

- An interruption in the Entire Net-Work communication between client and server.
- EntireX Broker ends.
- EntireX Broker times out during the subprogram execution.
- The subprogram or a Spectrum service ends the Spectrum dispatch service.

If a message cannot be returned to the client, it is written to the communication log.

For a complete list of communication errors and how to resolve them, see *Construct Spectrum Messages*.

Traditional Debugging Tools

In Natural applications, logic errors are diagnosed using one of two techniques:

- Temporarily add `WRITE`, `DISPLAY`, or `INPUT` statements to show the contents of variables and the execution sequence of the program logic.
- Use the Natural Debugging facility to step through the code and the variable contents.

When a client application invokes Natural services, these traditional debugging tools are not available. Both of these traditional debugging techniques pause the execution of the program for user input. However, because dispatch services run in batch mode by default, no I/O statements are possible. Nevertheless, the Spectrum dispatch service may have reported Natural runtime errors or unexpected values back to the client. Each of these requires investigation.

Construct Spectrum Debugging Tools

The debugging tools supplied with Construct Spectrum allow you to:

- Save the data for client requests to a Natural library on the server. This data can then be used to recreate the request on the server and run it online. You can then use all of the traditional Natural debugging facilities to diagnose problems. For information, see **Generating Debug Data**, page 167, and **Using Natural Debugging Tools**, page 175.
- Use output statements, including WRITE, PRINT, and DISPLAY, in your Natural subprograms to write data to the Natural source buffer and save the source buffer to a Natural library. You can then examine this data after the call returns to the client. Use this technique if you do not need to run client requests online. For information, see **Generating Debug Data**, page 167, and **Using Natural Debugging Tools**, page 175.
- Examine the data transmitted between the client and the server. For information, see **RequestProperty Property**, page 186.
- Examine the data expected by a subprogram proxy. Use this feature if you suspect the data formats used by the client and server components differ. For information, see **Diagnostics Window**, page 177.

Types of Errors

Errors that are returned by the Spectrum Dispatch Client (SDC) fall into two categories: runtime errors and communication errors. A third category, Spectrum system messages, are not returned to the SDC. These messages must be viewed in the Spectrum Administration subsystem.

While most errors can be fixed on the client, others must be fixed on the server. Construct Spectrum provides methods that help you track the origin and reason for errors. These methods allow you to determine what needs to be fixed and where the repair must be made. The types of errors you will encounter while designing your Construct Spectrum client/server application are:

- Visual Basic runtime errors
- Communication errors
- Natural runtime errors
- Construct Spectrum-related errors
- Errors that do not return an error message

This chapter describes the Construct Spectrum tools and procedures to help debug these last three types of errors: Natural runtime errors, Construct Spectrum-related errors, and errors that do not return an error message.

Visual Basic Runtime Errors

Visual Basic runtime errors can be trapped by using the Visual Basic On Error statement. These errors are the easiest to resolve because they occur in your Construct Spectrum application in Visual Basic and allow you to use the Visual Basic-provided debugging features to pinpoint the problem. Runtime errors are always caused by programming errors in your code or by some problem related to the client environment, such as a missing file.

Note: You also code business validations in your Visual Basic maintenance objects to raise runtime errors when a validation fails. The Construct Spectrum client framework traps these errors and displays them as pop-up messages attached to a GUI control.

For more information about validating your data, see **Validating Your Data**, page 261, *Construct Spectrum SDK for Client/Server Applications*. For a complete list of runtime errors and how to resolve them, see *Construct Spectrum Messages*.

Communication Errors

Communication errors occur when there are problems establishing a connection to the server. These errors are returned by the Spectrum Dispatch Client's error properties. If `ErrorSource` contains "ETB", a communication error has occurred.

For more information, see *Construct Spectrum Messages*. Also refer to the EntireX Broker Error Reference documentation.

Natural Runtime Errors

Natural runtime errors may occur in your subprograms. These errors are always returned to the client application by the Spectrum Dispatch Client. When the client application uses the `CallNat` method of the Spectrum Dispatch Client's dispatcher object to call a remote subprogram and the `CallNat` is returned, check the dispatcher object's error properties. If `ErrorSource` contains "NAT", a Natural runtime error has occurred.

Construct Spectrum-Related Errors

These errors are returned by the Spectrum Dispatch Client. If `ErrorSource` contains "SPE", a Construct Spectrum-related error has occurred.

For more information, including a complete list of Construct Spectrum errors and how to resolve them, see *Construct Spectrum Messages*.

Errors that Do Not Return an Error Message

These errors do not return an error message, but they can cause your program to behave unexpectedly.

Generating Debug Data

Generating debug data is a service provided by the Spectrum dispatch service. The Spectrum dispatch service automatically saves the source area contents to the Natural system file. The source area's contents are generated based on values found in the trace options set on the client. Values assigned in your user record determine the location and name of the stored debug data.

For information, see **Specify Where to Save Debug Data**, page 172.

Note: If you intend to use the Trace function, you must install Construct Spectrum with printer 2 and 3 assigned to batch. For more information, see *Construct Spectrum and SDK Installation Guide for Mainframes*.

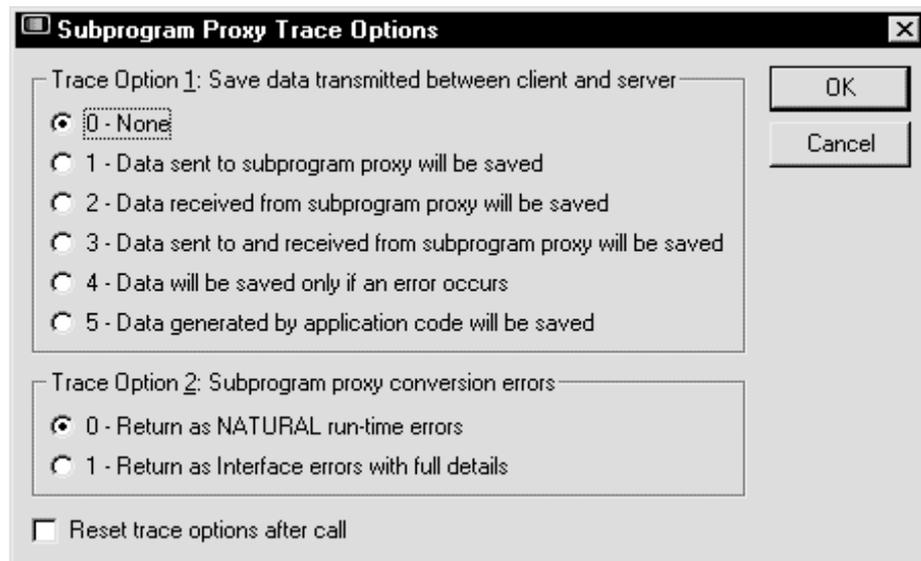
Save Parameter and Debug Data

For each request handled by the Spectrum dispatch service, it is possible to save the parameters passed in or out of the subprogram proxy. These parameter values are saved to a text member within the Natural System file. It is also possible to save data that the application code generates into the source area.

The Spectrum Dispatch Client and Spectrum dispatch server support a trace option that determines how much debug data is saved during a remote CallNat. The trace option is set on the client before issuing the CallNat method. The Spectrum dispatch server then examines the trace option during the CallNat to determine how much data to save.

Set Trace Options

- To set a trace option:
 - 1 Place a break point in the Visual Basic code just before the Dispatcher.CallNat method as follows:
 - For a maintenance dialog, in the InvokeRemoteMethod function of the Visual Basic maintenance object.
 - For a browse dialog, in the CallDBLayer function of the BrowseBase class.
 - 2 Enter “SetTraceOptions Dispatcher” in the Visual Basic Debug window, where Dispatcher is the reference variable of a Dispatcher object.
The Trace Options window is displayed:



Remote Dispatch Server Trace Options Window

- Use this window to set trace option 1 or 2.
The following sections describe each of these options.

Trace Option(1)

Trace Option(1) controls how you save data to the Natural system file.

Trace Option(1) causes the Spectrum dispatch service to issue an END TRANSACTION command. As a result of the END TRANSACTION, the current data is saved to the debug file.

You can assign Trace Option(1) one of the following values:

Value	Result
0	No tracing. Nothing is written to the Natural system file.
1	Spectrum dispatch service saves only data received from the client and sent to the subprogram proxy and writes it to the system file.
2	Spectrum dispatch service saves only data received from the subprogram proxy and returned to the client and writes it to the system file.
3	Spectrum dispatch service saves both the data received from and the data returned to the client.

Value	Result (continued)
4	Server saves data only when a Natural runtime error occurs in the server application. The data saved will be the contents of the subprogram proxy parameters at the time the error occurred. These values may differ from the values sent to the subprogram proxy.
5	Any data that the subprogram proxy or the subprogram writes to the Natural source area is saved.

Note: When using trace option(1) = 3 and a subprogram that clears the source area is called, data received from the client is lost. Only data transmitted to the client is saved. In this case, use value 1 to save data received from the client.

If a subprogram writes data to a source area, it is then saved by the dispatcher. To write to the source area, the application subprograms must contain a printer definition, such as `DEFINE PRINTER (DEBUG=1) OUTPUT 'SOURCE'`. The subprogram can then write out debug data using Natural `DISPLAY`, `WRITE`, and `PRINT` statements.

By default, all generated subprogram proxies contain a printer definition allowing debug data to be written to the source area. This eliminates the need for you to place this code in the generated proxy if you need to allow generation of application debug data from inside the generated proxy routine.

To write debug data to the source area, you will write code to the beginning of the module that will write the debug data (the `START-OF-PROGRAM` user exit if using Construct-generated code). To view a sample of this default and tailored code, see **Create Debug Data**, page 170.

For more information about the subprogram proxy, see **Using the Subprogram-Proxy Model**, page 103.

Create Debug Data

The following example shows code samples of how to include debug information in your applications and code samples of what you might see returned.

```
IF *LEVEL EQ 1 THEN
    DEFINE PRINTER(DEBUG=1) OUTPUT 'SOURCE'
END-IF
FORMAT(DEBUG) PS=0 LS=250 SG=OFF ZP=OFF AD=Z
```

To create better, more readable debug information, the DEFINE PRINTER statement should be bounded by an IF condition that does not execute in application subprograms. The DEFINE PRINTER statement is still required in each module that is expected to perform WRITE statements in the source area. However, based on the IF statement, the code is never executed; it only exists to allow for the definition of a logical printer name for the debugging target. By disallowing execution of the DEFINE PRINTER statement in application code, the print queue remains open across all subprograms using it. Each DEFINE PRINTER closes the print queue. While no information is lost, a new page header is forced each time, causing less readable debug data to be produced.

Example of debug code in a series of subprograms

```
Subprogram 1 (SUBP1)
WRITE *PROGRAM

Subprogram 2 (SUBP2)
WRITE *PROGRAM

Subprogram 3 (SUBP3)
WRITE *PROGRAM

Subprogram 4 (SUBP4)
WRITE *PROGRAM
```

Results when debugging using the IF statement

```
SUBP1
SUBP2
SUBP3
SUBP4
```

Results when debugging without using the IF statement

```
*/  
Page 1
```

```
SUBP1  
*/  
Page 1
```

```
SUBP2  
*/  
Page 1
```

```
SUBP3  
*/  
Page 1
```

```
SUBP4
```

Use any output statement to generate information into the source area:

```
WRITE (DEBUG) NOTITLE 'Prompt 1:' #VAR1
```

OR

```
PRINT (DEBUG) NOTITLE 'Prompt 2:' #VAR2
```

OR

```
DISPLAY (DEBUG) NOTITLE 'Prompt 3:' #VAR3
```

Note: The Natural subprograms called from the client execute in batch Natural processes. The output will go to the printer or terminal unless you redirect the output to the source area using the DEFINE PRINTER statement.

For more information about using the debug data saved with Trace Option(1), see **Using Natural Debugging Tools**, page 175.

Trace Option(2)

This option controls how the generated subprogram proxies handle runtime errors. It works in conjunction with the Generate Trace Code field of the subprogram proxy specification. It is used to help uncover the cause of data format and data length incompatibilities between the client and the server.

You can assign Trace Option(2) one of the following values:

Value	Result
0	All errors occurring within the subprogram proxy are handled as normal Natural runtime errors. As a result, control does not return to the Spectrum dispatch service and the current Error Transaction is invoked. The default error transaction returns a message to the client and restarts the Spectrum dispatch service.
1	Format conversion errors are trapped in an ON ERROR block of the generated subprogram proxy. A Natural runtime error does not occur for these errors, so the Spectrum dispatch service resumes control after the ON ERROR processing. If this option is used in conjunction with the Generate Trace Code parameter of the subprogram proxy, the field name and data values that triggered the error are returned to the Spectrum dispatch service and transferred to the client.

Tip: Using Trace Option(2)=0 while running a Spectrum dispatch service online can be an effective way of determining runtime problems.

Specify Where to Save Debug Data

Settings in your user record determine where debug information is stored and how file names are determined. User records are maintained in the Spectrum Administration subsystem.

For more information on the Spectrum Administration subsystem, see **Overview of the Spectrum Administration Subsystem**, page 19, *Construct Spectrum Administration*.

For more information about user records, see **Defining Groups Using Natural Security**, page 80, and **Defining Users Using Spectrum Security**, page 83, *Construct Spectrum Administration*.

Access the Maintain User Table Panel

- To access the Maintain User Table panel:
 - 1 Enter “SA” in the Function field on the Construct Spectrum Administration Subsystem main menu.
The System Administration main menu is displayed.
 - 2 Enter “MM” in the Function field on the System Administration main menu.
The System Administration Maintenance menu is displayed.
 - 3 Enter “US” in the Function field on the System Administration Maintenance menu.
The Maintain User Table panel is displayed:

```

BSUS__MP          Construct Spectrum Administration Subsystem          BSUS__11
Apr 14              Maintain User Table                               1:45 PM

Action (A,B,C,D,M,N,P)  _
User ID.....: SYSTEM__
Password.....:

Name.....: DEFAULT SYSTEM USER (NO PASSWORD)_____

Debug Library.....: SYSSPEC_
Debug Filename.....: U ('Timestamp; 'User ID)

Preferred Language.....: 01

Groups.....: SYSTEM_  _____ *
                _____
                _____
                _____

Direct Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
confm help retrn quit    flip pref                main flip pref
main
User flip pref                main
User SYSTEM displayed successfully
    
```

Maintain User Table Panel

Debug Library is the name of the Natural library where the debug file is saved. If no library is specified or if user information is provided by Natural security, the library name defaults to the current user ID. Debug Filename can be:

Value	Result
T	File name is determined by concatenating “T” with the current time value.
U	File name is the same as the current user ID.

To view the generated debug members, use the Natural EDIT or LIST command.

Running Spectrum Dispatch Services Online

Instead of using trace option (1) with assigned value 5 (which writes to the source area), you can use a Natural session to initiate the service online. Initiating a dispatch service from a Natural session allows I/O to the terminal. This method is similar to the debugging method discussed in **Debugging Tools on the Client and Server**, page 177, but the Natural session running the dispatch service cannot perform any other tasks.

This type of dispatch Service stays active and locks control of your online Natural session until you send it a shutdown request or until it times out because of server non-activity.

➤ To start a server online:

- 1 Invoke Natural using the SYSSPEC profile.
- 2 Enter the following command at the Next prompt in the SYSSPEC library:

```
`START servicename`
```

Note: You can also specify the Natural startup parameters in a Natural profile. For more information, see the *Construct Spectrum and SDK Installation Guide for Mainframes*.

Use the INPUT Statement as a Debugging Tool

If you decide to run the dispatch service online, use the INPUT statement for debugging. The INPUT statement allows you to interrupt and restart the execution of code. Use these interruptions to generate a printed copy of the INPUT statement or to copy the INPUT statement to the source area with a %C command.

If application tracing is set (trace option 1 = 5), the dispatch service writes the inputs copied to the source area into the designated debug source member.

Tip: To guarantee that others using the same services do not generate terminal I/O when running the service online, bound your debug statements with `IF *USER = 'youruserID' THEN and END-IF`. As long as 'youruserID' is set to the *USER of the session in which the dispatch service has been initiated online, only your online dispatch service generates messages to the terminal.

Tip: When running a server online, you can shut it down using the Broker console shutdown command. It is best to use a unique server class/server name/service to ensure that you do not shut down another server inadvertently.

Using Natural Debugging Tools

Debugging client/server applications can be difficult because of their distributed nature. To make the debugging process easier, Construct Spectrum includes an invoke subprogram proxy function that simulates client calls. Using this function lets you reproduce problems like runtime errors without the added complexity of communication between the client and server.

To help you use Natural to simulate client calls, the client component of generated applications can tell the server application component that data being transmitted must be saved on the server. Using this server-based data to drive the server component allows you access to Natural debugging techniques, such as embedded INPUT or WRITE statements. In addition, by executing the server component locally on the server machine, you can use the Natural Debugging Facility.

For more information, see **Generating Debug Data**, page 167.

For information on using the Natural Debugging facility, see the Natural Utilities documentation.

Invoke Subprogram Proxies Online

Once the debug data exists on the server, use the Invoke Proxy function in the Construct Spectrum Administration subsystem to invoke the same subprogram proxy that the client attempted to call. The function uses the debug data to perform the function the client originally requested. Once execution of the target proxy begins, one of two things can happen:

Result	Response
A runtime error occurs	The system traps this error and presents it as a message on the Invoke Proxy panel.
No runtime error occurs	The Invoke Proxy panel displays a message indicating that execution of the proxy completed successfully.

If you added debug code to the target proxy and subprogram, the system is able to present the terminal output of these statements.

If the problem is not a runtime error, use the Natural Debugging facility to place break and watch points in the target code. You can monitor these points using the Invoke Proxy function to examine variable contents and line-by-line execution.

Access the Invoke Proxy Function

The Invoke Proxy function is one of the options accessible through the Application Administration main menu. For a description of how to access the Construct Spectrum Administration subsystem, see **Invoking the Spectrum Administration Subsystem**, page 31, *Construct Spectrum Administration*.

- To access the Invoke Proxy function:
- 1 Enter “AA” in the Function field on the Construct Spectrum Administration Subsystem main menu.
The Application Administration main menu is displayed.
 - 2 Enter “IP” in the Function field on the System Administration main menu.
The Invoke Proxy panel is displayed:

```

BSSIDBGP          Construct Spectrum Administration Subsystem          BSSIDBG1
May 08                                Invoke Proxy                                09:03 AM

                                     Debug Library: DEVDG___
                                     Member  : DEVJM___
                                     DBID   : ___
                                     FNR    : ___

Direct Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      help  retrn quit          flip                                main

```

Invoke Proxy Panel

Using this panel, you can activate the Natural Debugging facility or put trace statements (INPUT, WRITE, DISPLAY, or PRINT) in the Natural server modules to help diagnose the error.

By default, the system uses the FUSER or FNAT defined for the session when retrieving the debug data, depending on the library name. To use an alternate FUSER or FNAT, specify the values in the DBID and FNR fields.

Tip: Manually change the data in the debug member to generate a runtime error. Use this test either to ensure that runtime situations can be handled properly by the system or to force execution of code that occurs only in the case of runtime errors.

Debugging Tools on the Client and Server

The following sections describe debugging tools you can use on the client and tools you can use on the server.

Diagnostics Window

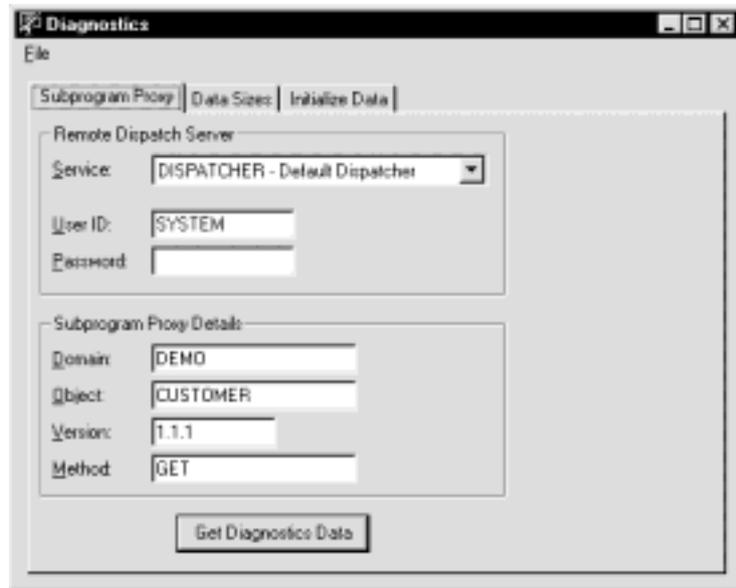
Application developers use the Diagnostics window during application development to diagnose parameter alignment problems between the client and server.

When invoking remote Natural subprograms from a client application, the parameters must match in both size and format on both sides of the call. The Diagnostics window obtains information about the remote subprogram. By examining the Dispatcher.RequestProperty array after invoking Dispatcher.CallNat, the Spectrum Dispatch Client can give you information about the local call.

The following table summarizes the information:

Returned by Diagnostics Program	Equivalent SDC Property
Number of level 1 blocks in the parameter data of the subprogram.	Dispatcher.RequestProperty _ ("Request.DataAreas")
Name of each block (corresponds to the level 1 field or structure name).	Dispatcher.RequestProperty _ ("Request.DataArea(2)").Name
Expanded size of data in each block.	Dispatcher.RequestProperty _ ("Request.DataArea(2)")._ PackedDataLength
Total size of the parameter data.	Not applicable.
Image of the initialized parameter data.	Dispatcher.RequestProperty _ ("Request.DataArea(2)")._ PackedData

The following example shows the Diagnostics window:

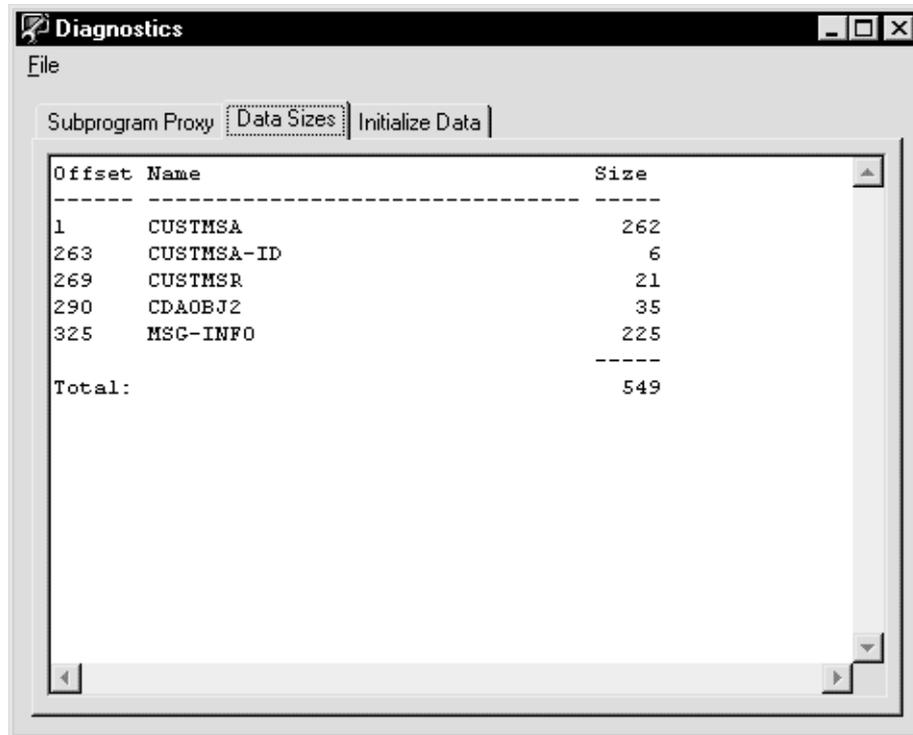


Diagnostics Window, Subprogram Proxy Tab

Use this window to simulate a CallNat by providing all the Spectrum dispatch service parameters necessary to do the CallNat.

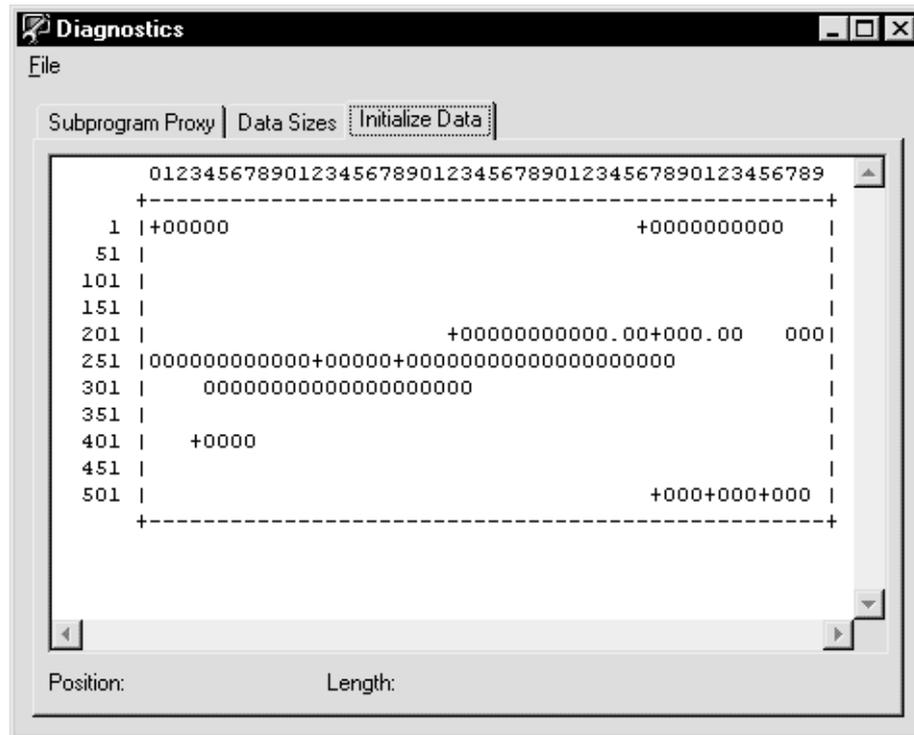
- To use the Diagnostics window:
- 1 Enter your user ID and password.
These values are required for all remote requests.
 - 2 Enter the domain name, object name, version number, and method name to identify the subprogram you want to call.
You can obtain this information from the Construct Spectrum Administration subsystem or from the library image file (LIF) for your application.
 - 3 Click Get Diagnostics Data to submit the request.
If the request is successful, the Data Sizes tab shows information about the level 1 blocks and the Initialize Data tab shows an image of the initialized parameter data.

The following examples show the tabs for the DEMO/CUSTOMER/1.1.1/GET request:



Offset	Name	Size
1	CUSTMSA	262
263	CUSTMSA-ID	6
269	CUSTMSR	21
290	CDAOBJ2	35
325	MSG-INFO	225
Total:		549

Diagnostics Window, Data Sizes Tab



Diagnostics Window, Initialize Data Tab

The Initialize Data tab shows the expanded version of the parameter data. If you highlight a portion of text, the position and length of the highlighted section are shown at the bottom of the window. You can use this information to help determine parameter alignment problems. In the example above, notice the first line of text on the right side of the window that reads “+0000000000”. If you know something about the format of the parameter data, you can infer that this value represents the PHONE-NUMBER field, an N10 field, in the Customer object. You can then compare the format of this data to the data sent to the server.

Translations Program

Construct Spectrum uses its own ASCII/EBCDIC translation tables to convert data when the client and server use different character sets. In most cases, you do not need to know anything about these tables. However, when your subprograms send or receive non-printable characters in alpha fields (format A), you may want to know what the translation tables do with those characters.

The Translations program shows you exactly how each byte value is translated from one character set to the other. The translation tables group the 256 characters in each character set into three sets:

Set	Description
Printable characters	Characters exist in both character sets; there is a well-defined mapping from one character set to the other.
Preserved characters	Characters have no corresponding character in the other character set and their byte values are the same in both character sets. For example, character 0 in ASCII is also character 0 in EBCDIC, and 255 is 255.
Altered characters	Characters have no corresponding characters in the other character set and their byte values are different in both character sets because the byte value is already being used by a printable character in one of the sets.

The Translations program uses colors to identify these three sets of characters. The following example shows the Translation Mappings window in shades of gray:

Translation Mappings																
EBCDIC to ASCII																
<input type="checkbox"/> Printable <input checked="" type="checkbox"/> Preserved <input type="checkbox"/> Altered																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
2	60	7F	81	82	83	84	85	86	87	88	89	91	92	93	94	95
3	96	97	98	99	A1	A2	A3	A4	A5	A6	A7	A8	A9	C0	C1	C2
4	20	[5B]	5D	C3	C4	C5	C6	C7	C8	C9	D0	. 2E	< 3C	[28	+ 2B	D1
5	& 26	D2	D3	D4	D5	D6	D7	D8	D9	E0	! 21	\$ 24	* 2A) 29	; 3B	^ 5E
6	- 2D	/ 2F	E2	E3	E4	E5	E6	E7	E8	E9	7C	, 2C	% 25	_ 5F	> 3E	? 3F
7	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	: 3A	# 23	@ 40	' 27	= 3D	" 22
8	80	a 61	b 62	c 63	d 64	e 65	f 66	g 67	h 68	i 69	8A	8B	8C	8D	8E	8F
9	90	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F	p 70	q 71	r 72	9A	9B	9C	9D	9E	9F
A	A0	~ 7E	s 73	t 74	u 75	v 76	w 77	x 78	y 79	z 7A	AA	AB	AC	AD	AE	AF
B	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C	{ 7B	A 41	B 42	C 43	D 44	E 45	F 46	G 47	H 48	I 49	CA	CB	CC	CD	CE	CF
D	} 7D	J 4A	K 4B	L 4C	M 4D	N 4E	O 4F	P 50	Q 51	R 52	DA	DB	DC	DD	DE	DF
E	\ 5C	E 1	S 53	T 54	U 55	V 56	W 57	X 58	Y 59	Z 5A	EA	EB	EC	ED	EE	EF
F	0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37	8 38	9 39	FA	FB	FC	FD	FE	FF

Translation Mappings Window

Troubleshooting

This section provides quick access to the most common components you can check when troubleshooting.

Registry Usage

The default framework stores preferences under the following Windows registry key:

```
HKEY_CURRENT_USER
  Software
    Software AG
      CST Frameworks
```

The name of this key is set in the AppSettings.bas module. It can be changed to any other key in HKEY_CURRENT_USER.

Other framework components store application preferences in subkeys of this key:

Component	Sub-Key	Constant in CSTObjectConstants.bas
Browse	BrowseObjects	BROWSE_SUBKEY
Maintenance preferences	MaintPreferences	MAINTENANCE_SUBKEY

The following SDC preferences are stored as values under the main registry key:

Value Name	Description
DispatchService	Name of the dispatch service to use. This dispatch service name is one of the names in SDC.ini.
UserID	User ID to use in calls to the dispatch service.

SDC.ini

The SDC.ini file stores Spectrum service definitions on the client. It is located in the Windows directory. To edit the SDC.ini, use the Spectrum Service Manager.

Tip: The order of the Spectrum service definitions in this file is irrelevant.

SDCApp.ini

In the Windows directory, you can use this file to specify which dispatcher to use if there is no DispatchService entry in the registry. This functionality is not generally used because the network error window lets you select a dispatch definition interactively. The syntax is:

```
[SDC]
DefaultDispatcher=<name from SDC.ini>
```

In the project directory, you can use this file to override the default LIF directory (which is the directory where the project is stored). The syntax is:

```
[SDC]
LibraryPath=<full pathname of a LIF directory>
```

Check for Necessary DLLs

The Ping function of the Spectrum Service Manager is the best tool to use to check that DLLs required by Spectrum are installed and are in the path. Pinging checks for the following DLLs (in this order):

- BROKERV.B.DLL (from ETB\BIN)
- CDED32.DLL (from Windows\System)

Construct Spectrum Add-In

The Construct Spectrum Add-In always uses the following registry key for the SDC preferences:

```
HKEY_CURRENT_USER
  Software
    Software AG
      Construct Spectrum Add-In
```

When you download or upload files, the Construct Spectrum Add-In uses a default library name, DBID, and FNR. It reads these from AppSettings.bas whenever you open a new project or use the Construct Spectrum Add-In for the first time in a Visual Basic session.

If you change these settings in AppSettings.bas, save the project and then restart Visual Basic to have the Construct Spectrum Add-In re-read these settings.

Visual Basic knows about the Construct Spectrum Add-In because of the following lines in Windows\VBADDIN.INI:

```
[Add-Ins32]
ConstructAddIn5.Connector=1
```

Useful SDC Properties

The SDC has many properties you can check when you get an SDC runtime error or a communication error. Use the Visual Basic Debug Immediate function to examine these properties. Some of these are displayed in the Network Error window.

Application Object

Property	Description
LIFDirectory	Directory where the SDC looks for LIF files. Defaults to project directory (or when running an EXE, where the EXE is located). May be overridden with the SDCApp.ini file in the same directory.
MainLibrary	Name of the main LIF file. Set in AppSettings.bas, with the DefaultLibrary variable.
UserID	User ID to use in calls to the dispatch service.

NaturalDataArea Object

Property	Description
LibraryImageFile	Name of the LIF file from which the data area definition was loaded.
Definition	Data area definition as read from the LIF file.
PackedData	Wire-buffer representation of the field values in the data area.
PackedDataLength	Size of the wire buffer representation (in characters).
Name	Name of the data area. The name may contain other components if this data area was created by using the FieldRef method of another NaturalDataArea object, or if the data area contains 1:V fields.

Dispatcher Object

Property	Description
ErrorSource	One of ETB, NAT, or SPE.
ErrorNumber	Error number, formatted according to the error source.
ErrorMessage	Error message.
ErrorValue(<i>i</i>)	Substitution parameter for the message.

RequestProperty Property

The SDCLib.Dispatcher object has a property called RequestProperty that returns information gathered during the last CallNat. The syntax for this property is:

```
RequestProperty(PropertyName As String) As Variant
```

The following property names are defined (the last column indicates whether this property is shown in the Network Error window):

Value Name	Data Type	Description	X
Request.AppService	String	Name of the application service definition (CUSTOMER, for example). This is the first parameter of the CallNat method. The application service definition is looked up in the LIF files.	X
Request.DataAreas	Integer	Number of Natural data area parameters passed into the CallNat method.	X
Request.DataArea(<i>i</i>)	Natural data area	1 <= <i>i</i> <= Request.DataAreas. Returns a reference to a NaturalDataArea object passed to the CallNat method.	X
Request.BlocksOut	String	Block header and data blocks (in wire-buffer format) passed to the subprogram proxy.	
Request.BlocksIn	String	Blocks header and data blocks (in wire buffer format) received from the subprogram proxy.	

Value Name (continued)	Data Type	Description	X
Request.Domain	String	Domain name read from the application service definition.	X
Request.Object	String	Object name read from the application service definition.	X
Request.Version	Long	Version number read from the application service definition.	X
Request.Method	String	Method name specified in the CallNat (or DEFAULT if not specified). The number of blocks and the blocks to send are looked up in the application service definition.	X
Request.InsideTransaction	Boolean	True if StartTransaction was called. All requests are sent to a dedicated dispatcher.	X
Request.DataOut	Byte Array	Full dispatcher request, starting with the request protocol bytes and the format byte.	
Request.RawDataIn	Byte Array	Binary response data received from the dispatcher, after all packets are assembled. Starts with the response protocol bytes and format byte.	
Request.DataIn	Byte Array	Dispatcher response message, after decryption, expansion, and translation to ASCII. Starts with the 4-digit dispatcher message number.	
Request.ReceivedData	String	Request.Data converted to a string with the StrConv(Request.DataIn, vbUnicode) function. If the dispatcher message number is "0000", contains the same as Request.BlocksIn.	
Packet.CountOut	Integer	Number of packets sent to the dispatcher.	

Value Name (continued)	Data Type	Description	X
Packet.DataOut(<i>i</i>)	Byte Array	(<i>i</i>) can be from 1 to Packet.CountOut. The bytes sent for packet <i>i</i> .	
Packet.CountIn	Integer	Number of packets received.	
Packet.DataIn(<i>i</i>)	Byte Array	(<i>i</i>) can be from 1 to Packet.CountIn. The bytes received for packet <i>i</i> .	
ETB.ConversationsID	String	Broker control block values for the last Broker call.	
ETB.Token	String	Generated to be unique.	
ETB.UserID	String	Always "SPECTRUM-DISPATCH-CLIENT".	

DEPLOYING YOUR CLIENT/SERVER APPLICATION

Once a Construct Spectrum project is developed and tested, the new application can be copied and installed on as many client machines as required. This chapter provides an overview of this procedure, as well as different considerations to keep in mind when deploying your client/server application.

The following topics are covered:

- **Transferring Data**, page 190
- **Distributing Your Application**, page 191

For related information, see **Deploying the Administration Subsystem**, page 125, *Construct Spectrum Administration*.

Transferring Data

Your test and development files may differ from your production environment. Before deploying your application, copy the definitions from your test or development file on the mainframe to your production environment. To copy the file, either:

- Use the supplied data transfer utilities.
- or
- Use the Construct Spectrum Administration subsystem to copy them manually.

Data Transfer Utilities

To copy definitions quickly, use the data transfer utilities. You can use these utilities to copy domains and groups between one Spectrum system file and another.

For more information, see **Data Transfer Utilities**, page 135, *Construct Spectrum Administration*.

Construct Spectrum Administration Subsystem

If desired, you can manually define and maintain the domain, application service definitions, and steplib information in the Construct Spectrum Administration subsystem.

For information about identifying where your application libraries reside on the server, see **Step 1: Define the Steplib Chain**, page 43.

For information on grouping application objects and services, see **Step 2: Define the Domain**, page 45.

For information on defining the domain, business object, and version of a Visual Basic business object, see **Access the Application Service Definitions**, page 112.

Distributing Your Application

- To distribute your application:
 - ❑ **Step 1: Create the Executable File**, page 191
 - ❑ **Step 2: Collect Files For Installation**, page 191
 - ❑ **Step 3: Install the Client Application**, page 192
 - ❑ **Step 4: Run the Application**, page 192

The following sections describe each of these steps in detail.

Step 1: Create the Executable File

The first step is to create the file to execute the application.

- To create the executable file:
 - 1 Open Visual Basic.
 - 2 Select Make EXE File from the File menu.

Step 2: Collect Files For Installation

Next, collect the following files for installation on the target PC:

- The executable file created in the previous step.
- All runtime support files required by the executable file.
- The library image files (installed in the same directory as the executable file).
- Any resource files your application accesses from Resource class for the client framework and any sound files used by validation errors.
- Any other data files used by your Construct Spectrum application.

Note: If the target PC has the Visual Basic runtime support files installed, you need only copy the executable and the library image files to the target PC.

The procedure to ready your files for installation differs depending on whether you are creating an installation tape, installing from disk, or using a server to copy files to the target PC.

To create a professional setup program for your application, use the Package and Deployment wizard in Visual Basic or another setup toolkit. These programs ensure that all required support files are included with your setup program. As with any external data file used by the application, you must add the library image files to your setup program manually.

Warning:

The Package and Deployment wizard detects that your client application uses the Spectrum Dispatch Client (SDC) and lists it as one of your application support files. Because the SDC is installed separately on the target PC, you must remove the check mark from the SDC in the list so it is not included with your setup program.

Step 3: Install the Client Application

Once you have a set of distribution files, you can install the client application on the target PC. This procedure differs depending on whether you are creating an installation tape, installation disks, or using a server to copy your files to the target PC. It also differs depending on which setup toolkit was used to create your setup program.

There are no prerequisites for installing the client application.

Step 4: Run the Application

Before running the client application, ensure:

- The Spectrum Dispatch Client is installed on the target PC.
- That either Entire Net-Work is installed and configured to access Entire Broker or Entire Broker is configured to use TCP/IP.

You can now run your newly installed application on the target PC. If installation was successful, your application behaves identically to your tested application in your development environment.

If an error message is displayed, see **Debugging Your Client/Server Application**, page 161, for possible causes.

Note: While all error messages are displayed on the client, some conditions can be remedied only by a system administrator on the mainframe.

USING THE SPECTRUM DISPATCH CLIENT

This chapter describes the Spectrum Dispatch Client (SDC), a key component of Construct Spectrum development. The SDC allows you to make calls from a client to Natural subprograms running on a server.

The following topics are covered:

- **Overview**, page 194
- **Calling a Natural Subprogram**, page 195
- **Spectrum Dispatch Client Components**, page 197
- **Advanced Features**, page 228

For related information, see:

- **Creating Applications Without the Framework**, page 235
This chapter describes the process of creating applications using Construct Spectrum without using the client framework.

Overview

The SDC gives application developers the ability to make calls from a client to Natural subprograms running on a server. The following examples show a parameter data area and code for a Natural subprogram.

Example of the parameter data area for the CUSTN Natural subprogram

```
DEFINE DATA
  PARAMETER USING NCUSTPDA
  PARAMETER USING NCUSTPDR
  PARAMETER USING CDAOBJ
  PARAMETER USING CDPDA-M
  . . .
END-DEFINE
```

Example of Natural code to call CUSTN

```
DEFINE DATA
  LOCAL USING NCUSTPDA
  LOCAL USING NCUSTPDR
  LOCAL USING CDAOBJ
  LOCAL USING CDPDA-M
END-DEFINE
*
ASSIGN NCUSTPDA.CUSTOMER-NUMBER = 10001
ASSIGN CDAOBJ.#FUNCTION = 'GET'
CALLNAT 'CUSTN' NCUSTPDA NCUSTPDR CDAOBJ CDPDA-M
. . .
END
```

Using the SDC, you can write similar Visual Basic code that declares these Natural data areas, assigns values to the fields in the data areas, performs a CALLNAT, and then examines the data areas to determine the results.

Note: The examples presented throughout this chapter use Visual Basic as a model for creating applications. You may choose to use another OLE-compliant programming tool with Construct Spectrum.

Calling a Natural Subprogram

- To call a Natural subprogram from the client:
 - ❑ **Step 1: Create Parameter Data Area Instances**, page 195
 - ❑ **Step 2: Assign Values to the Fields**, page 195
 - ❑ **Step 3: Use the CallNat Method on the Client**, page 196
 - ❑ **Step 4: Check the Success of the CALLNAT**, page 196

The following sections describe these steps in detail.

Step 1: Create Parameter Data Area Instances

- To create the PDA instances:
 - 1 Declare the variables for the Natural data areas expected by your subprogram.
For example:

```
Dim ncustpda As NaturalDataArea
Dim ncustpdr As NaturalDataArea
Dim cdaobj As NaturalDataArea
Dim cdpda_m As NaturalDataArea
```

In this example, the variable names are similar to the names of the external PDAs. For CDPDA-M, the dash character was changed to an underscore because the dash is not valid in a Visual Basic variable name.

- 2 Associate the name of the Natural data area with each variable.
To do this, call a routine that creates an instance of a Natural data area. For example:

```
Set ncustpda = SDCApp.Allocate("NCUSTPDA")
Set ncustpdr = SDCApp.Allocate("NCUSTPDR")
Set cdaobj = SDCApp.Allocate("CDAOBJ")
Set cdpda_m = SDCApp.Allocate("CDPDA-M")
```

This example calls the Allocate method of the SDCApp object (described later in the chapter).

Step 2: Assign Values to the Fields

This step sets up the input parameters for the call. To assign values to the fields in the PDAs, read and write the fields in the data areas.

The following example writes one field in each of the NCSTPDA and CDAOBJ data areas:

```
ncustpda.Field("CUSTOMER-NUMBER") = 10001
cdaobj.Field("#FUNCTION") = "GET"
```

In this example, the NaturalDataArea object's Field property reads and writes the fields.

Step 3: Use the CallNat Method on the Client

This step uses the CallNat method to call a remote subprogram. The following example uses a communications object called Dispatcher:

```
Dispatcher.CallNat "CUSTN", ncustpda, ncustpdr, cdaobj, cdpda_m
```

where:

CUSTN	Is the name of the Natural subprogram to call.
ncustpda ncustpdr cdaobj cdpda_m	Are the names of the data areas passed into the subprogram.

The syntax of the CallNat method on the Dispatcher resembles a CALLNAT in a Natural program.

Step 4: Check the Success of the CALLNAT

Because this CALLNAT occurs between two machines over a network, an error may occur. To confirm the success of the CALLNAT, examine the error properties for the Dispatcher object. The Successful property is True if the CALLNAT succeeded. If the Successful property is False, check the ErrorNumber, ErrorSource, and ErrorMessage properties to find out what went wrong.

The following example checks the success of the CALLNAT:

```
If Dispatcher.Successful Then
    ' The call was successful. Read the fields in the data areas.
    ...
Else
    MsgBox "An error occurred." & _
        " Number = " & Dispatcher.ErrorNumber & _
        " Source = " & Dispatcher.ErrorSource & _
        " Message = " & Dispatcher.ErrorMessage
End If
```

Summary

The previous examples illustrate the process of calling Natural subprograms from the client. There are many other details you must first specify before this example can run successfully. These include defining the Natural data areas (see Step 1), locating and invoking the Natural subprogram (see Step 3), and initializing the SDCApp and Dispatcher objects. These steps are described later in this chapter.

Spectrum Dispatch Client Components

The SDC provides the following key functions:

- Natural data area simulation
- Client/server communication

The following components provide Natural data area simulation:

Component	Description
Data area definitions	Define fields in the Natural data areas used by your client applications.
Data area allocator	Reads data area definitions and creates data area objects.
Data area objects	Provide properties and methods to read and write Natural data areas for your client application.

The following components provide client/server communication:

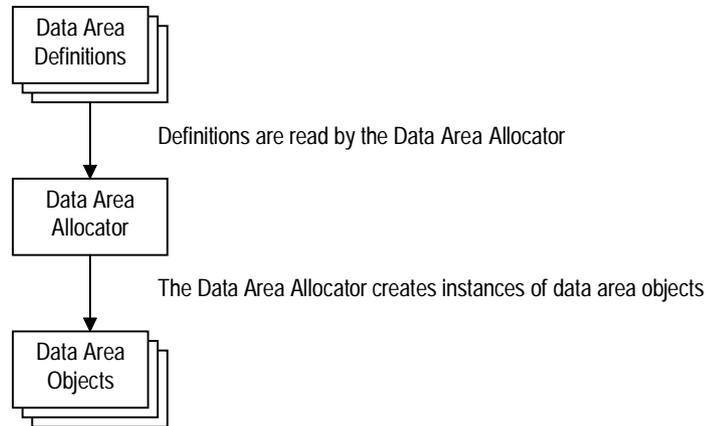
Component	Description
Application service definitions	Define the Natural subprograms called by your client application.
Dispatch service definitions	Define the parameters required to communicate with the Spectrum dispatch service.
Dispatcher objects	Provide properties and methods to interact with the Spectrum dispatch service.

For more information, see *Construct Spectrum Messages*.

The following sections provide more details about each of these components.

Natural Data Area Simulation

When a client application calls a Natural subprogram, it uses parameter data areas (PDAs) to pass parameters to the subprogram and receive parameters from the subprogram. Using Construct Spectrum, you can simulate Natural data areas in Visual Basic. The SDC components that provide this capability are the data area definitions, the data area allocator, and the data area objects:



Components Used to Simulate Natural Data Areas

Data Area Definitions

Data area definitions use the same syntax as an inline data area in Natural code. These definitions are stored in library image files.

For information, see **Library Image Files and the Steplib Chain**, page 227.

Example of the NCUSTPDA data area definition

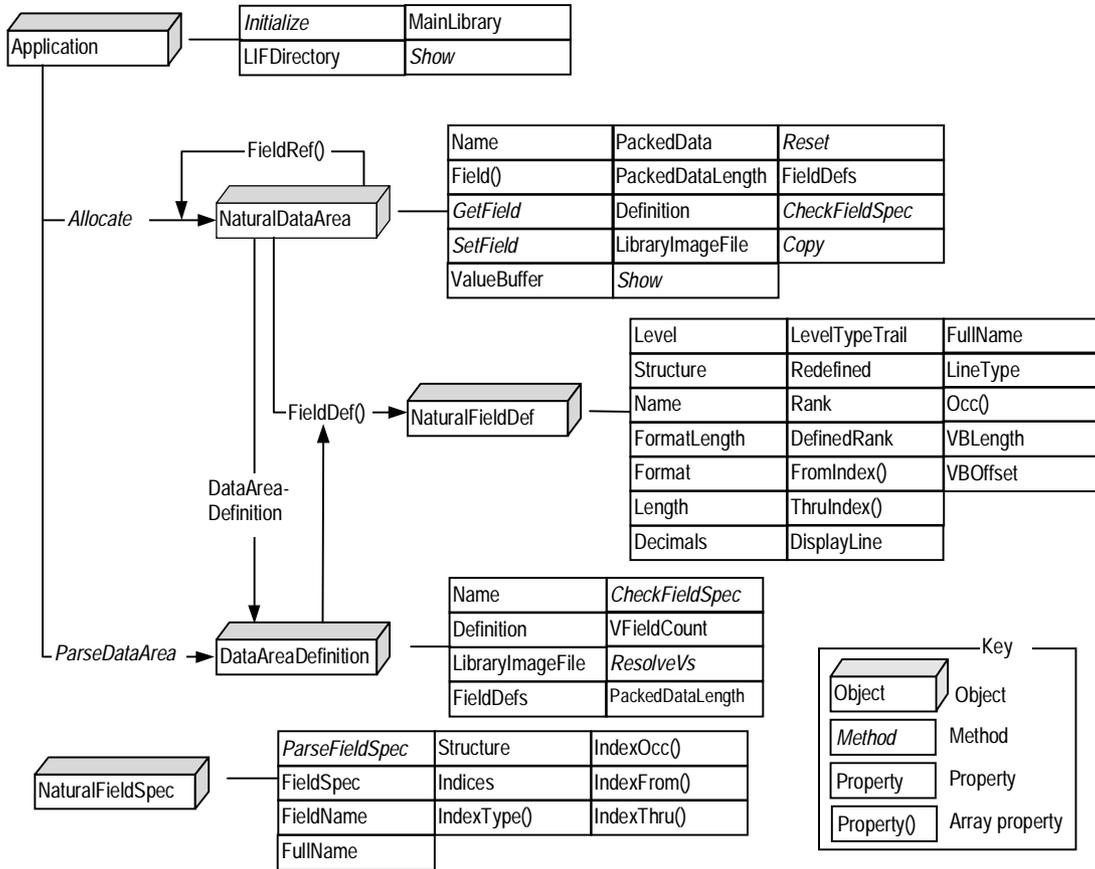
```
[DataArea NCUSTPDA]
01 CUSTOMER
  02 CUSTOMER-NUMBER (N5)
  02 BUSINESS-NAME (A30)
  02 PHONE-NUMBER (N10)
  02 MAILING-ADDRESS
    03 M-STREET (A25)
    03 M-CITY (A20)
    03 M-PROVINCE (A20)
    03 M-POSTAL-CODE (A6)
  02 SHIPPING-ADDRESS
    03 S-STREET (A25)
    03 S-CITY (A20)
    03 S-PROVINCE (A20)
    03 S-POSTAL-CODE (A6)
  02 CONTACT (A30)
  02 CREDIT-RATING (A3)
  02 CREDIT-LIMIT (P11.2)
  02 DISCOUNT-PERCENTAGE (P3.2)
  02 CUSTOMER-WAREHOUSE-ID (A3)
  02 CUSTOMER-TIMESTAMP (T)
01 CUSTOMER-ID (N5)
01 REDEFINE CUSTOMER-ID
  02 STRUCTURE
    03 CUSTOMER-NUMBER (N5)
```

The SDC supports the following features in a data area definition:

- All Natural field formats: A, B, C, D, F, I, L, N, P, and T
- Scalar fields
- One, two, and three-dimensional arrays
- Structures
- Structure arrays
- Redefinitions, including the FILLER keyword
- Arrays with a variable number of occurrences (1:V)

Data Area Simulation Objects

Many different SDC objects are involved in data area simulation. These objects and their properties and methods are illustrated in the following object diagram:



SDC Objects Involved in Data Area Simulation

Application Object

The application object is one of the externally-creatable objects exposed by the SDC. It has the following properties and methods related to data area simulation:

Property or Method	Description
Initialize method	Tells the SDC the name of the library image file directory and name of the main library.
LIFDirectory property	Returns the name of the library image file directory set with the Initialize method.
MainLibrary property	Returns the name of the main library set with the Initialize method.
Allocate method	Allocates a NaturalDataArea object.
Show method	Displays a pop-up window showing the values of all fields in one or more data areas (values can be edited).
ParseDataArea method	Similar to Allocate, but creates a DataAreaDefinition object that can be used to parse a data area. The DataAreaDefinition does not store field values.

A client application creates one global instance of the application object and uses it to create NaturalDataArea objects.

Example of declaring and initializing the application object

```
Public SDCApp As SDCLib6.Application

Public Sub Main
    Set SDCApp = New SDCLib6.Application
    SDCApp.Initialize App.Path, "LIBRARY"
End Sub
```

This example creates a global Application object called SDCApp and then uses the object's Initialize method to set the library image file directory and main library.

For more information on the library image file directory and the main library, see **Library Image Files and the Steplib Chain**, page 227.

Create NaturalDataArea Objects

The data area allocator reads data area definitions from library image files. It then creates NaturalDataArea objects that know the structure of their data area definitions and allow you to read and write fields in their data area.

- To create NaturalDataArea objects:
- Call the Allocate method of the Application object. The Allocate method has the following syntax:

```
Function Allocate (DataAreaName As String, _  
                  ParamArray VSubstitutions() As Variant) _  
                  As NaturalDataArea
```

where:

DataAreaName Is the name of a Natural data area.

VSubstitutions Is the parameter used when the data area has one or more 1:V arrays. For information, see **1:V Fields**, page 233.

NaturalDataArea Class

The data area allocator creates data area objects that are instances of the NaturalDataArea class. Each object knows the structure of its own data area definition and allows you to read and write fields in that data area.

The NaturalDataArea class defines the properties and methods of the simulated Natural data areas. Each instance of this class stores details about its structure and maintains the field values for a single Natural data area. A client application can create as many instances of the same or different data areas as required.

The properties and methods of the NaturalDataArea class are:

Property or Method	Description
CheckFieldSpec property	<p>Checks whether a field name is defined in the data area. Raises a runtime error if the field name is not valid. For example:</p> <pre>dataarea1.CheckFieldSpec "CUSTOMER-NUMBER" dataarea1.CheckFieldSpec "ROW(1)"</pre>
Copy method	<p>Creates a copy of a NaturalDataArea object with the same definition and the same field values. Field values changed in one do not affect the other. For example:</p> <pre>Dim data1 As NaturalDataArea Dim data2 As NaturalDataArea ' Allocate a data area. Set data1 = Nat.Allocate2 (DATAAREA_CSASTD) ' Create a copy of this data area. Set data2 = data1.Copy()</pre>
DataAreaDefinition property	<p>Provides information about the structure of a Natural data area, such as the name, format, length, and level number of each field.</p>
Definition property	<p>Returns a multiple-line string containing the data area definition as read from the library image file.</p>
Field property	<p>Reads and writes the value in a field. This property receives a field name as a parameter. If the field is part of an array, also specify index values as part of the field name. For example:</p> <pre>With dataarea1 .Field("CUSTOMER-NUMBER") = 10001 .Field("PHONE-NUMBER(1)") = "4165551234" .Field("STREET(1,1)") = "134 Hill Blvd." .Field("CUSTOMER-NAME") = sname End With</pre>

Property or Method	Description (continued)
FieldDef property	<p>Returns a <code>NaturalFieldDef</code> object defining a field. (For information, see NaturalFieldDef Class, page 210.) If the field is part of an array, any specified index values are ignored. For example:</p> <pre>Dim flddef As NaturalFieldDef Set flddef = dataareal.FieldDef("M-CITY") If flddef.FormatLength = "A20" Then ... End If ' The following two lines do the same thing. Set flddef = dataareal.FieldDef("SALARY") Set flddef = dataareal.FieldDef("SALARY(1)")</pre> <p>You can also enumerate fields in the data area using a numeric index instead of a string field name. For example:</p> <pre>For i = 1 to dataareal.FieldDefs Print dataareal.FieldDef(i).Name Next</pre>
FieldDefs property	Returns the number of field definitions in the data area definition.
FieldRef property	Creates a new <code>NaturalDataArea</code> object containing a subset of the fields. For information, see FieldRef Property , page 228.
GetField method	<p>Reads the value in a field. Similar to the <code>Field</code> property, except index values are not specified as part of the field name but as optional parameters. For example:</p> <pre>With dataareal Print .GetField("CUSTOMER-NUMBER") Print .GetField("PHONE-NUMBER", 1) Print .GetField("STREET", 1, 1) End With</pre>
LibraryImageFile property	Returns the full file name of the library image file from which the data area definition was loaded.
Name property	Returns the name of the data area represented by the object. This name was passed to the <code>Allocate</code> method.
PackedData property	<p>Returns field values for the data area as an alphanumeric string. Assigning an alphanumeric string to this property replaces the field values in the data area with the values in the string. The length of the string must be the defined. The following example copies all field values from one data area to another:</p> <pre>dataarea2.PackedData = dataareal.PackedData</pre>

Property or Method	Description (continued)
PackedDataLength property	<p>Returns the length of the packed data. For example:</p> <pre>If Len(pdata) <> dataareal.PackedDataLength Then MsgBox "The packed data is not " & _ "the right length." Else dataareal.PackedData = pdata End If</pre>
Reset method	<p>Resets the fields in the data area to their default values. For example:</p> <pre>dataareal.Reset</pre> <p>You can also pass a field name into the Reset method to reset only that field. For example:</p> <pre>dataareal.Reset "CUSTOMER-NUMBER"</pre> <p>You can also reset structures and multiple occurrences of an array. For example:</p> <pre>dataareal.Reset "CUSTOMER" dataareal.Reset "STREET(*,*)" dataareal.Reset "STREET(1,*)"</pre>
SetField method	<p>Writes the value in a field. Similar to the Field property, except index values are not specified as part of the field name but as optional parameters. For example:</p> <pre>With dataareal .SetField 10001, "CUSTOMER-NUMBER" .SetField "4165551234", "PHONE-NUMBER", 1 .SetField "134 Hill Blvd.", "STREET", 1, 1 End With</pre>
Show method	<p>Displays a pop-up window showing the values of all fields in the data area. The syntax is:</p> <pre>object.Show</pre> <p>You can edit the field values.</p>
ValueBuffer property	<p>Sets or returns a copy of the internal block of memory that stores field values (the value buffer). Use this property to copy a field value from one data area to another. For example:</p> <pre>data1.ValueBuffer = data2.ValueBuffer</pre>

Case Sensitivity

Field names passed into the procedures of the NaturalDataArea class are not case-sensitive. You can type the field name in uppercase, lowercase, or mixed case.

Tip: To be consistent with Natural, specify all field names in uppercase.

Alphanumeric Fields

When reading an alphanumeric field (format A), the returned value does not contain trailing blanks. If a field contains only blanks, the value is returned as an empty string. When assigning a value to a field, the value is truncated if it is longer than the field or padded with spaces (internally) if it is shorter than the field.

Fully Qualified Field Names

Whenever a field name is passed into the procedures of the NaturalDataArea class or DataArea, the field name can include the level 1 structure name as a qualifier. The level 1 structure name, however, is required if there is more than one field with the same name in the same data area.

Example of using the level 1 structure name as a qualifier

```
01 CDAPROXY
  02 DATA-LENGTH(I4)
  02 DOMAIN(A8)
  02 OBJECT(A32)
  02 METHOD(A32)
01 CDAOBJ
  02 OBJECT(A20)

With dataarea
  .Field("DOMAIN") = "TEST"
  .Field("CDAPROXY.OBJECT") = "EMPLOYEE"
  .Field("CDAOBJ.OBJECT") = "EMPLOYEE"
End With
```

Redefined Fields

The SDC allows you to redefine fields, arrays, and structures just as in Natural.

Example of a redefined field

```
01 ACCOUNT(A12)
01 REDEFINE ACCOUNT
  02 COST-CENTER(A3)
  02 ACCT(A4)
  02 PROJECT(A5)
```

When the Cost-Center, Acct, or Project fields are updated, the change is also reflected in the Account field. Similarly, when the Account field is changed, the Cost-Center, Acct, and Project fields are updated.

Redefinitions that change the format or interpretation of data may introduce side effects that are implementation dependent.

Example of the side effects of using redefined fields

```
01 OBJECT-VERSION(N6)
  02 VERSION(N2)
  02 RELEASE(N2)
  02 MAINT-LEVEL(N2)
```

```
With myver
  .Field("VERSION") = 2
  .Field("RELEASE") = 1
  .Field("MAINT-LEVEL") = 1
  Print .Field("OBJECT-VERSION")' Prints: 20101
  .Field("RELEASE") = -1
  Print .Field("OBJECT-VERSION")' Prints: <implementation defined>
End With
```

Errors When Compiling

When cataloging a Natural module, the Natural compiler checks whether fields referred to in Natural source code are actually part of the data area. If a field name is not valid, if the wrong number of index values is specified for an array field, or if the data type is not compatible, the Natural compiler generates a compile error.

The Visual Basic compiler cannot check for these errors because it does not have knowledge of Natural. Because the SDC provides a runtime Natural simulation layer, a Visual Basic developer will not discover an invalid field name until the statement that uses it is executed.

Read Arrays and Structures

You must specify the necessary index values when reading or writing one, two, or three-dimensional arrays. The following examples show two different ways to read array fields.

Example of reading arrays with the GetField method

```
01 BROWSE-RECORDS(1:20)
   02 NAME(A5)
   02 OTHER-COLUMNS(A20/1:5)
01 ...

For irow = 1 To 20
  Print .GetField("NAME", irow); " ";

  For icol = 1 To 5
    Print .GetField("OTHER-COLUMNS", irow, icol); " ";
  Next
  Print
Next
```

Example of specifying a field with occurrences

```
Print .Field("OTHER-COLUMNS(" & irow & "," & icol & ")")
```

If the field has more than one dimension, specify the index values with a comma separating them in the Field property. You can also read a structure field and return it as a Byte array (as though the entire structure is defined as a B1 array). This is useful when moving occurrences of a structure array.

The following example shows how to read and write occurrences of a structure array. This example shuffles occurrences of the Item array down to simulate deleting the occurrence number stored in the DeleteItem variable.

Example of a data area definition

```
01 ITEM(1:10)
   02 NUMBER (N5)
   02 DESCRIPTION (A30)
   02 UNIT-COST (P7.2)
   02 QUANTITY (N5)
   02 TOTAL-COST (P7.2)
```

Example of reading occurrences of the Item array

```

With dataareal
  For i = DeleteItem + 1 To 10
    .Field("ITEM(" & i - 1 & ")") = .Field("ITEM(" & i & ")")
  Next
End With

```

Runtime Errors

Many different runtime errors can result from using NaturalDataArea objects.

DataDefinitionArea Class

This class provides information about the structure of a Natural data area, such as the name, format, length, and level number of each field. The NaturalDataArea and DataAreaDefinition classes have many properties in common because they both store the definition of a Natural data area. However, unlike the NaturalDataArea class, the DataAreaDefinition class does not store field values.

The SDC provides two ways to create an instance of a DataAreaDefinition: using the ParseDataArea method of the Application class to parse an inline data area definition or using a data area definition in an external LIF file. Optionally, you can use the DataAreaDefinition property of a NaturalDataArea object. In the SDC, a NaturalDataArea object uses a DataAreaDefinition object to store the structure of the data area. The DataAreaDefinition property returns a reference to that DataAreaDefinition object.

Property or Method	Description
CheckFieldSpec property	Checks whether a field name is defined in the data area. Raises a runtime error if the field name is not valid. For example: <pre>dataareal.CheckFieldSpec "CUSTOMER-NUMBER" dataareal.CheckFieldSpec "ROW(1)"</pre>
Definition property	Returns a multiple-line string containing the entire data area definition as read from the library image file.
FieldDef property	Returns a NaturalFieldDef object defining a field. For information, see NaturalFieldDef Class , page 210.

Property or Method	Description (continued)
FieldDefs property	Returns the number of field definitions in the data area definition.
PackedDataLength property	Returns the length of the packed data. For example: <pre>If Len(pdata) <> dataareal.PackedDataLength Then MsgBox "The packed data is not " & _ "the right length." Else dataareal.PackedData = pdata End If</pre>

NaturalFieldDef Class

NaturalFieldDef is an SDC class that returns the definition for a single field in a data area definition. The FieldDef property of the NaturalDataArea class creates and returns an instance of the NaturalFieldDef class. All properties defined by this class are read-only. These properties are:

Property	Description
Decimals	Returns the decimal length portion of the Natural format. If the format is not numeric or packed numeric, it returns 0. Returns 0, 0, 2, and 0 in the FormatLength example.
DefinedRank	Returns the number of dimensions of the field in the data area definition. This property works similar to the Rank property, except it returns the number of dimensions regardless of any structure arrays it might be part of.
Format	Returns the Natural format character. Returns N, A, P, and D in the FormatLength example.
FormatLength	Returns the format and length of the field in Natural syntax. Returns N6, A20, P8.2, and D for the following example: <pre>With employee Print .FieldDef("PID").FormatLength Print .FieldDef("FIRST-NAME").FormatLength Print .FieldDef("SALARY").FormatLength Print .FieldDef("HIRE-DATE").FormatLength End With</pre>

Property	Description (continued)
FromIndex and ThruIndex	<p>Returns the low and high index values for each dimension of an array field. Returns 1,10 and 5,7 in the following example:</p> <pre data-bbox="635 422 1334 577">01 VALUES(N10/1:10,5:7) With data.FieldDef("VALUES") For i = 1 To .Rank Print .FromIndex(i) & ":" & .ThruIndex(i) Next End With</pre>
FullName	Returns the fully qualified field name (includes the level 1 structure name).
Length	Returns the length portion of the Natural format. If the format is D, L, or T, it returns 0. Length returns 6, 20, 8, and 0 in the FormatLength example.
Level	Returns the field's level number in the data area definition.
LevelTypeTrail	<p>Returns a string that determines the nesting of this field in the data area definition. This string has one character for each level. Each character can be one of the following:</p> <ul data-bbox="635 976 823 1117" style="list-style-type: none"> • F (field) • S (structure) • R (redefine) • X (filler)

Property	Description (continued)
LevelTypeTrail (continued)	<p>For the following data area example:</p> <pre> 01 ROW-COUNT (N2) 01 ROW (1:10) 02 ID (N6) 02 ACCOUNT-NO (A16) 02 REDEFINE ACCOUNT-NO 03 DIVISION (A4) 03 FILLER 1X 03 GROUP (A5) 03 FILLER 1X 03 ENTITY (A5) </pre> <p>LevelTypeTrail returns:</p> <pre> Print .FieldDef("ROW-COUNT").LevelTypeTrail ' Prints "F" Print .FieldDef("ROW").LevelTypeTrail ' Prints "S" Print .FieldDef("ID").LevelTypeTrail ' Prints "SF" Print .FieldDef("ACCOUNT-NO").LevelTypeTrail ' Prints "SF" Print .FieldDef("DIVISION").LevelTypeTrail ' Prints "SRF" Print .FieldDef(7).LevelTypeTrail ' Prints "SRX" </pre>
Name	Returns the name of the Natural field.
Occ	Returns the number of occurrences for each dimension of an array field.
Rank	<p>Returns whether the field is a scalar field or part of an array. Rank indicates the number of index values that must be used when reading or writing the field values:</p> <ul style="list-style-type: none"> • 0 (scalar) • 1 (one-dimensional array) • 2 (two-dimensional array) • 3 (three-dimensional array)
Redefined	Returns True if the field is redefined later in the data area definition.
Structure	Returns the structure name if the field is part of a level 1 structure.
ThruIndex	See FromIndex.

Client/Server Communication

The other major function of the SDC is client/server communication. Many components work together to enable client/server communication. These include:

- Application service definitions
- Dispatcher objects
- Dispatcher service definitions

The following sections describe these components in more detail.

Level 1 Block Optimization

Before you can understand application service definitions, you must understand level 1 block optimization. The SDC and the subprogram proxies implement this performance optimization feature to minimize the amount of data that is transmitted between the client and server for each remote CALLNAT.

With level 1 block optimization, each level 1 field in the parameter data of the Natural subprogram becomes a numbered block. Each block can contain one or more Natural fields, structures, or structure arrays. Instead of sending all parameter data between the client and server for each remote CALLNAT, the SDC and Spectrum dispatch service transmit a subset of the blocks in each direction.

To understand why this is useful, consider the following. For most Natural subprograms, each field in the parameter data can be assigned a directional attribute to indicate whether a field passes data into the subprogram, out of the subprogram, or both.

Note: These directional attributes are not supported by Natural. However, they may be defined in the application service definitions supported by the SDC and coded in user exits in subprogram proxies.

The following table summarizes these directional attributes:

Directional Attribute	Description
IN	Passed from the caller to the subprogram.
OUT	Returned from the subprogram to the caller.
IN/OUT	Passed from the caller to the subprogram, optionally modified by the subprogram, and then returned to the caller.

If the parameter data is organized such that each block (level 1 field) contains only In, Out, or In/Out parameters, then the SDC can use level 1 block optimization to send only the In and In/Out parameters to the subprogram proxy. The subprogram proxy can send only the Out and In/Out parameters back to the client. In some cases, the size of the In or Out parameters is small compared to the total size of the parameter data. Level 1 block optimization can make a significant difference to the size of the data being transmitted over your network.

Note: This block optimization feature does not allow directional attributes to be assigned at a level of granularity finer than level 1 fields.

Occasionally, it may not be possible to assign a static directional attribute to a parameter because it may change its direction depending on the values of other parameters. This is illustrated in the following example:

Example of parameter data for a Natural Construct object subprogram

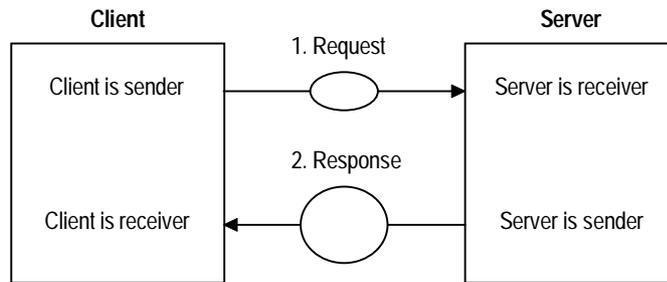
```

DEFINE DATA
  01 CUSTOMER                                /* Object PDA
    02 CUSTOMER-NUMBER (N5)
    02 BUSINESS-NAME (A30)
    02 PHONE-NUMBER (N10)
    ...
  01 NCUSTPDA-ID
    02 ...
  01 NCUSTPDR
    ...
  01 CDAOBJ
    02 #FUNCTION (A15)
    ...
  01 MSG-INFO
    02
    ...
END-DEFINE

```

The object PDA is either In, Out, or In/Out, depending on the #FUNCTION flag in CDAOBJ. When #FUNCTION contains Get, the object PDA contains data returned from the subprogram to the caller, so it is an Out parameter. When #FUNCTION contains Update, the caller is passing data in the object PDA to the subprogram, and depending on whether the subprogram performs edits on the data, the subprogram may also return updated values in the object PDA, so it is either an In or an In/Out parameter.

When using level 1 block optimization, the sender always decides which blocks are sent to the receiver. Sender and receiver differs from client and server because the client and server are both senders and receivers.



Client and Server are Both Sender and Receiver

When a request is sent to the server, the client is the sender and the server is the receiver. When the response is sent to the client, the server is the sender and the client is the receiver.

In the example above, different-sized ellipses show how the size of the request data may be different from the size of the response data because the set of blocks may be different.

Application Service Definitions

Application service definitions are defined on the server in the Construct Spectrum Administration subsystem and on the client in a library image file. The following table compares the information stored on the server and on the client:

Information	Stored on Client	Stored on Server
Domain name	X	X
Object name	X	X
Object version number	X	X
Method names	X	X
Name of subprogram proxy to call for each method	X	
Steplib chain to use when calling a subprogram	X	

Information	Stored on Client	Stored on Server
Number of level 1 fields in the parameter data area for each method's subprogram		X
Name of the level 1 fields sent to the server for each method		X

Example of an application service definition in a library image file

```
[AppService CUSTOMER]
Domain=DEMO
Object=CUSTOMER
Version=4.4.1
Method=BROWSE,,4,1+3+4
Method=DEFAULT,,5,1+2+3+4+5
Method=DELETE,,5,2+3+4
Method=EXISTS,,5,2+4
Method=GET,,5,2+4
Method=INITIALIZE,,5,4
Method=NEXT,,5,2+4
Method=STORE,,5,1+4
Method=UPDATE,,5,1+3+4
```

where:

[AppService CUSTOMER]	Introduces the application service definition and identifies the application service definition name.
Domain, Object, and Version	Identify the application service definition in the Construct Spectrum Administration subsystem.
Method	Defines a method within the application service.

Each method line contains four values separated by commas:

- A logical method name used in your Visual Basic code.
- A physical method name that corresponds to a method name in the application service definition on the server. If this name is the same as the logical method name, it can be omitted, as in the example above.
- The number of level 1 fields in the parameter data of the subprogram associated with the method. In the example above, the subprogram for the Browse method has four level 1 fields in its parameter data, and the subprograms for all other methods have five level 1 fields in their parameter data areas.
- The names of level 1 fields sent to the server for the method. In the previous example, only the first, third, and fourth level 1 fields are sent to the server when calling the Browse method.

The application service definitions on the client and server work together to allow a client application to identify which subprogram to call on the server. To use the CallNat method on the client, do not specify a Natural subprogram to call. Instead, specify the name of an application service definition. The SDC uses this name to look up the domain name, object name, and version number, and passes these values to the Spectrum dispatch service running on the server. The dispatch service uses the values to look up the subprogram proxy to call.

The following example shows a CallNat method on the client using the application service definition from the previous example:

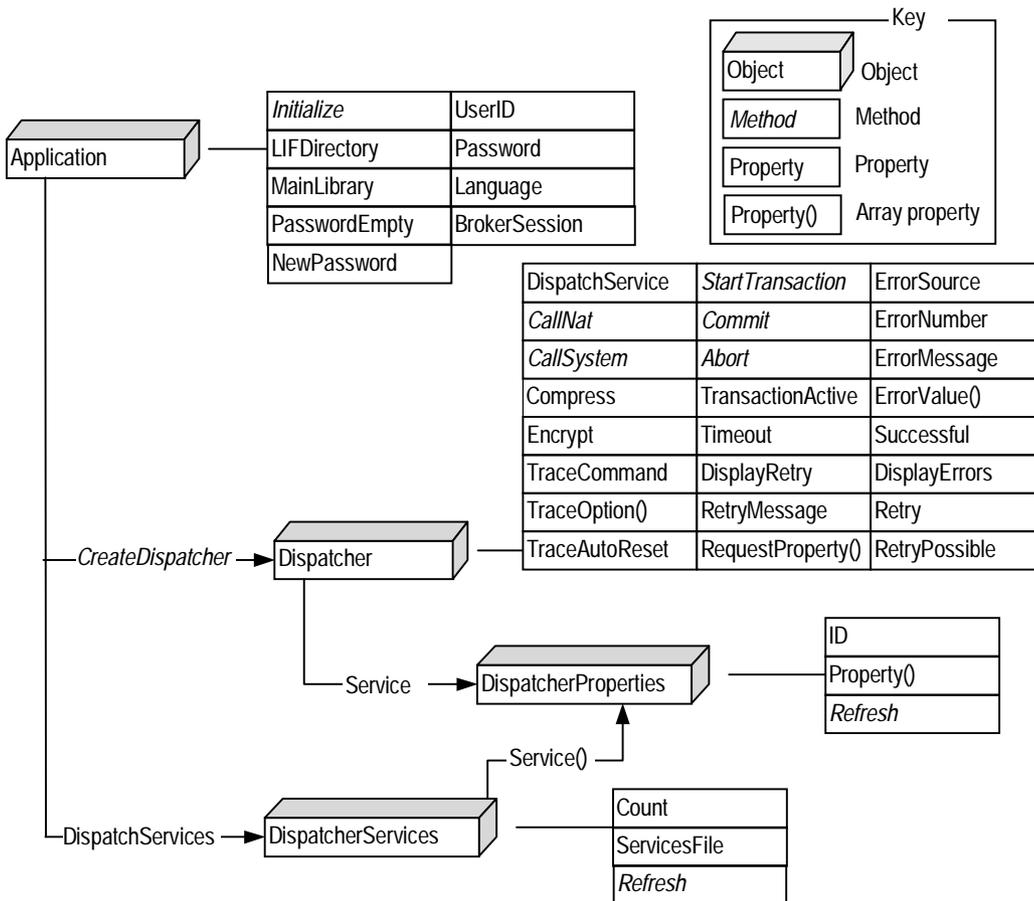
```
Dispatcher.CallNat "CUSTOMER.GET", ncustpda, ncustpda_id, ncustpdr, _  
                    cdaobj, cdpda_m
```

Notice how the GET method name is appended to the CUSTOMER application service name. If you do not specify a method name in the CallNat, the SDC uses the DEFAULT method name and this method must exist in the application service definition.

Dispatcher Objects and Dispatch Service Definitions

Dispatcher is an SDC class that handles communication between the client and server. It contains the networking components of the SDC.

The properties, methods, and related objects of the Dispatcher class are:



Dispatcher Objects

Dispatcher objects are created using the CreateDispatcher method of the Application object.

Example of creating Dispatcher objects

```
Dim Dispatcher As Dispatcher
Set Dispatcher = SDCApp.CreateDispatcher()
```

The properties and methods of the Dispatcher object are separated into the following functional groups:

- Service selection
- Remote subprogram invocation
- Timeout, retry, and resume handling
- Compression and encryption
- Tracing
- Database transaction control
- Error reporting

The following sections describe each of these groups in more detail.

Service Selection

You may have multiple Spectrum dispatch services running on one or more server platforms simultaneously. There could even be different types of Spectrum dispatch services, each with its own defaults, security settings, FUSER, and so on running at the same time. Before sending any request, the client must first identify which Spectrum dispatch service to connect to. You do this by setting the DispatchService property to the ID of a valid Spectrum dispatch service.

The available dispatch services are defined in the Construct Spectrum Administration subsystem on the server platform. On the client, these dispatch services are defined with the Spectrum Service Manager.

Each dispatch service definition specifies the following values:

- EntireX Broker ID
- Server class
- Server name
- Service

If you are familiar with EntireX Broker, you will recognize that this combination of values uniquely identifies an EntireX Broker service. Each Spectrum dispatch service is actually an EntireX Broker service.

Remote Subprogram Invocation

To send a request to the Spectrum dispatch service, use the `CallNat` and `CallSystem` methods. These methods return `True` if the call was successful and `False` if the call was unsuccessful.

The `CallNat` method invokes a Natural subprogram on the server.

Syntax of the CallNat method

```
Function CallNat (ByVal AppServiceName As String, _  
                 ParamArray DataAreas() As Variant) As Boolean
```

The name of the application service is always required. Following this name, you can specify zero or more instances of the `NaturalDataArea` class passed as parameters to the target subprogram. The parameters are passed by reference. When the subprogram returns, any changes the subprogram made to fields in the data areas are also available in the `NaturalDataArea` objects. To take advantage of level 1 block optimization, you can include a method name in the first parameter.

Example of implementing level 1 block optimization

```
Dispatcher.CallNat "CUSTN.GET", custpda, custpda_id, custpdr, _  
                  cdaobj, cdpdam
```

where:

GET Is the method name appended to the subprogram name. Use a period (.) to separate the two. Only the blocks specified in the method definition are sent to the server in the request data.

Use the `CallSystem` method to send system commands to the Spectrum dispatch service or to invoke an arbitrary proxy.

Syntax of the CallSystem method

```
Function CallSystem (ByVal DomainName As String, _  
                   ByVal ObjectName As String, _  
                   ByVal Version As Long, _  
                   ByVal MethodName As String, _  
                   ByVal SendData As String, _  
                   ByRef ReceiveData As String) As Boolean
```

where:

CallSystem Is the method that allows you to send system commands directly to the Spectrum dispatch service or invoke an arbitrary subprogram proxy by specifying its domain, object, version, and method.

Timeout, Retry, and Resume Handling

The CallNat and CallSystem methods do not return until the server sends back a response. In effect, your calling application is locked up while the server is processing the request.

If the server does not respond, your application may not regain control and the user will have to terminate the application. For this reason, the Dispatcher object has a request timeout. The timeout indicates the maximum number of seconds to wait for the server to respond. When the specified number of seconds elapse, the dispatcher does one of two things:

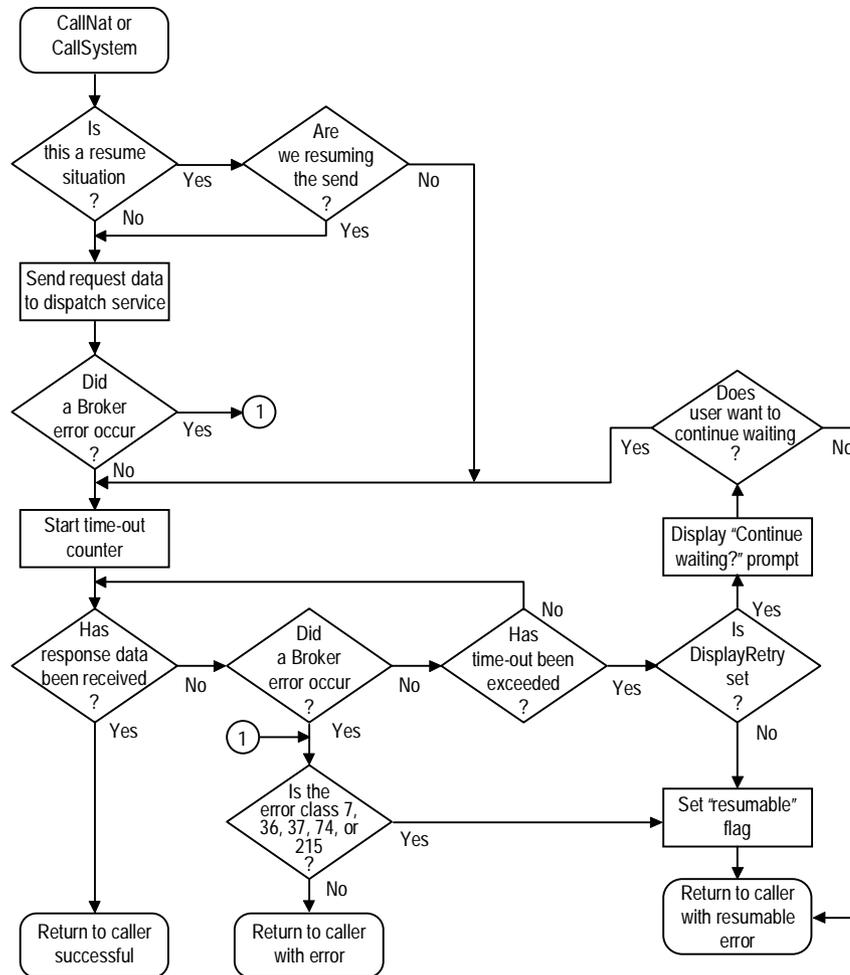
- Returns control to your calling application.
- or
- Asks the user whether or not to continue waiting.

Use the DisplayRetry property to tell the Dispatcher object what to do. To return control to your calling application, set DisplayRetry to False. To ask the user whether to continue waiting, set DisplayRetry to True and, optionally, set the RetryMessage property to a message string that is displayed to a user. The default message is: “The server is not responding. Would you like to continue waiting?”

The Timeout property determines the timeout duration in seconds and can be set to any value from -1 to 32767. Zero (0) returns control to the client application immediately. Negative one (-1) is the default and uses the timeout value specified in the dispatch service definition.

Tip: You can change this timeout value using the Spectrum Service Manager. For information, see **Using Construct Spectrum Tools**, page 109, *Construct Spectrum Administration*.

The following flowchart illustrates the life-cycle of a full request and response combination:



Life Cycle of a Full Request/Response Combination
Showing Timeout Functionality

This example illustrates the SDC's ability to resume the processing of a request because of a timeout or a recoverable EntireX Broker error.

Some EntireX Broker errors, such as resource shortages or a temporary interruption in Entire Network, are recoverable. If such an error occurs in the middle of processing a request, either when sending the request data to the server or receiving the response data from the server, the SDC can return the error to the calling application, which can then decide whether to resume the request or not.

To determine if the request is resumable, check the `RetryPossible` property after returning from the call. If this property returns `True`, you may set the `Retry` property to `True` and then reissue the call.

Example of resuming a call

```

Do
    If .CallNat("CUSTN.GET", custpda, custpda_id, custpdr, _
                cdaobj, cdpdam) Then
        ' Request was successful.
        Exit Do
    Else
        smsg = "The following error occurred: " & _
                .ErrorSource & ":" & .ErrorNumber & " - " & _
                .ErrorMessage & vbLf & vbLf & _
                "Click OK to try again or Cancel to quit."
        If MsgBox(smsg, vbOkCancel) = vbCancel Then
            Exit Do
        End If
        If .RetryPossible Then .Retry = True
    End If
Loop

```

If an error occurs in this example, the error message is displayed to the user, along with a prompt asking if the user wants to try again. If the user chooses to try again, the same call is performed.

What happens during the second call depends on the setting of the `Retry` property. If the error is resumable (`RetryPossible` is `True`), set `Retry` to `True` and the previous request is resumed. If the error is not resumable (`RetryPossible` is `False`), the second call initiates an entirely new request.

Compression and Encryption

The SDC can compress or encrypt the request data it sends to the Spectrum dispatch service.

Compression can significantly reduce the size of the data. This can reduce the transmission time, especially over slow network connections such as dialup connections. The compression algorithm reduces sequences of repeating characters, which are quite common when the request and response data contain partially-filled Natural data areas.

To enable compression, set the `Compress` property to `True`. To enable encryption, set the `Encrypt` property to `True`. These properties remain set until you change them.

Note: These properties only compress and encrypt the request data sent from the client to the Spectrum dispatch service. The decision to compress or encrypt the response data is made in the subprogram proxy on the server.

Tracing

Tracing options allow you to track the data transmission to and from the server. You set these tracing options, depending on the type of data you want to trace, by setting the properties of the Dispatcher object.

The following Dispatcher object properties are available to set trace options:

- TraceOption array property with indices 1 to 15.
- TraceCommand string property.
- TraceAutoReset Boolean property, which automatically resets the trace options after the call to the Spectrum dispatch service.

For more information about setting tracing options and understanding the result, see **Debugging Your Client/Server Application**, page 161.

Database Transaction Control

Each request sent to the Spectrum dispatch service can be handled by a different copy of the Spectrum dispatch service. While processing a request, you have exclusive access to the server. Once the server sends the response data back to the client, the server is available for your next request or a request sent by someone else.

The SDC also gives you exclusive access to a specific server across more than one request. To have this exclusive access, you must specify when you want to start having exclusive access to a server and when you are finished with it. While you have exclusive access, the server is dedicated to your client application and only accepts requests from you. No other client application can send requests to that server (unless you pass a reference to the Dispatcher object for another client application). Try to release the server as soon as possible, as you are preventing others from using it and there may be a limited number of servers running.

When you have exclusive access to a server, you can also issue END TRANSACTION or BACKOUT TRANSACTION statements from the client application and be assured that only your requests are affected. The Dispatcher class has three methods and one property to support exclusive use of a server:

Method or Property	Description
StartTransaction method	Tells the Dispatcher object that you want exclusive access to a server.
Commit method	Sends a request to the server to issue an END TRANSACTION statement and releases the server.
Abort method	Sends a request to the server to issue a BACKOUT TRANSACTION statement and releases the server.
TransactionActive property	Returns True if you have exclusive access to a server.

Each Spectrum dispatch service has a transaction timeout value that ensures a client application does not have exclusive access to the server for too long. The timeout period begins as soon as the server sends the response data back to the client application. If the client application does not send any more requests to the server within the timeout period, the server issues a BACKOUT TRANSACTION statement and returns to the server pool. If this happens, the client application is not notified until it tries to send the next request. The request fails with an `sdcerrTransactionTerminated` error.

Note: The transaction timeout period is set on the Maintain Services panels in the Construct Spectrum Administration subsystem.

To prevent transaction timeout, try to send all requests in succession and then release the server. If your application interacts with the user between requests (or if an error occurs and you display it to the user), there is a greater possibility of transaction timeout occurring because the user may not respond immediately.

The server is also automatically released when the Dispatcher object is destroyed (after all object references to it are released).

Error Reporting

Errors that can occur in the Dispatcher object include:

Error Types	Description
Runtime errors	Raised using the standard OLE automation error handling mechanism. For more information about runtime errors, see Deploying Your Client/Server Application , page 189, or <i>Construct Spectrum Messages</i> .
Communication errors	Occur during a remote call from the client application to the subprogram. Error details are returned in the error properties of the Dispatcher object: <code>ErrorSource</code> , <code>ErrorNumber</code> , <code>ErrorMessage</code> , <code>ErrorValues</code> , and <code>Successful</code> . For more information about communication errors, see Deploying Your Client/Server Application , page 189, or <i>Construct Spectrum Messages</i> .

User Identification and Authentication

The Application object has UserID, Password, and Language properties. These properties must be set before the first request is sent to the server, but may be changed at any time after that. These properties are:

Application Properties	Description
UserID	Identifies who you are to the server.
Password	Provides authentication of your user ID.
PasswordEmpty	Returns whether the Password property is set.
Language	Identifies internationalized servers.

If the server uses security, it can authenticate the user ID and password for each request and then check whether the user has the necessary permissions to execute the request. If the server does not use security, any user ID and password assigned to these properties is ignored.

To indicate the spoken language, assign one of the Natural *LANGUAGE codes to the Language property. This code is sent to the server with each request. Whenever the server returns a message string, it looks up the correct translation based on the code.

Library Image Files and the Steplib Chain

Library image files (LIFs) are special text files that contain SDC definitions. Each LIF contains up to three different types of definitions:

- Data area definitions
For more information, see **Data Area Definitions**, page 198.
- Application service definitions
For more information, see **Application Service Definitions**, page 215.
- Steplib definitions

Syntax of the steplib definition

```
[StepLibs]  
CST441S  
SYSTEM
```

A steplib definition allows multiple applications to share a set of LIFs. Each application may have its own main library, which contains just the definitions specific to that application. Shared definitions can be placed in other LIFs, which can be included in each application's steplib chain.

When searching for data area and application service definitions, the SDC first examines the main library's LIF. If it does not find the definition there, it looks for a steplib definition in the file. If it finds the steplib definition, it examines the LIFs for the libraries in the steplib definition, beginning with the first LIF on the list.

Advanced Features

The following sections introduce two advanced features you can use when developing your applications. It includes:

Feature	Description
FieldRef Property	Defines objects as parameters without duplicating data areas to pass objects to a Natural CALLNAT.
1:V	Defines arrays with variable numbers of occurrences.

FieldRef Property

The CallNat method of the Dispatcher class only accepts NaturalDataArea objects as parameters to pass to subprograms. You can, however, pass individual fields to a subprogram in Natural code.

Example of passing individual fields to a subprogram

```
ASSIGN CBROWSEA.COUNT = 10
CALLNAT 'CUSTB' CBROWSEA.COUNT
          CBROWSEA.ROWS(*)
```

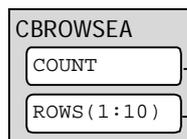
In this example, one field and all occurrences of an array are passed into a subprogram.

Example of how NOT to pass parameters to subprograms

```
Dim cbrowsea As NaturalDataArea

Set cbrowsea = SDCApp.Allocate("CBROWSEA")
cbrowsea.Field("COUNT") = 10
Dispatcher.CallNat "CUSTB", cbrowsea.Field("COUNT"), _
                          cbrowsea.Field("ROWS(*)")
```

In this example, the field values in CBROWSEA are passed to the Dispatcher object's CallNat method.



```
Dispatcher.CallNat "CUSTB", cbrowsea.Field("COUNT"), cbrowsea.Field("ROWS.(I)")
```

CBROWSEA Fields Passed to the CallNat Method

The problem with this example is that the Field property returns a value, not an object. The second and subsequent parameters to the CallNat method must be NaturalDataArea objects. Because the Field property returns a value, the CallNat method encounters a runtime parameter type mismatch error.

Note: The reason the CallNat method accepts only objects as parameters is so the dispatcher can maintain references to the objects and update them when the response comes back from the server.

A better way to simulate Natural code is to create separate NaturalDataArea objects for each parameter you are sending to the subprogram. The following example illustrates these differences:

Example of creating separate NaturalDataArea objects for each parameter

```
Dim cbrowsea As NaturalDataArea
Dim mycount As NaturalDataArea
Dim myrows As NaturalDataArea

Set cbrowsea = SDCApp.Allocate("CBROWSEA")
Set mycount = SDCApp.Allocate("CBA-C")           ' 01 COUNT(N3)
Set myrows = SDCApp.Allocate("CBA-R")           ' 01 ROWS(A32/1:10)

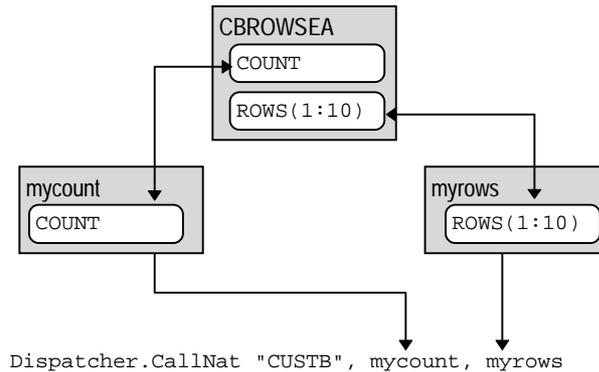
cbrowsea.Field("COUNT") = 10

' Copy the CBROWSEA fields into the temporary data areas.
mycount.Field("COUNT") = cbrowsea.Field("COUNT")
myrows.Field("ROWS") = cbrowsea.Field("ROWS")

Dispatcher.CallNat "CUSTB", mycount, myrows

' Copy the fields from the temporary data areas back into CBROWSEA.
cbrowsea.Field("COUNT") = mycount.Field("COUNT")
cbrowsea.Field("ROWS") = myrows.Field("ROWS")
```

In this example, the two newly-defined objects (mycount and myrows) are NaturalDataArea objects containing copies of COUNT and ROWS respectively. These two objects are then passed into the CallNat method:



CBROWSEA Fields Defined as Objects to the CallNat Method

This example shows how to create additional data areas for the individual fields passed to the subprogram. These data areas must be initialized from the CBROWSEA data area before issuing the CALLNAT. After the CALLNAT, the data areas must be copied back into CBROWSEA. This code looks quite different from the original Natural code.

A better solution is to create a pointer to a field within a NaturalDataArea object and pass that pointer to the CallNat method of the Dispatcher object, effectively passing the field by reference. Construct Spectrum provides the FieldRef property to do that. This property of the NaturalDataArea class creates an instance of the NaturalDataArea object that does not contain its own data, but rather points to a field in the data area that created it.

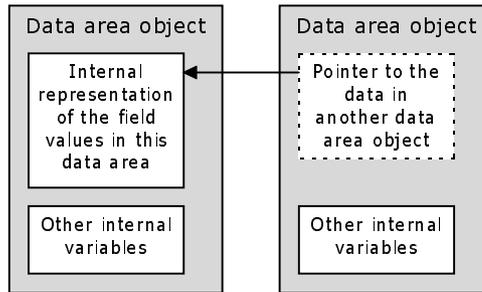
Syntax of the FieldRef property

```
Function FieldRef (ByVal FieldName As String) As NaturalDataArea
```

where:

FieldName Is the field the FieldRef property points to.

The FieldRef property creates a new instance of the NaturalDataArea object with the same field definitions as the field indicated by FieldName. However, any time a field in the new data area is read or written, the field in the original data area is accessed. This effectively creates two data areas referring to the same data.



Using the FieldRef Property to Create Two Data Areas that Refer to the Same Data

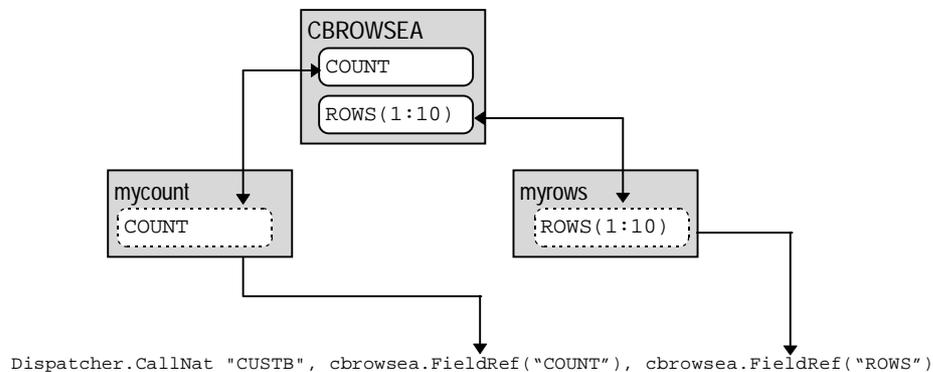
You can now rewrite your original Visual Basic code using the FieldRef property instead of the Field property.

Example of using the FieldRef property

```
Dim cbrowsea As NaturalDataArea

Set cbrowsea = SDCApp.Allocate("CBROWSEA")
cbrowsea.Field("COUNT") = 10
Dispatcher.CallNat "CUSTB", cbrowsea.FieldRef("COUNT"), _
    cbrowsea.FieldRef("ROWS")
```

In this example, the FieldRef property creates a temporary NaturalDataArea object that is passed to the CallNat method. Each temporary data area contains a pointer to the original data.



Using the FieldRef Property to Create Two Data Areas that Refer to the Same Data

Example of using the CUSTA Natural data area

```

01 CUSTOMER-NUMBER(N5)
01 FIRST-NAME(A20)
01 LAST-NAME(A20)
01 MAILING-ADDRESS
  02 STREET(A30)
  02 CITY(A20)
  02 PROVINCE(A20)
  02 POSTAL-CODE(A6)
01 SHIPPING-ADDRESS
  02 STREET(A30)
  02 CITY(A20)
  02 PROVINCE(A20)
  02 POSTAL-CODE(A6)

```

You can use the FieldRef property to obtain a pointer to the Mailing-Address or Shipping-Address structures so you can process them individually, as the following example shows:

Example of Visual Basic code

```

Dim mycust As NaturalDataArea
Dim mycustref As NaturalDataArea

Set mycust = SDCApp.Allocate("CUSTA")

For i = 1 To 2
  If i = 1 Then
    Set mycustref = mycust.FieldRef("MAILING-ADDRESS")
  Else
    Set mycustref = mycust.FieldRef("SHIPPING-ADDRESS")
  End If

  ' At this point, mycustref is an alias to either the mailing
  ' address or the shipping address fields of the mycust data area.

  With mycustref
    Print .Field("STREET")
    Print .Field("CITY")
    Print .Field("PROVINCE")
    Print .Field("POSTAL-CODE")
  End With
Next

```

This example did not specify the level 1 structure name to qualify the field name when reading the Street, City, Province, or Postal-Code fields. This is because the Natural-DataArea object returned by the FieldRef property only contains definitions for the Mailing-Address or Shipping-Address fields.

1:V Fields

In a Natural parameter data area, you may specify an array with a variable number of occurrences by using the index notation 1:V.

Example of specifying an array

```
01 #ROWS(1:V)
02 ...
```

Some of the library image files may already contain similar data area definitions. However, you must specify the number of occurrences for each V to create an instance of this data area. Specify the number of occurrences by using the optional VSubstitutions parameter when you call the Application.Allocate method.

Example of specifying the number of occurrences for your array

```
Function Allocate (ByVal DataAreaName As String, _
                  ParamArray VSubstitutions() As Variant) _
                  As NaturalDataArea
```

For your arrays to operate successfully, you must provide a value for each V in the data area definition or a runtime error will occur. The parameters following DataAreaName in the Allocate call are called a V substitution list. The following examples illustrate an Allocate call.

Example of a PDA

```
[TESTPDA]
01 PARM1(A5)
01 PARM2(A3/1:V,1:V)
01 PARM3(1:V)
02 PARM4(N3/1:V)
```

Example of instantiating the PDA

```
[TESTPDA]
01 PARM1(A5)
01 PARM2(A3/1:10,1:5)
01 PARM3(1:20)
02 PARM4(N3/1:5)
```

Example of calling the Allocate method

```
Set nda = SDCApp.Allocate("TESTPDA", _
                          "PARM2", 10, 5, _
                          "PARM3", 20, _
                          "PARM4", 5)
```

In this example, the V substitution list consists of groups of parameters. Each group identifies a field and provides the substitution values for the 1:V specifications for that field. There must be as many groups as there are fields with 1:V specifications.

You can also store the V substitution list in an array and pass the array as a parameter to the Allocate method.

Example of passing the array to the Allocate method

```
Dim vlist(1 To 7) As Variant

vlist(1) = "PARM2": vlist(2) = 10: vlist(3) = 5
vlist(4) = "PARM3": vlist(5) = 20
vlist(6) = "PARM4": vlist(7) = 5

Set nda = SDCApp.Allocate("TESTPDA", vlist)
```

When you read the FieldDef property to obtain the lower and upper bounds of an array defined with 1:V, 1 is returned for the lower bound and the value specified for that field's V is returned for the upper bound.

Example of obtaining the upper and lower bounds of an array

```
With nda.FieldDef("PARM4")
    Print .FromIndex(1) & ":" & .ThruIndex(1)      ' Prints "1:20"
    Print .FromIndex(2) & ":" & .ThruIndex(2)      ' Prints "1:5"
End With
```

You can create instances of the same data area with different numbers of occurrences.

Example of using the same data area with varying numbers of occurrences

```
Dim data1 As NaturalDataArea
Dim data2 As NaturalDataArea

Set data1 = SDCApp.Allocate("#ROWS", 15)
Set data2 = SDCApp.Allocate("#ROWS", 100)

With data1.FieldDef("#ROWS")
    Print .FromIndex(1) & ":" & .ThruIndex(1)      ' Prints "1:15"
End With

With data2.FieldDef("#ROWS")
    Print .FromIndex(1) & ":" & .ThruIndex(1)      ' Prints "1:100"
End With
```

Note: The size of the data area must be no greater than allowed by Natural.

CREATING APPLICATIONS WITHOUT THE FRAMEWORK

This chapter describes how to create a Construct Spectrum application without using Construct-generated framework components. Working through the steps of creating a simple application, you will learn how to create and deploy your application. While the simple application is designed to run using Microsoft's Visual Basic, you can use any other development tool that fully supports OLE automation.

The following topics are covered:

- **Setting Up the Server Components**, page 236
- **Generating Subprogram Proxies**, page 240
- **Creating the Library Image Files (LIFs)**, page 244
- **Developing the Client Application**, page 248

For information about creating Construct Spectrum applications using components generated using earlier versions of Natural Construct, see **Moving Existing Applications to Construct Spectrum**, page 201, *Construct Spectrum SDK for Client/Server Applications*.

Setting Up the Server Components

The following sections describe how to set up the server-side components to prepare an environment for the client to be able to communicate with the server. You can create the server-side components entirely within the Natural environment.

Create or Select Application Services

When creating new application services or selecting existing services for deployment in a client/server environment, ensure that the Natural subprograms follow certain rules. Natural subprograms primarily execute as remote services in environments where no input and output devices are defined. Therefore, there are some restrictions imposed on your Natural programs.

The following sections identify issues to consider when you are developing new application services or adapting existing services for a client/server environment.

No Terminal I/O

Avoid the use of all commands that require input from the user or write information to any external source other than a database file. This includes the INPUT statement as well as the WRITE, PRINT, and DISPLAY statements.

While the INPUT statement cannot be used to input data from the user, you can use the statement to retrieve data that was stacked using the STACK TOP DATA statement.

Only use the WRITE, PRINT, and DISPLAY statements to write information to the Natural source area for the purpose of debugging your application. If you are running servers in batch mode, you can send these statements to the batch output queue. The data is only viewable after the batch job ends. For information, see **Debugging Your Client/Server Application**, page 161.

Subprogram Interface

Construct Spectrum is only able to communicate with application services that are implemented as subprograms. If necessary, you may invoke programs from inside the called subprograms by using the FETCH RETURN statement.

No Global Data Area (GDA)

Called services do not normally define a global data area (GDA), as the contents of the GDAs used by a subprogram are not preserved between calls. However, you can use GDAs to overcome a shortage of local data area (LDA) storage when necessary.

Parameter Data Area (PDA) Data Size Limitation

All data transmitted between the client and server is converted into printable characters. For example, an I2 integer requires 6 bytes of data during transmission: a sign byte and five digits. The size of this converted data cannot exceed 32K. When checking or cataloging a subprogram proxy, Natural displays an error if the size of the converted data exceeds 32K.

Subprogram Behavior

All subprograms invoked as application services must return to the calling routine. The subprogram, and any called routines, cannot execute `STOP`, `FETCH`, or `TERMINATE` statements. They should also avoid statements that affect the caller, such as `RELEASE STACK`, `STACK COMMAND`, `STACK DATA`, and `RELEASE VARIABLES`. Called subprograms should not modify the `*ERROR-TA` value.

Externalize Parameters

The best design strategy is to externally define the parameters of your subprogram proxies. Only when the parameters are externally defined can the parameter data areas (PDAs) be downloaded and incorporated into the Spectrum Dispatch Client.

However, Construct Spectrum does allow you to generate subprogram proxies for subprograms that define their parameters inline. If the subprogram accepts or returns large amounts of data that is strictly input parameters or output parameters, consider grouping all input parameters into one level 1 structure and all output parameters into another level 1 structure. Group parameters that are both input and output into a third level 1 structure. This allows data being sent between the client and the server to be optimized so that only input data is sent to the server and only output data is returned to the client.

Timing Issues

Some application services perform tasks that require extensive processing. The length of time spent in the application service affects the timeout values triggered in a client system. If an application requires extensive time to execute, it may be necessary to define and associate such long-running processes with Spectrum dispatch services that use inflated timeout values. These resource-intensive application services can execute under a specially-configured dispatch service. This includes defining special services in the EntireX Broker attribute files.

Example of Creating a Simple Natural Subprogram

This section describes how to create a small application service and generate the associated subprogram proxy.

➤ To create your server-based components for the sample application:

1 Create a parameter data area named GCDA.

Use the following configuration and compile the PDA in the SAMPLE library:

```

Parameter GCDA      Library SAMPLE                      DBID 17 FNR 38
Command                                                    > +
I T L Name                F Leng Index/Init/EM/Name/Comment
All -----
  1 GCD-DATA
  2 #OPERAND-1             I   4
  2 #OPERAND-2             I   4
  2 #RESULT                 I   4

----- Current Source Size: 143 Free: 43402 ----- S 4 L 1

```

Example of GCDA Parameter Data Area

For an example of this module, refer to SAMPLE_A in the SYSSPEC library.

- 2 Type the following code in the Natural editor and compile it as a Natural subprogram named GCDN in the SAMPLE library:

```

>
> + Subprogram GCDN Lib SAMPLE
Top .....1.....2.....3.....4.....5.....Mode Struct..
0010 *****
0020 ** This module accepts numbers as input paramters and returns the
0030 ** greatest common divisor as the result.
0040 *****
0050 DEFINE DATA
0060 PARAMETER USING GCDA /* Input and output parameters
0070 LOCAL
0080 01 #TEMP(I4) /* Local variable used in calculation
0090 END-DEFINE
0100 **
0110 ** Repeat while the second operand value is not equal to 0.
0120 REPEAT
0130 WHILE #OPERAND-2 NE 0
0140 DIVIDE #OPERAND-2 INTO #OPERAND-1 REMAINDER #TEMP
0150 ASSIGN #OPERAND-1 = #OPERAND-2
0160 ASSIGN #OPERAND-2 = #TEMP
0170 END-REPEAT
0180 **
0190 ASSIGN #RESULT = #OPERAND-1
0200 END
.....Current Source Size: 799 Char. Free: 42970 ..... S 21 L 1

```

Example of GCDN Natural Subprogram

For an example of this module, refer to SAMPLE_N in the SYSSPEC library.

Generating Subprogram Proxies

The following sections describe how to make the newly-created GCDN Natural subprogram accessible from the client. To do this, generate a subprogram proxy using the Subprogram-Proxy model.

Subprogram-Proxy Model

The Subprogram-Proxy model is available in the Generation subsystem on the server and as a model wizard in the Construct Windows interface. In the example, the model wizard is used.

For more information about using the model, see **Using the Subprogram-Proxy Model**, page 103.

- To generate a subprogram proxy for the GCDN subprogram:
 - 1 Access the Standard Parameters window for the Subprogram-Proxy wizard.
 - 2 Enter the following information in the window:

Note: Many of the input values for the Subprogram-Proxy model are automatically determined and set by the model itself.

The screenshot shows the 'SUBPROGRAM-PROXY Wizard' dialog box, specifically the 'Standard Parameters' step. The dialog has a title bar with a question mark and a close button. On the left, there is a vertical sidebar with three steps: 'Start', 'Standard Parameters' (which is selected and highlighted), and 'Finish'. The main area is titled 'Enter the standard parameters for this model' and contains several input fields and checkboxes. The fields are: 'Module:' with the value 'GCD'; 'System:' with the value 'DEMO'; 'Title:' with the value 'Subprogram proxy for GCD'; 'Description:' with the text 'This subprogram proxy communicates with the module that calculates the greatest common divisor of two numbers'; 'Subprogram:' with the value 'GCDN' and a browse button (...); 'Domain:' with the value 'SAMPLE' and a browse button (...); 'Object name:' with the value 'GCDN'; and 'Version:' with the value '1.1.1'. There is also an 'Edit 1:V Overrides' button. At the bottom, there are five buttons: 'Validate', 'Cancel', '< Back', 'Next >', and 'Finish'. There are also three checkboxes: 'Generate trace code', 'Compress network data', and 'Encrypt network data', all of which are currently unchecked.

Subprogram-Proxy Wizard — Standard Parameters

- 3 Generate and stow the GCD subprogram proxy.
Generation creates two new items:
 - the generated GCD subprogram proxy
 - the application service definition (generated into the Construct Spectrum Administration subsystem)

Application Service Definition

The application service definition is automatically created when you generate a subprogram proxy using the Subprogram-Proxy model. It defines the name and location of the target subprogram to the Spectrum dispatch service, as well as which methods the target subprogram supports.

The target subprogram can be any Natural subprogram, although there are many advantages to creating the subprogram using the Object-Maint-Subp and Object-Browse-Subp models.

To tailor your application service definition, use the following procedure. The example uses the GCD subprogram proxy.

- To customize the generated application service definition:
 - 1 Access the Construct Spectrum Administration subsystem main menu.
 - 2 Enter “AA” in the Function field.
The Application Administration main menu is displayed.
 - 3 Enter “MM” in the Function field.
The Application Administration Maintenance menu is displayed.
 - 4 Enter “AS” in the Function field.
The Maintain Application Service Definitions panel is displayed.
 - 5 Type the following values in the fields indicated:
 - “D” in the Action field
 - “SAMPLE” in the Domain field
 - “GCD” in the Object field
 - “01/01/01” in the Version field

- 6 Press Enter.
The application service definition associated with GCD is displayed:

```

BSIF__MP          Construct Spectrum Administration Subsystem          BSIF__11
June 27           Maintain Application Service Definitions             3:15 PM

Action (A,B,C,D,M,N,P)  _

Domain.....: SAMPLE_____ *
Object.....: GCDN_____
Version.....: 01 / 01 / 01
Description.....: GCDN_____
Default subprogram proxy: GCD_
Steplib.....: _____ *

01           Method Name          Subprogram          Steplib *
-----
1  DEFAULT_____
2  _____
3  _____
4  _____
5  _____

Command: _____
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
confm help retrn quit          flip pref bkwrd frwr          main
Appl Srvc Definition DEMO-PRODUC displayed successfully
    
```

Example of the Application Service Definition Panel

The application service definition for GCD was generated with one method — Default. The Default method is generated automatically for each application service definition unless the target Natural subprogram was generated using either the Object-Maint-Subp or Object-Browse-Subp model.

The definition does not specify a steplib, although one is required to access the target subprogram. Because the specified domain, SAMPLE, has a steplib defined, the application service definition also uses SAMPLE by default. For more information, see **Step 1: Define the Steplib Chain**, page 43.

Creating the Library Image Files (LIFs)

Before you can call the target subprogram from the client application, you must create a file on the client that describes the subprogram and any parameter data areas it uses. This file is called a library image file (LIF) because it contains an image (or a copy) of the Natural objects in your application library on the server platform. Definitions of all objects used in the client application must be in the LIF. If your client application uses objects from multiple libraries on the server, you must create one LIF for each library.

The file name for a LIF is the same as the name of the library, plus the extension .LIF. For example, the LIF for the CSTDEMO library is CSTDEMO.LIF. All LIFs the client application uses must be stored in the same LIF directory on your PC (or on a network file server).

Each application can have its own LIF directory, or multiple applications can share a single LIF directory containing many different LIFs.

The following sections describe how to use the Construct Spectrum Add-In in Visual Basic to create and tailor LIFs for your application. Use the Download function to download LIF definitions from an application library on the server to your client.

Construct Spectrum Add-In

Use the Construct Spectrum Add-In to download the subprogram and parameter data area (PDA) definitions to a library image file.

Before You Start

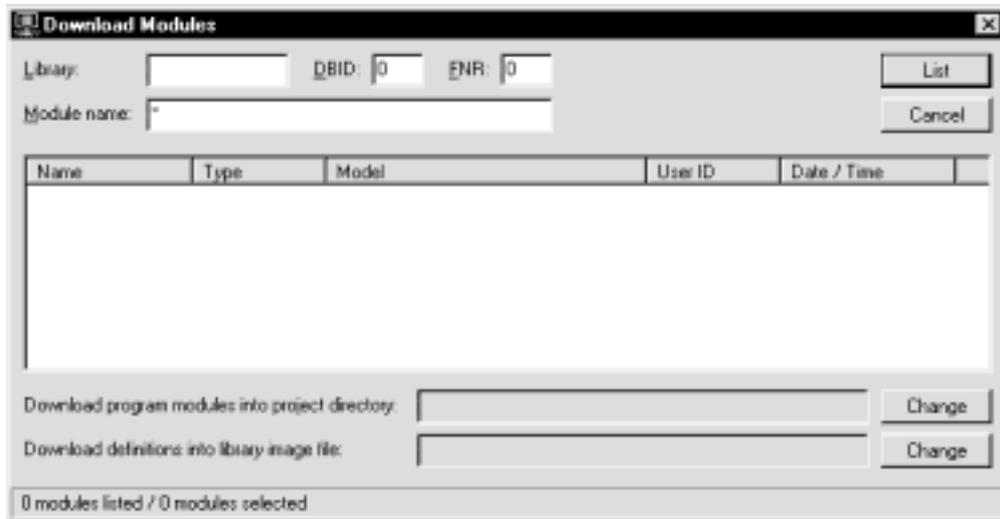
- Ensure that you know the library name, database ID (DBID), and file number (FNR) of the FUSER system file. This file resides on the server and contains the subprogram you created earlier in an application library in the FUSER file.
- Choose or create a LIF directory on your PC for the library image file.

Note: You will create the client application in this directory in a later step.

- Ensure that you know the name of the subprogram proxy and all PDAs used by the subprogram.
- Ensure that the Spectrum Dispatch Client is installed and configured properly. For information, see the *Construct Spectrum and SDK Installation Guide for Windows*.

Download Definitions

- To download the definitions:
- 1 Start Visual Basic if it is not already running.
 - 2 Select Construct Spectrum > Download Generated Modules from the Visual Basic Add-Ins menu.
The Download Modules window is displayed:



Download Window

- 3 Type the following values in the fields indicated:
 - Name of the application library in Library
 - Database ID in DBID
 - File number of the FUSER file containing the library in FNR

Note: If you have already used the Download function, the DBID, FNR, and library name used for that download are filled in automatically.

- 4 Enter “GCD*” in Module name to list all modules beginning with “GCD” in the library:

Name	Type	Model	User ID	Date / Time
------	------	-------	---------	-------------

Searching the Module Field in the Download Window

- 5 Click List.
After a few seconds, a list of the modules matching the wildcard pattern is displayed. If an error message is displayed, see **Downloading the Client Modules**, page 188, *Construct Spectrum SDK for Client/Server Applications*, or *Construct Spectrum Messages*.

Note: Only subprogram proxies and PDAs are displayed. Other Natural object types, such as programs, maps, and copy code members, are not displayed because they cannot be downloaded.

- 6 Select all subprogram proxies and PDAs associated with your subprogram.

Tip: To select more than one item on the list, use the standard Windows multiple-select actions (Shift-Click and Ctrl-Click), or use the mouse to drag a marquee around the items you want to select.

- 7 Click Download.
The selected modules are downloaded.
- 8 Click Close to close the Download Modules window.

If an error message is displayed during the download process, see *Construct Spectrum Messages* for information about resolving the error.

The download process creates a new library image file in the LIF directory or updates an existing LIF. The following section describes how to develop a client application that uses the LIF definitions to call the subprogram on the server.

Developing the Client Application

This section describes the minimum requirements to develop a client application that calls your subprogram. Although the example uses Microsoft's Visual Basic for development, you can use any development tool that supports OLE automation.

This section assumes you are familiar with the following OLE automation concepts. If any of these are unfamiliar, refer to the appropriate documentation for the development tool you are using:

OLE Automation Term	Definition
object library (or type library)	Provides definitions of all the objects, methods, and properties exposed by an OLE automation server.
externally creatable object	Object exposed by an OLE automation server that can be created outside the server.
dependent object	Object exposed by an OLE automation server that can only be accessed using a method of a higher-level object, such as an externally-creatable object.

- To develop your client application:
 - ❑ **Step 1: Create a New Project**, page 249
 - ❑ **Step 2: Add a Reference to the SDC Object Library**, page 249
 - ❑ **Step 3: Write Code to Initialize the SDC**, page 250
 - ❑ **Step 4: Create the User Interface**, page 251
 - ❑ **Step 5: Write Code to Call the Subprogram**, page 252
 - ❑ **Step 6: Run the Application**, page 253

The following sections describe each of these steps in more detail.

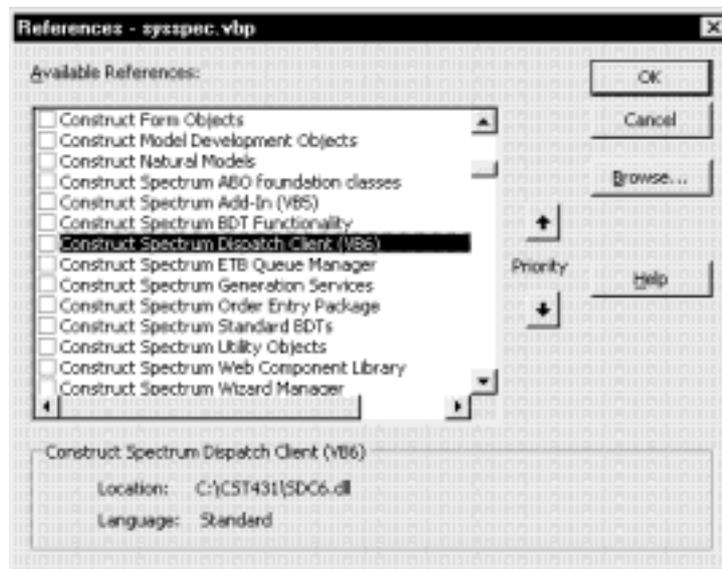
Step 1: Create a New Project

- To create a new project:
 - 1 Start Visual Basic.
 - 2 Create a new Standard EXE project.
Save all project components in the LIF directory you created earlier. This makes keeping track of all project components easier.

Step 2: Add a Reference to the SDC Object Library

Before you can use objects in the Spectrum Dispatch Client (SDC), you must add a reference to its object library in your Visual Basic project.

- To add a reference to the object library for the Spectrum Dispatch Client:
 - 1 Select References from the Tools menu.
The References window is displayed.
 - 2 Ensure that the Construct Spectrum Dispatch Client (VB6) is selected:



References Window

- To view the object library:
 - 1 Select Object Browser from the View menu.
The Object Browser window is displayed.
 - 2 Select SDCLib from Libraries/Projects.

Step 3: Write Code to Initialize the SDC

➤ To initialize the SDC:

- 1 Select Add Module from the Project menu.
A new module is added to your Construct Spectrum project.
- 2 Add the following code to the module:

```
Public SDCApp As New SDCLib6.Application
Public Dispatcher As SDCLib6.Dispatcher
Public Sub Main
    SDCApp.Initialize App.Path, "CSTDemo"
    SDCApp.UserID = "GUEST"

    Set Dispatcher = SDCApp.CreateDispatcher()
    Dispatcher.DisplayErrors = True

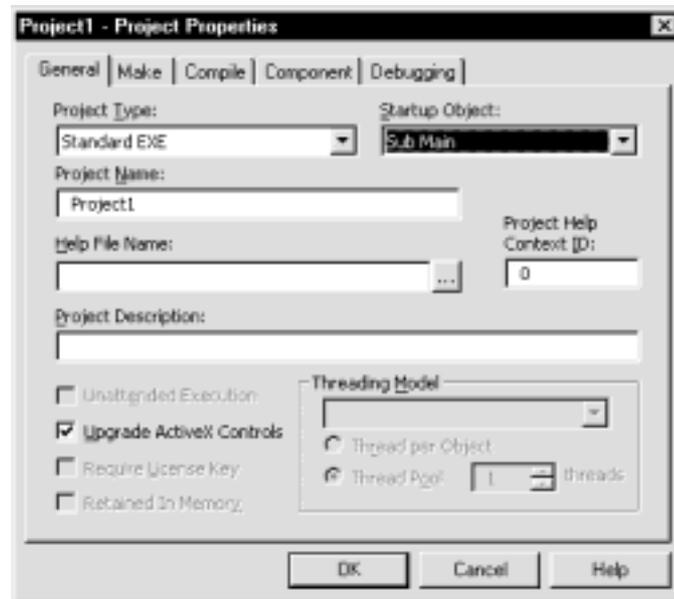
    Form1.Show
End Sub
```

where:

Application	Is an externally-creatable object exposed by the Spectrum Dispatch Client. It is used to create all other objects.
App.Path	Is a Visual Basic property that returns the name of the directory containing your saved project. In this example, the project is stored in the LIF directory. App.Path returns the name of the directory where your executable project is located. Your library image file must always be in that directory.
Dispatcher	Is an object used to communicate with the server platform.
Form1	Contains the user interface for the client application.
Initialize method	Tells the SDC the name of the LIF directory and name of the application library. Together, these two values tell the SDC the name of the library image file.
Set Dispatcher = CreateDispatcher()	Creates a Dispatcher object with methods that allow you to call the subprogram on the server. If the DisplayErrors property is set to True, the Dispatcher object automatically displays communication errors. You do not have to write additional code to display errors.
GUEST	Is a predefined user ID in the Construct Spectrum Administration subsystem containing the required security definitions for this example. Every call to the server platform requires the caller's user ID to be known.

This code creates two object variables used throughout the application: SDCApp and Dispatcher.

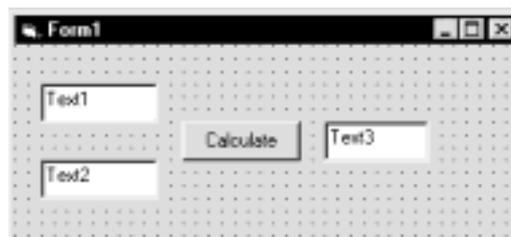
- 3 Select <App.Name> Properties from the Project menu.
The General tab in the Project Properties window is displayed.
- 4 Select “Sub Main” from the Startup Object field:



Project Properties Window

Step 4: Create the User Interface

- To create the user interface for the client application:
- 1 Ensure Form1 is open in design mode.
 - 2 Add three TextBox controls and a CommandButton control, arranged as follows:



Example of the Layout of Form 1

- 3 Set the control properties as follows:

Control	Property	Value
Text1	(Name)	txtOperand1
	Text	<empty>
Text2	Type	txtOperand2
	Text	<empty>
Text3	Type	txtResult
	Text	<empty>
CommandButton	Type	cmdCalculate
	Caption	Calculate

Step 5: Write Code to Call the Subprogram

- To code the Click event of the command button to call the subprogram:
 - 1 Double-click the Command button.
The Code window is displayed.
 - 2 Add the following code to the cmdCalculate_Click procedure:

```
Private Sub cmdCalculate_Click()
    Dim parms As NaturalDataArea

    Set parms = SDCApp.Allocate("GCDA")

    parms("#OPERAND-1") = Val(txtOperand1.Text)
    parms("#OPERAND-2") = Val(txtOperand2.Text)

    Screen.MousePointer = vbHourglass

    Dispatcher.CallNat "GCDN", parms

    Screen.MousePointer = vbDefault

    If Dispatcher.Successful Then
        txtResult.Text = parms("#RESULT")
    End If
End Sub
```

This code declares and allocates a Natural data area corresponding to the PDA expected by your subprogram. Next, it assigns the numeric values in txtOperand1 and txtOperand2 to the #OPERAND-1 and #OPERAND-2 fields in the data area. It then calls the subprogram with the CallNat method for the Dispatcher object. The mouse pointer changes to an hourglass icon for the duration of this call. Finally, if the call is successful, the contents of the Result field are displayed in txtResult. If the call is unsuccessful, the Dispatcher object automatically displays the error message (because you set the DisplayErrors property to True).

Step 6: Run the Application

Note: Before running the application, save the project. This ensures that the App.Path property in Sub Main returns the correct directory for the Initialize call.

Tip: If you forget to save a new project, App.Path returns the working directory from which you started Visual Basic. However, when you save to disk, App.Path returns the name of the directory in which the project file is saved.

- To run the application:
- 1 Press F5.
 - 2 Type a number into each operand text box.
 - 3 Click Calculate.
The result is displayed in Result.

Note: The first call to the communication server platform will take a few seconds as the EntireX Broker DLLs must be loaded into memory and initialized. Subsequent calls are much faster.

The Dispatcher object may display the following error in the cmdCalculate_Click procedure:

```
Numeric overflow
```

Possible cause:

The value being assigned to the #OPERAND-1 or #OPERAND-2 field is too large for the Natural format.

Resolution:

Enhance the code to check that values entered by the user into the text boxes are not too large for the Natural format.

If the Dispatcher object displays an error, see **Debugging Your Client/Server Application**, page 161, or *Construct Spectrum Messages* for information about resolving the error.

For information about packaging the client application and installing it on another PC, see **Deploying Your Client/Server Application**, page 189.

- 4 Close the window to return to design mode when you are finished testing the application.

APPENDIX A: GLOSSARY

The following terms are used throughout the Construct Spectrum documentation set. Each term is listed with its meaning.

Term	Definition
active server page (ASP) script	Script that activates the WebApp.cls page handler, which opens the specified web page.
ActiveX business object (ABO)	Visual Basic class that represents a Natural business object on the client. The ABO wraps the Spectrum calls required to communicate with the Natural subprogram exposed by a subprogram proxy.
ActiveX DLL	Data link library containing one or more ABOs. It is used to package and deploy web applications.
application library	Natural library containing the server application components of a client/server application.
application service definition	Definition in the Construct Spectrum Administration subsystem that identifies the methods exposed by a subprogram. The definition is created automatically by the Subprogram-Proxy model. You can modify these settings on the Maintain Application Service Definition panel in the Construct Spectrum Administration subsystem.
application services	Natural subprogram implementing methods that can be called as remote services.
architecture	High-level description of the organization of functional responsibilities within a system. The architecture conveys information about the general structure of systems. It defines relationships between system components, but not the implementation of components.
browse command handler	Defines the commands linked to a browse dialog. It also acts as the initial target of commands, typically redirecting them to other application components. See also command handler , page 257.
browse data cache	Area containing database records returned from the server. Records are usually displayed in a browse dialog.

Term	Definition (continued)
browse dialog	Generic GUI browse window called to display any browse data residing on a mainframe or PC.
browse process	<p>Process by which framework components and generated browse components retrieve data and, optionally, display it in a browse dialog.</p> <p>For example, a browse process can retrieve rows of data, search for specific values, and then perform calculations and conditional processing. Users can display the results in a browse dialog, if desired.</p>
business data type (BDT)	<p>Type validation on the client that applies business semantics to a field. Typically, BDTs are used to format field data specified by the user.</p> <p>For example, if an application has an input field to enter a phone number, you can associate a BDT with the field to reformat the number with hyphens. A user can enter “7053332112”. When the user moves to the next field or performs another action, the number is automatically reformatted as 705-333-2112.</p> <p>Construct Spectrum supplies standard BDTs, which you can customize, or you can create your own. BDT modifiers are added to the keyword components of a field in Predict.</p>
BDT class	Collection of all BDT procedures.
BDT controller class	Collection of methods available to members of a BDT class. See also BDT class .
BDT controller object	Supplied client framework component that is an instance of the BDT controller class and uses the methods available to that class. Each application declares a BDT controller object, which records and maintains a list of names for each BDT and points to the BDT definition. See also business data type (BDT) , page 256.
BDT modifier	Additional logic users supply to modify the formatting or validation rules for a BDT. For example, <code>BTD_NUMERIC</code> ensures that only numeric values are entered in a field. You can also add a modifier to round numeric values. To increase flexibility, each BDT defines its own modifiers.
BDT procedure	Code that implements a BDT.

Term	Definition (continued)
business object	Conceptual abstraction that groups the attributes and behaviors associated with a business entity, such as Customer or Order. See also Visual Basic business object , page 269.
Business-Object-Super-Model model	Model (available in the Construct Windows interface and Generation subsystem) that generates multiple modules for both web or client/server applications that do not use the Construct Spectrum client framework.
Business-Object-Super-Model wizard	Wizard that generates maintenance and browse subprograms and subprogram proxies for business objects.
cardinality	Number of dimensions of information. Information with the same number of dimensions has the same cardinality.
child model	Individual model for which a super model (parent model) collects parameters and generates specifications.
client application	Portion of a Construct Spectrum client/server application that runs on a Windows platform.
client framework	Supplied set of cooperating Visual Basic classes that form a reusable design. It provides a skeleton of functionality, which you can customize or fill with generated and hand-coded Visual Basic modules. The client framework reduces the size of generated components and allows them to interact. It includes forms, classes, procedures, global variables, and constants that are shared among generated application components. It supplies both client and server components.
code block	One or more lines of code in a Visual Basic module that can be manipulated in the code editor as a block.
command block	Code block that tells the Natural Construct nucleus to treat the text within the block as a separate module and to apply the specified command to the block. Super models use command blocks to generate multiple modules.
command handler	Object, generally a Visual Basic class, that processes a command. The client framework calls command handlers when a user clicks a menu command or toolbar button. One command handler can handle multiple commands. See also command handler list , page 258, and hook , page 261.

Term	Definition (continued)
command handler list	List of command handlers for each command ID. The last command handler hooked to a command ID is called first. See also hook , page 261.
command ID	Unique identifier for an application-specific command sent when a user clicks a menu command or toolbar button. Define these commands by specifying a single command ID as “constant” for each unique menu and toolbar command.
complex redefine	Redefinition of a data area containing multiple data types, multiple redefinitions of a data field, or multiple levels of redefined fields.
compression	Reduce the byte size required to transmit data to and from the client and server. Data is compressed when it is sent and then decompressed when it reaches its destination. This reduces the size of data transmissions and improves network performance.
Construct Spectrum	Application consisting of a client and server component. The client component is a Construct Spectrum application running in Visual Basic. The server component is a set of subprograms accessed remotely by the client component.
Construct Spectrum Add-In	Customized functionality added to the Visual Basic environment.
Construct Spectrum Administration subsystem	Mainframe subsystem used to maintain and query tables defining Construct Spectrum application services and security.
database record	Logical view of database information. A database record can be comprised of one or more logically related database files or tables. Construct Spectrum represents database information in parameter data areas (PDAs).
DBID	Acronym for database ID, which is the number identifying the server database containing application components.

Term	Definition (continued)
debugging tools	Utilities you can use to locate and analyze logic errors. You can simulate client calls online and use traditional debugging tools, such as: <ul style="list-style-type: none">• Trace options, which allow you to save data from a client call to a file on the server and then use the data to recreate situations that caused errors.• Input and output statements, such as INPUT, PRINT, and WRITE, which allow you to step through the program for testing purposes.• Natural Debugging facility, which you can use to establish a debug environment. For information, see the Natural Debugging facility in the Natural documentation.
dependent object	Object exposed by an OLE automation server that can only be created using the method of a higher-level object. See also externally-creatable object , page 260.
deployment	Movement of an application from a development environment to a production environment.
dialog	GUI form running on the client.
dispatcher or dispatch service	Server component used to broker communications between server components and client framework components. See also Spectrum dispatch service , page 267.
domain	Entity that defines a collection of related business objects (for example, Test, Admin, and Sales).
double-byte character set (DBCS)	Related collection of characters in some non-Latin languages that require two bytes to display.
download data	Transfer (copy) modules from the server to the client.
encapsulation	Technique in object-oriented programming in which the internal implementation details of an object are hidden from users of the object. Methods control how the object data is manipulated. Encapsulation allows internal implementations to change without affecting the way an object is used externally.
encryption	Encoding data so it is unusable for individuals without access to the decryption algorithms. Construct Spectrum allows you to encrypt sensitive data, such as payroll information, during network transmission. Data is decrypted when it reaches its destination.

Term	Definition (continued)
Entire Broker service settings	Collection of Entire Broker-related parameters, including Entire Broker ID, server class, server name, and service.
Entire Broker stub	Entire Broker DLL on a Windows platform.
event	Action recognized by an object, such as pressing a key or clicking a mouse. You write code to respond to events.
externally-creatable object	Object exposed by an OLE automation server that can be created outside the server. See also dependent object , page 259.
field	Component of a database record. The term also refers to areas on a panel in which values are entered.
FNR	Acronym for the file number that identifies a specific server database file containing application components.
foreign key	Key field pointing to a record in an external file. For example, the demo application has an Order file containing a foreign key to the Warehouse field in the Warehouse file. Foreign keys can be set up with a browse function, enabling users to search for and select values.
form	<p>Window (dialog) that acts as the interface for an application. You add controls and graphics to a form to create the effect you want. Construct Spectrum supplies forms in the client framework and generates form modules for business object maintenance dialogs.</p> <p>When you run a project, forms are compiled into GUI dialogs that the user interacts with while using the application. Some forms, such as the generic BrowseDialog form, are dynamically configured at runtime by the client framework to alter the look of the form.</p> <p>Form definitions are saved in files with the extension .frm.</p>
form section	Portion of a web page containing a block of related information.
framework templates	Structure or container supplied for applications. These customizable templates include header, footer, navigation bar, messages area, and constants.
generate	Process of producing code from specifications.

Term	Definition (continued)
generated module	Generated component for either the client or server portion of an application. Generated server modules include Natural subprograms, subprogram proxies, and parameter data areas. Generated client modules include object factories, dialogs, and maintenance objects.
generation data cache	In-memory hierarchical data structure that allows you to quickly retrieve stored generation data.
grid	Displays 2-dimensional data for a client/server application in a table format. One-dimensional data shows one type of data, such as a phone number, name, or quantity. Two-dimensional data shows additional information in a grid or table. For example, the detail lines on an order can be displayed in a grid with each grid row corresponding to a unique line item. Each column in the grid corresponds to a discrete piece of information about the line, such as an item name, price, or quantity.
grid control	GUI control that displays related information in a table format. For example, purchase order line items can be displayed in a grid. The grid control supplied with Construct Spectrum sizes itself to the minimal width required to display all grid components. You can configure the grid control as desired.
group	Collection of users defined in the Construct Spectrum Administration subsystem.
GUI	Acronym for graphical user interface.
GUI control override	Use Predict keywords to force a GUI control derivation. See also keyword , page 262.
hook	Associate a command handler object with a command ID. See also command handler , page 257, command handler list , page 258, and command ID , page 258.
host	See server , page 266.
HTML fragment	Portion of HTML that is not a complete web page.
HTML template	HTML that may contain replacement tags, which are dynamically exchanged for content or nested HTML templates at runtime.
HTML Template wizard	Wizard used to generate HTML templates.

Term	Definition (continued)
http request	Parameterized list of named value pairs sent by a browser client to a web application.
instantiation	Process of creating an instance of a class. The result is an object.
internationalization	Adapting an application to make it easy to localize. See also localization , page 262.
job control language (JCL)	Command language used for batch jobs that tells the computer what to do.
keyword	Predict metadata type that acts as a label or identifier.
Level 1 data block	Level one field or structure and its subfields in a Natural parameter data area (PDA).
Level 1 data block optimization	Technique to improve the performance of client/server applications by reducing the volume of data transmitted across a network. Rather than sending all data blocks associated with an object, only the required blocks are sent.
library image file (LIF)	File that defines Natural definitions used by the Spectrum Dispatch Client.
LIF definitions module	BAS module in a Visual Basic project containing the definitions for application services, parameter data areas, and subprograms.
localization	Process of translating and adapting a software product for use in a different language or country.
lookups	Return descriptive information when a user requests a browse dialog or enters a value in a foreign key field on a maintenance dialog. For example, assume the Warehouse Number field is a foreign key field in the Order dialog and Warehouse Name is a descriptive field attached to the foreign key value. When a user enters a valid warehouse number, the lookup returns the name of the warehouse for display in the dialog.
maintenance dialog	GUI dialog from which a user can perform one or more actions on a business object. For example, a Customer Order object can be represented on a maintenance dialog. Using this dialog, an authorized user can add, delete, or update customer order information.

Term	Definition (continued)
MDI child	Window or dialog opened from an MDI parent window in a client/server application. For example, the Order maintenance dialog in the demo application is an MDI child to the MDI frame window.
MDI frame	Standard Visual Basic MDI frame supplied with the Construct Spectrum client framework.
MDI parent	MDI window from which other windows are opened and displayed in a client/server application. The MDI frame supplied with the client framework is an MDI parent.
menu	<p>On a mainframe server, a panel or window listing available functions. To access a function, users enter a value in an input field or move the cursor to a value and press Enter.</p> <p>In Windows, a pull-down dialog listing the available functions. To access a function, users select an option from the menu using the cursor or a keystroke combination.</p>
menu bar	Displays the menus available for user selection. By default, Construct Spectrum client/server applications contain File, Edit, Actions, Window, and Help menus on the menu bar, each containing standard menu commands.
metadata	Information about data. Metadata describes how physical data is formatted and interrelated. It includes descriptions of data elements, data files, and relationships between data entities. Typically, metadata is maintained in a repository known as a data dictionary, such as Predict.
method	Procedure that operates on an object and is implemented internally by the object. For example, the Update method updates a Customer Order object after changes to the order information.
model	Template used to generate modules. Each model contains one or more specification panels. Using these panels, you can specify parameters for a desired module and then generate the corresponding code. Natural Construct provides numerous models, including the Object-Maint-Subp and Subprogram-Proxy models.
module	Single application component, such as a hand-coded Natural program, subprogram, or data area or a Natural Construct-generated program, subprogram, data area, or subprogram proxy.

Term	Definition (continued)
multi-level security	Security you can define at a high level or at a detailed level affecting many objects. For example, you can apply multi-level security to domains, objects, and methods.
multiple-document interface (MDI)	Microsoft Windows paradigm for presenting windows whereby a parent window can encompass one or more child windows. See also MDI child , page 263, and MDI parent , page 263.
Natural Construct nucleus	Sophisticated driver program that invokes the model subprograms at the appropriate time in the generation process and performs functions common to all models, such as opening windows and performing PF-key functions. The nucleus communicates with the model subprograms through standard parameter data areas (PDAs). These PDAs contain fields assigned by Natural Construct, as well as fields required by a model.
Natural Debugging facility	Utility available in a Natural environment to help you locate and analyse logic errors. To access the facility, use the Invoke Proxy function in the Construct Spectrum Administration subsystem. The subprogram proxy sets up an online environment that simulates the client/server environment and allows you to use all the features of the Natural Debugging facility.
navigation bar	Menu bar on a web page containing links to other pages or actions.
node	Individual computer or, occasionally, another type of machine in a network.
nucleus	See Natural Construct nucleus , page 264.
object	Any application component, such as a form or record. A business object is a group of services related to a common business entity, such as Customer, Order, or Department.
object factory	Visual Basic module that identifies all objects and methods in an application and instantiates objects upon request.
Object Factory wizard	Visual Basic Add-In that updates an object factory in a Construct Spectrum web application.
object library	Provides definitions for all the objects, methods, and properties exposed by an OLE automation server. Equivalent to type library , page 268.

Term	Definition (continued)
OLE	Acronym for object linking and embedding.
OLE automation server	Code component that passes objects to other applications so they can programmatically manipulate the objects.
overflow condition	Situation where there are more fields than can be displayed on a dialog.
package	Collection of all modules necessary to implement a business object. A package combines components and classes to provide both browse and maintenance services for a database table. It is composed of a set of modules generated from a multi-module generation. An application is made up of one or more packages.
page handler	Visual Basic class that exchanges replacement tags on an HTML template with database content or another HTML template.
Page Handler wizard	Construct Spectrum Add-In that generates page handlers for web applications.
parent model	Super model that collects parameters for child models and generates specifications.
parse area	Code in a page handler that locates and exchanges HTML replacement tags.
ping	Request sent to a service to determine whether the service is running.
platform	Piece of equipment that, together with its operating system, serves as a base on which you can build other systems. For example, an MVS mainframe computer can serve as a platform for a large accounting system.
project	Collection of files used to build an application in Visual Basic.
project group	Collection of two or more Visual Basic projects, for example, web and ABO projects. A project group uses a .vbg extension.
property	Characteristic of an object, such as size, caption, or color. In Construct Spectrum, it refers to the data settings or attributes for an object in Visual Basic.

Term	Definition (continued)
regenerate/preserve status	Status indicating whether a code block in a module is regenerated or preserved during regeneration of the module. If you mark a block to be regenerated, it is replaced or deleted. If you mark a block to be preserved, it is not changed during regeneration.
remote call	Communication with an object residing in a different location, such as a server.
replacement tag	HTML tag that is replaced with database content or another HTML template when the web page is assembled. Some replacement tags can be used to remove existing sections of HTML. For example, you can use a security tag to specify content that only certain users can access.
resource	Text or binary value that can be localized. See also localization , page 262.
run	Execute or invoke a module or application.
security cache	File used to store recently-accessed security data.
server	Computer that provides services to another computer (called a client) and responds to requests for services. On multitasking machines, a process that provides services to another process is called a server.
server application	Application that runs on a server machine.
service	Software service that runs on a server. Several services can run on one server.
service exit	Exposed exit routine called by the Spectrum dispatch service; it can be replaced by a user-supplied routine.
service log	File used to store service log data.
shutdown	Command sent to a service to terminate the service.
software development kit (SDK)	See toolkit , page 268.
Spectrum client/server application	Application created using the Construct Spectrum wizards and add-ins. Users access mainframe business functions and data through a Visual Basic component running on a Windows platform.

Term	Definition (continued)
Spectrum Control record	Record that is created daily and contains system control and statistic data for a Spectrum dispatch service.
Spectrum Dispatch Client (SDC)	Provides the Construct Spectrum data exchange, which facilitates calls from a client to Natural subprograms running on a server.
Spectrum dispatch service	Middleware component that encapsulates broker calls on the server, provides directory services, enforces security, and invokes backend Natural services.
Spectrum security service	Component of the Construct Spectrum Administration subsystem that controls access to application libraries, objects, and methods.
Spectrum Service Manager	Client tool supplied with Construct Spectrum that allows you to specify which Spectrum services the client uses to communicate with the server.
Spectrum service settings	Collection of parameters used to configure a Spectrum service.
Spectrum web application	Application created using the Construct Spectrum wizards and add-ins. It allows users to access mainframe business functions and data from a web browser.
Spectrum web framework	Group of Visual Basic modules and classes that collaborate to dynamically generate web pages.
Status bar	Area that displays status information about a selected item, application, or business object in a client/server application. It contains sections for a message, status indicators, and the current date and time. Status bars are also displayed at the bottom of an MDI form.
steplib chain	Hierarchy of Natural libraries that determines the location from which modules are executed.
Sub Main procedure	First Visual Basic procedure executed when you run a Construct Spectrum application. Each Visual Basic application has one Sub Main procedure.
subprogram proxy	Natural subprogram called by a Spectrum dispatch service to translate data formats between the client and a Natural subprogram on the server. Each subprogram requires a subprogram proxy, which allows Construct Spectrum to provide a common interface to any subprogram.

Term	Definition (continued)
super model	Model that generates multiple components of a Construct Spectrum client/server or web application. Using a minimum number of input parameters, a super model determines the specifications for all models required to generate individual components of a package. See also package , page 265.
target module	See target subprogram , page 268.
target object	See target subprogram , page 268.
target subprogram	Any Natural subprogram.
template parser	Class used to parse HTML or other templates.
toolbar	Bar that provides quick access to commonly used commands in an application. A user clicks the appropriate toolbar button to perform the action it represents. Any action that can be performed from a toolbar can also be invoked from a menu.
toolbar button	Icon on a toolbar that allows users to perform an action.
toolkit	Set of related and reusable classes that provide general-purpose functionality. An application incorporates classes from one or more toolkits. Toolkits, or software development kits (SDKs), emphasize code reuse and are the object-oriented equivalent of subroutine libraries. For example, a toolkit can be a collection of classes for lists, associative tables, or stacks.
trace options	Options that specify how to trace messages sent between the client and server.
type library	Library containing definitions for all objects, methods, and properties exposed by the OLE automation server. See also object library , page 264.
upload data	Transfer modules from the client to the server.
variant	Visual Basic term identifying a late-binding data type. Variants allow Construct Spectrum subroutines or functions to accept different types of data. The exact type is determined when they receive the value in Visual Basic.
VB-Client-Server-Super-Model model	Model that generates all modules required for a fully functional client/server application. The super model can generate all modules required for maintenance and browse services for up to 12 business objects at a time. See also super model , page 268.

Term	Definition (continued)
verification rules	<p>Predict-defined business rules that are implemented in the object subprogram on the server and the maintenance object on the client. They also provide default values for derived fields represented by GUI controls, such as check boxes, option buttons, or drop-down combo boxes.</p> <p>You can use verification rules to force users to make a selection based on one or more choices. For example, if an application has an input field for the state name, you can attach a verification rule to the field in Predict so that only valid state names are accepted.</p>
Visual Basic browse object	<p>Visual Basic class that configures an instance of a browse base class. This class delivers information about the columns and keys supported by the browse subprogram to the client framework, which configures and displays the browse dialog at runtime. See also Visual Basic business object, page 269.</p>
Visual Basic business object	<p>Conceptual browse or maintenance object comprised of class modules or objects with a domain on the client. It implements business rules and encapsulates communication with the Spectrum Dispatch Client (SDC).</p>
Visual Basic maintenance object	<p>Visual Basic class instantiated by a maintenance dialog to:</p> <ul style="list-style-type: none"> • encapsulate calls to the SDC • implement validation in the maintenance dialog <p>See also Visual Basic business object above.</p>
Web Super wizard	<p>Construct Spectrum Add-In to Visual Basic that generates multiple HTML templates and page handlers for a web application.</p>
web class	<p>Visual Basic class that responds to requests for a web page (ASP requests).</p>
web application ASP	<p>ASP (active server page) script used to instantiate a Spectrum web application.</p>
wildcard	<p>Character or symbol that qualifies a selection, such as “*”, “<”, or “>”. For example, using a value followed by an asterisk (*) indicates a range of file names beginning with that value. To list all modules that begin with “Maint”, enter “Maint*” as the selection criteria.</p>

Term	Definition (continued)
XML extract	Extract information from Predict and other sources, which is stored on the client as metadata in XML format. This includes information about business objects, as well as the formatting used by wizards to build application components. See also metadata , page 263.

APPENDIX B: UTILITIES

This chapter describes the utility subprograms supplied with the Spectrum Administration subsystem. To invoke these subprograms, you must be in the SYSSPEC library.

The following topics are covered:

- **Response Subprogram**, page 272
- **Spectrum Interface Subprogram**, page 278
- **Conversation Factory Utility**, page 289
- **Character Translation Subprogram**, page 290
- **Multi-Tasking Verification Utility**, page 291
- **Log Utilities**, page 292

Response Subprogram

The SPUREPLY subprogram is mainly used by servers to send responses back to a client. The response can be defined as a SYSERR message or a hardcoded text string.

Features and Benefits

SPUREPLY has the following benefits and features:

- Defines a standard protocol for exchanging messages.
- Enables messages to be multilingual if you define them in SYSERR.
- Performs message substitution of :1::2::3: within SYSERR messages.
- Can send other information in addition to a message.

Response Length Limitation

The maximum supported response length is 5000 bytes.

Supported Methods

SPUREPLY supports the following methods (defined in SPLREPLY). One of these methods must be assigned to the SPAREPLY.METHOD parameter before calling SPUREPLY:

Method	Description
SEND-REPLY	Sends a single message reply, with the End of Conversation option.
SEND-WITHOUT-EOC	Sends a multi-part reply. Use the SEND-REPLY method to send the last message of the reply.
LOOKUP-MESSAGE	Looks up the error message text, but does not send it.
SEND-MESSAGE-ONLY	Sends the message text without the standard protocol information.
SEND-MESSAGE-ONLY-WITHOUT-EOC	Same as SEND-MESSAGE-ONLY, but does not include End of Conversation option.

Message Protocol

All messages sent to the client use the following protocol:

Message	Protocol
SIGNATURE(A6)	MSG111 constant. Defines the structure of the send buffer.
RESPONSE-CODE(N4)	Response code passed to SPUREPLY in SPAREPLY. Successful responses use a response code of zero. Other predefined response codes are: <ul style="list-style-type: none"> • 001 (Replica ID was not matched) • 9999 (Natural runtime error)
REPLICA-ID(A32)	Replica ID passed from SPAETB.
SPECTRUM-SERVICE(A32)	Passed from SPAETB.
SYSERR-LIBRARY(A8)	Name of the SYSERR library containing the message.
MSG-NR(N4)	Number of the SYSERR message.
MESSAGE(A1/1:V)	Message area of send buffer. In most cases, this area contains a message looked up in SYSERR by SPUREPLY. Additional information can also be passed in this area.

Call Interface

SPUREPLY supports the following interface:

```

PARAMETER USING SPAREPLY      /* Specific parameters
PARAMETER                      /* The message portion of the send buffer
01 SPAREPM
  02 INPUT-OUTPUTS
    03 BUFFER-LENGTH (I2)
    03 MSG-BUFFER (A1/1:V)
PARAMETER USING SPAETB        /* Parameters to SPUETB
PARAMETER USING ETBCB         /* Standard broker control block
PARAMETER USING CDPDA-M       /* Standard message area

```

These data areas are described in the following sections.

SPAREPLY Data Area

This data area is passed to SPUREPLY. It contains the following data:

Parameter	SPAREPLY	Library	S441	DBID	17	FNR	60
Command							> +
I T L Name				F	Leng	Index/Init/EM/Name/Comment	
Top	-----						
*							
*							
*	Data Area Name: SPAREPLY						Function
*	Created on....: Jun 12, 02						=====
*	Created by....: SAG						This data area is passed to
*							SPUREPLY which is used to
*							send a reply back to a client.
*							
*							The reply structure is
*							defined in SPLREP.
1	SPAREPLY						
2	INPUTS						
3	METHOD		I	1	/* See SPLREPLY		
3	RESPONSE-CODE		N	4	/* This response, use zero for		
*							/* successful response.
3	SYSERR-INFO						
4	MSG-NR		N	4	/* SYSERR Message number		
4	SYSERR-LIBRARY		A	8	/* Defaults to SYSSPEC		
4	MSG-DATA		A	32	(1:3) /* Subs. values		
*							/* May contain *NNNN references
3	TRANSLATE		L	/* Translate character set. If			
*							/* currently EBCDIC, message
*							/* will be translated to ASCII
*							/* and vise-versa.
3	EMBEDDED-MSG-INFO						/* This structure is only used
*							/* when the message to be looked
*							/* up is only a portion of the
*							/* data to be sent. In this case
*							/* you must indicate where the
*							/* message is in the send buffer
4	MSG-START		I	2	/* Byte location of start of msg		
4	MSG-LENGTH		I	2	/* Total length of message		
							/* portion

SPAREPLY Data Area

The fields in this data area are:

Field Name	Description
METHOD (I1)	Indicates whether you want to perform a send with EOC, send without EOC, or just look up the message text. Assign a value from SPLREPLY.
RESPONSE-CODE (N4)	Contains the response code value sent to the client.
MSG-NR (N4)	If a message is looked up in SYSERR, contains the message number.

Field Name	Description (continued)
SYSERR-LIBRARY (A8)	Name of the library in which to look up messages. By default, all messages are looked up in the SYSSPEC library. If this is not true, specify the library name.
MSG-DATA (A32/1:3)	Contains up to three values for substitution into the message. These values replace the :1::2::3: placeholders in the SYSERR message. Substitution values can be looked up in SYSERR by specifying message data in *nnnn format.
TRANSLATE (L)	Indicates whether the message is translated (from EBCDIC to ASCII or vice versa).
MSG-START (I2)	If the message retrieved from SYSERR represents only a portion of the data to be sent, indicates the starting position of the message portion of the send buffer.
MSG-LENGTH (I2)	Indicates the length of the message portion of the send buffer. This field is only required when MSG-START is assigned.

SPAREPM Data Area

This data area is an example of a standard message area that can be passed to SPUREPLY. Use SPAREPM to send messages up to 250 characters in length. After SYSERR messages are looked up, the resulting message text is returned in this parameter. The values in SPAREPLY.MSG-START and SPAREPLY.MSG-LENGTH determine where the message is assigned. If these values are zero, the message is returned, starting at position 1 and continuing to SPAREPM.BUFFER-LENGTH.

The SPAREPM data area contains the following fields:

```

cal      SPAREPM   Library S441                      DBID    17 FNR    60
Command
I T L Name                                     F Leng Index/Init/EM/Name/Comment
All -----
*   Data Area Name: SPAREPM                      Function
*   Created on....: Jun 12, 02                    =====
*   Created by....: SAG                          This data area can be used as
*                                                    the second parameter to
*   SPUREPLY. When a message number
*   is passed to SPUREPLY, the
*   message text is returned in
*   this parameter.
*   Alternatively, the message to
*   be sent can be passed to
*   SPUREPLY using this parameter.
1 SPAREPM
2 INPUT-OUTPUTS
3 BUFFER-LENGTH          I    2 INIT<250>
3 MSG-BUFFER             A    1 (1:250)
R 3 MSG-BUFFER
4 MSG-STRING             A    250
----- Current Source Size: 1201  Free: 100104 ----- S 17  L 1

```

SPAREPM Data Area

To send information other than a standard message, copy SPAREPM and define the fields you want to send (up to 5000 bytes). To reflect the size of data to be sent, assign the BUFFER-LENGTH field.

Example of a call

```

/*
/* SYSSPEC/1001: Invalid request:1:sent to:2:expecting:3:
ASSIGN SPAREPLY.MSG-NR = 1001
ASSIGN SPAREPLY.MSG-DATA(1) = #COMMAND
ASSIGN SPAREPLY.MSG-DATA(2) = *PROGRAM
ASSIGN SPAREPLY.MSG-DATA(3) = ''CREATE''
ASSIGN SPAREPLY.RESPONSE-CODE = 1 /* Invalid command
PERFORM SEND-MESSAGE
*
*****
DEFINE SUBROUTINE SEND-MESSAGE
*****
*
  IF #I-AM-ASCII NE #CLIENT-IS-ASCII THEN
    ASSIGN SPAREPLY.TRANSLATE = TRUE
  END-IF
  ASSIGN SPAREPLY.METHOD = SPLREPLY.SEND-REPLY /* Send with eoc
  CALLNAT 'SPUREPLY' SPAREPLY
              SPAREPM
              SPAETB
              ETBCB
              MSG-INFO
END-SUBROUTINE /* SEND-MESSAGE

```

Example of send buffer

```

MSG1110001ATTACH-MANAGER--B0B218EC55E1AE01          AURORA-CONVERSATION FACTORY
SYSSPEC 1001Invalid request CMD SH sent to SPSCFACT expecting 'CREATE'

```

where:

MSG111	Is the message signature.
0001	Is the response code.
ATTACH-MANAGER-- B0B218EC55E1AE01	Is the server replica ID.
AURORA-CONVERSATION-FACTORY	Is the Spectrum service.
SYSSPEC	Is the name of the SYSERR library used.
1001	Is the SYSERR message number.
Invalid request CMD SH sent to SPSCFACT expecting 'CREATE'	Is the message text.

Spectrum Interface Subprogram

Writing robust servers can be a complex task. There are many possible errors that can occur, and ensuring that each error is handled in the proper way is very difficult. Some errors are caused by resource shortages, so it is desirable to retry the call again after a brief pause. Other errors are fatal and should result in the server shutting down. Still other errors, like wait timeouts, are normal and expected.

To help simplify and standardize the task of writing servers, Construct Spectrum supplies a subprogram that wraps the Broker ACI calls. This wrapper subprogram, called SPUETB, handles many situations that have to be coded to make direct Broker calls. To ensure that errors are handled and logged properly, use SPUETB for all broker calls.

Features and Benefits

The following sections contain a summary of the capabilities offered by SPUETB.

Broker Error Handling

Most Broker errors are handled internally by SPUETB. If the errors are due to resource shortages, SPUETB pauses for two seconds and then tries the call again. The subprogram continues to retry the call for up to 20 seconds.

When implementing server receive loops, SPUETB handles all wait timeouts (Broker error 74) and returns to the receive state.

Fatal errors cause the server to shutdown if SPUETB is granted shutdown permission. SPUETB can also handle message length errors and return a message to the sender indicating that the message was too long.

Error Logging

All errors returned from Entire Broker are logged in the Spectrum Communication Log. Use this log to help detect problems with your programs or environment.

Shutdown Requests

SPUETB responds to shutdown requests from Entire Broker. These requests can be initiated using the EntireX Broker Control Center.

Server Timeouts

Whenever the server has not received a message for the length of time specified on the service record, the server shuts down.

Command Handling

SPUETB registers for the CMD service and responds to all command requests. Command requests include the CMD CALLNAT command, which allows you to supply the name of the subprogram call.

SPUETB Interface

SPUETB is called using the following interface:

```
DEFINE DATA
  PARAMETER USING SPAETB      /* Specific Parameters
  PARAMETER USING ETBCB      /* Standard broker control block
  PARAMETER
  01 SEND-BUFFER(A1/1:V)
  01 RECEIVE-BUFFER(A1/1:V)
  01 RESERVED-AREA(A1/1:V) /* Reserved for SPUETB use
  PARAMETER USING CDPDA-M    /* Standard message area
END-DEFINE
```

As in a direct call to Entire Broker, the caller is responsible for filling in the Broker control block. Additionally, the caller can specify the degree of error handling and support for common functions handled by SPUETB.

The data areas are described in the following sections.

Data Areas

SPAETB Data Area

```

Parameter SPAETB      Library S441
Command
I T L Name           F Leng Index/Init/EM/Name/Comment   > +
Top -----
*
*
*   Data Area Name: SPAETB                               Function
*   Created on....: May 05, 02                           =====
*   Created by....: SAG                                   This data area is passed to
*                                                         SPUETB which is used to
*                                                         encapsulate calls to Entire
*                                                         Broker. Use SPLETB to assign
*                                                         constant values.
*
*   1 SPAETB
*   2 FORCE-PDA                                           A   1 (1:V) /* This field is only here
*                                                         /* to force the caller to create
*                                                         /* a separate LDA to call SPUETB
*                                                         /* rather than using SPAETB.
*                                                         /* This way, initial values can
*                                                         /* be placed in the LDA so that
*                                                         /* defaults get assigned.
2 INPUTS
*   3 METHOD                                               I   1 /* 0 = Normal call
*                                                         /* See SPLETB for other methods
*   3 ENCAPSULATED-FUNCTIONS                             L   /* Set desired functions ...
*   4 SUPPORT-SERVER-COMMANDS                           L   /* SPUETB will automatically
*                                                         /* register a command service
*                                                         /* whenever a regular service is
*                                                         /* registered. CMD is used as
*                                                         /* the broker service name.
*                                                         /* SPUETB will handle all
*                                                         /* command requests directly.
*   4 ALTER-RECEIVE-SERVICE                             L   /* Automatically change the
*                                                         /* service name on receive to
*                                                         /* an '*' to allow commands
*   4 SHUTDOWN-PERMISSION                               L   /* If true, SPUETB is allowed to
*                                                         /* shutdown the server directly.
*                                                         /* See SHUT-DOWN-REASONS
*   4 SHUTDOWN-REASONS                                  L   /* Set desired shutdown reasons:
*                                                         /*      only set after method 6
*   5 EXPLICIT-SHUTDOWN                                  L   /* Shutdown request from BROKER
*                                                         /* or from Spectrum console.
*   5 TIMEOUT-REACHED                                   L   /* See TIMEOUT-HANDLING
*   5 TERMINAL-ERROR                                    L   /* Non-recoverable broker error.

```

```

4 TIMEOUT-HANDLING          I  4 /* 0 = Return all timeouts so
*                          /* that caller can handle
*                          /* >0= Reissue call for this
*                          /* many seconds. Set to
*                          /* max desired idle period.
*                          /* -1= Reissue call indefinitely
*                          /* -1 is normally used by
*                          /* ATTACH servers which
*                          /* should run forever.

3 ERROR-HANDLING
4 HANDLE-TRUNCATION-ERROR  L  /* SPUETB will respond to
*                          /* ETB error 00200094. This
*                          /* won't be sent back to caller

4 RESERVED                 A  8 /* Reserved for future.
4 USE-SPECTRUM-ERROR-LOG   L  /* Log all errors on the Spec.
*                          /* file. Warning, this will
*                          /* cause an ET to be issued.
4 WRITE-ERRORS-TO-CONSOLE L  /* CALL 'CMWTO' with errors
4 WRITE-ERRORS-TO-PRINT-FILE-0 L /* Write errors to Natural
*                          /* print file 0
4 MAX-RETRY-TIME          I  2 /* Number of seconds to continue
*                          /* to retry call in the event of
*                          /* a Broker resource shortage.
*                          /* Defaults to 20 seconds.
4 MESSAGE-DATA            *                          /* These fields are used to
*                          /* build helpful error messages
*                          /* when broker calls fail.

5 CALLING-PROGRAM          A  8 /* Name of caller.
5 SPECTRUM-SERVICE        A 32 /* Name of spectrum service
*                          /* if known.
5 CALL-DESC               A 32 /* Description of the call
2 INPUT-OUTPUTS
3 REPLICAS-ID            A 32 /* Assigned at first LOGON
*                          /* do not adjust
3 CLIENT-MODE             L  /* In this mode, errors need
*                          /* not be logged and checks
*                          /* for broker error cycles
*                          /* are not performed.
3 OPTION                  A 50 /* SPUETB option
2 OUTPUTS
3 RESULT                  I  1 /* See SPUETB
*                          /* 0 = Normal request
*                          /* 1 = Attach request
*                          /* 2 = Command request
*                          /* 3 = Timeout
*                          /* 4 = Non-terminal error
*                          /* 5 = Terminal error
*                          /* 6 = Restarting after error.

```

SPAETB Data Area

The fields in the SPAETB data area are:

Field Name	Description
FORCE-PDA (A1/1:V)	Due to the number of input settings that must be assigned before calling SPUETB, the preferred method of assigning them is to use supplied LDAs, initialized with common defaults settings. The following LDAs are supplied: <ul style="list-style-type: none"> • SPAETBC (used by Broker client programs) • SPAETBS (used by Broker server programs)
METHOD (I1)	Determines the type of processing performed by SPUETB. Assign the method values using one of the constants in SPLETB.METHODS.
SPLETB.NORMAL-CALL	Used for all broker calls except LOGON, REGISTER, DEREGISTER, and LOGOFF.
SPLETB.LOGON	Uses the value of SPAETB.SPECTRUM-SERVICE to look up the Broker ID, user ID, and corresponding password with which to log on to Entire Broker. It also executes the Broker Logon function.
SPLETB.REGISTER-SERVER	Uses SPAETB.SPECTRUM-SERVICE to look up the Broker ID, Server Class, Server Name, and Service and uses these values to Register with Entire Broker. If the SUPPORT-SERVER-COMMANDS parameter is set to TRUE, this method also registers an additional service, CMD, to accept commands.
SPLETB.SHUTDOWN-SERVER	Invokes the Broker Deregister and Logoff functions. It is used by servers only. Always issue a shutdown request before ending server programs. Assign SHUTDOWN-PERMISSION=TRUE if you want SPUETB to perform a shutdown automatically.
SPLETB.LOG-SUPPLIED-ERROR	Requests that an application error be logged by SPUETB. The error must be passed in the MSG-INFO.##MSG field of the CDPDA-M data area. The message is logged to locations specified in the SPAETB.ERROR-HANDLING structure.
SPLETB.LOG-Natural-ERROR	Only called from ON ERROR blocks or error transactions (assign to *ERROR-TA). Tells SPUETB to log the last Natural error that occurred. The error is logged to locations specified in the SPAETB.ERROR-HANDLING structure.

Field Name	Description (continued)
SPLETB.GET-SERVICE-DEFAULTS	<p>Assigns the following fields based on the values established at the time of the initial LOGON method.</p> <ul style="list-style-type: none"> • ETBCB.BROKER-ID • ETBCB.SERVER-CLASS • ETBCB.SERVER-NAME ETBCB.SERVICE, • ETBCB.USER-ID • ETBCB.TOKEN • ETBCB.SECURITY-TOKEN • SPAETB.TIMEOUT-HANDLING • SPAETB.SPECTRUM-SERVICE
SPLETB.LOGOFF	<p>Performs a Broker Logoff function. For other methods, refer to SPLETB.</p>
SUPPORT-SERVER-COMMANDS (L)	<p>Tells SPUETB to automatically support command services such as PING, SHUTDOWN, etc. SPUETB automatically registers a separate service using CMD as the service name. All command requests are handled by SPUETB; the caller need not code any specific support for commands.</p>
ALTER-RECEIVE-SERVICE (L)	<p>Used in conjunction with SUPPORT-SERVER-COMMANDS. If this field is set to true, SPUETB automatically changes the service name specified on any receive function to an asterisk (*). This allows the receive to be satisfied by either a request for the main service or a request for the command service.</p>
SHUTDOWN-PERMISSION (L)	<p>If true, SPUETB can shutdown the current program. Normally, it is only set for server programs. To determine which events allow SPUETB to shutdown the running server, assign the fields in SPAETB.SHUTDOWN. SPUETB always logs any errors prior to shutting down.</p>
EXPLICIT-SHUTDOWN (L)	<p>Allows SPUETB to shutdown the server as a result of an explicit SHUTDOWN command.</p>
TIMEOUT-REACHED (L)	<p>Allows SPUETB to shutdown the server when the server timeout value is reached. This timeout value is passed in the TIMEOUT-HANDLING parameter and defaulted from the Server Timeout field on the Spectrum service record.</p>
TERMINAL-ERROR (L)	<p>Allows SPUETB to shutdown the server in response to a fatal Broker error.</p>

Field Name	Description (continued)
TIMEOUT-HANDLING (I4)	<p>Tells SPUETB how to handle timeouts when executing Broker RECEIVE functions. Can be one of the following:</p> <ul style="list-style-type: none"> • -1 Execute forever (use SPLETB.NO-TIME-LIMIT) to assign this value. • 0 Return to the caller after the first receive timeout. • >0 Execute for this many seconds, then either return to the caller or execute shutdown processing (based on SHUTDOWN-PERMISSION and TIMEOUT-REACHED parameters). <p>This field is derived from the Server Timeout value on the Spectrum service record. If no server timeout is specified, the following defaults are used:</p> <ul style="list-style-type: none"> • Services without Attach Servers -1 • Services with Attach Servers 1200 (= 20 minutes)
RESERVED (A8)	Reserved for future use.
USE-SPECTRUM-ERROR-LOG (L)	Logs all errors to the Spectrum log file.
WRITE-ERRORS-TO-CONSOLE (L)	Writes all errors to the operator console.
WRITE-ERRORS-TO-PRINT-FILE 0(L)	Writes all errors to Print file 0.
MAX-RETRY-TIME (I2)	Indicates the length of time to continue trying to execute a Broker call in the event of a Broker resource shortage. This defaults to 20 seconds.
CALLING-PROGRAM (A8)	Identifies the caller of SPUETB. This name is used when logging error messages.
SPECTRUM-SERVICE (A32)	To use the Logon and Register methods of SPUETB, specify the name of the Spectrum service in this field. Also used when writing error messages.
CALL-DESC (A32)	Free-format description of the call used when logging error messages.
REPLICA-ID(A32)	Replica id assigned to the server (output field only).

Field Name	Description (continued)
CLIENT-MODE(L)	If this flag is set, SPUETB does not log errors and checks for broker error cycles are not performed.
RESULT (I1)	<p>Interpreted after the call to determine the results of the call. The SPLETB data area defines the following constants to check the results:</p> <ul style="list-style-type: none"> • NORMAL-REQUEST Broker call completed normally. • ATTACH-REQUEST Broker call resulted in an Attach request. Only returned to Attach Services. • TIMEOUT Receive timeout was reached and shutdown permission for timeouts was not granted to SPUETB. • NON-TERMINAL-ERROR Non-terminal broker error occurred. This error is automatically logged by SPUETB. See ETBCB.ERROR-CODE. • TERMINAL-ERROR Terminal Broker error occurred, but shutdown permission was not granted to SPUETB. The error is automatically logged.

ETBCB Data Area

ETBCB is a standard data area representing the fields that must be passed to Entire Broker when using the Broker ACI. The calling program should use ETBCB12 or ETBCB13, depending on the version of the Broker stub in use.

SEND-BUFFER

The send buffer is used in conjunction with the Broker Send function. The size of this buffer must be greater than or equal to the value of ETBCB.SEND-LEN.

RECEIVE-BUFFER

The receive buffer is used in conjunction with the Broker Receive function or blocked Sends. The size of this buffer must be greater than or equal to the value of ETBCB.RECEIVE-LEN.

RESERVED-AREA

This pass area is reserved for future use. Define and pass the SPAETBP.NOT-USED(*) parameter in place of this parameter.

CDPDA-M

This is a standard message area. Whenever SPUETB encounters a non-recoverable error, it returns with the error text in MSG-INFO.##MSG and MSG-INFO.##RETURN-CODE is assigned "E".

Using SPUETB

For an example of using SPUETB, refer to the SPSTIMS Timestamp Server example. If you need to do your own character set translation (because your messages contain a mixture of printable and binary data), refer to SPSTIMS2.

CMD TRACE

The TRACE command enables and disables tracing of a running server. This feature is used in conjunction with the CSUDEBI utility. The TRACE command accepts a RID to target the command to a specific replica.

There are two separate forms of the TRACE command; the one you choose depends on whether you want to enable or disable tracing.

- To enable tracing:
 - 1 Use the CMD TRACE LOCATION=n [options] command.
- To disable tracing:
 - 1 Use the CMD TRACE OFF command.

Valid Keywords

Valid trace locations are defined in the CSLDEBUG local data area in SYSCST. The following table shows the trace keywords:

Keyword	Description
QHANDLE	A valid queue handle is required when setting the message location to 10. This is a quoted value consisting: 'bkrid, user-ID, token, (unpacked) security-token, conv-ID'
ERROR-TRIGGER	Forces a runtime error at a specified point within the running server. Errors can only be triggered on lines that are currently being traced. The syntax of the value assigned to this field is: Program,Line,NATnnnn,Skip' where: <ul style="list-style-type: none"> • Program is the name of the program where the runtime error is to be triggered. • Line is the line number where the error is to be triggered. • NATnnnn is the error to be triggered. • Skip is used if the error is not to be triggered on the next execution of the statement, but rather after executing the statement this many times.
FILTER-MASK	A100 string of 0 and 1 values. "1" is used to represent statements that are to be traced. Each mask character is related to a constant in the SPLTRACE local data area.
FILTER-PROGRAM	List of up to five programs (in quotes and separated by commas) used to limit the programs that produce trace output. You can use special characters in the program name to serve as pattern-matching characters. For details, refer to the PATTERN option for the Natural EXAMINE statement.

Example of enabling tracing

```
CMD TRACE
RID=BBCB0B5A1BD5AF9F201FACB0B5A14D5AF9F201, LOCATION=10, QHANDLE='BKR045
,SPSCFACT,AAC
B0B5A1BD5AF9F201FACB0B5A14D5AF9F201,00000000000000000000000000000000
00000000000000000000000000000000,0000000000000220',ERROR-
TRIGGER='SPUETB,5420,NAT0082',FILTER-MASK=1000110000000010000
0001000000011000000000000000000000000000000000000000000000000000, FILTER-
PROGRAM='SPU*,SP?SEC'
```

Trace Response

The trace response is normally a confirmation message indicating whether the trace request was successful. The response uses the SPUREPLY protocol (MSG111).

Test the Trace Facility

To test the trace functions, use the CMD CALLNAT SPUTRTST command.

CMD CALLNAT

It is possible to CALLNAT any subprogram, provided the subprogram implements a generic interface. This interface is defined as follows:

```
DEFINE DATA
  PARAMETER USING SPACALLN /* Standard callnat parameters
  PARAMETER USING SPAREPLY /* Reply message parameters
  PARAMETER
    01 RECEIVE-SEND-BUFFER(A1/1:15000)
END-DEFINE
```

The CALLNAT command takes the form:

```
CMD CALLNAT subpname parameter_string
```

where:

Subpname	Is the name of the subprogram you want to CALLNAT.
parameter_string	Is any set of characters to be passed to the specified subprogram using the RECEIVE-SEND-BUFFER.

For an example of how to write a new CALLNAT interface subprogram, refer to the SPUCMDT subprogram.

Conversation Factory Utility

Construct Spectrum includes a facility called a Conversation Factory. This facility works in conjunction with high-level callnat and message queue APIs to facilitate the simple transfer of data between two platforms. The benefits offered by the Conversation Factory and supporting APIs include:

- Allow communication between a client and server without knowledge of Broker ACI.
- Allow a conversation to be established between two processes, each acting as clients.
- Support multiple concurrent conversations between the same two participants. For example, the Construct generate server listens for specifications on one conversation and cancels requests on another.
- Are used in conjunction with servers launched from the client to establish a conversation between the client who launched a service and the service itself.

On the server, the Conversation Factory consists of the following four subprograms:

Subprogram	Description
SQUOPEN	Opens a new conversation.
SQUSEND	Sends information from one end of the conversation to the other.
SQURECV	Receives information.
SQUCLOSE	Closes the conversation.

For an example of how to use the Conversation Factory APIs, refer to the SQEXAMPL subprogram.

Character Translation Subprogram

When writing your own servers, it is sometimes necessary to perform character-set translation. The preferred approach to character translation is to use the translation routines assigned to the Broker Service in the Broker Attribute File. However, sometimes you may want to send a message that contains a mixture of binary and printable data where only a portion of the message is to be translated. Use the SPUTLATE subprogram for this purpose.

SPUTLATE allows you to pass in a string, along with an array of character positions to be translated. It is supplied in source form. For an example of calling SPUTLATE, refer to SPSTIMS2.

Determine a Character Set

Sometimes a server receives a message it cannot interpret. Normally, the server returns a reply to the sender indicating that the message is invalid. If the server performs its own translation, it needs to know the character set of the received message so that the reply can be sent back in the client's character set. SPUASCII helps determine whether a string of characters is ASCII or EBCDIC format. For an example of calling SPUASCII, refer to SPSTIMS2.

Multi-Tasking Verification Utility

Use this utility to verify that ADALNK has been configured to be re-entrant and that the Natural batch nucleus that uses it is also re-entrant. A re-entrant Natural nucleus is required to run Spectrum services in a batch multi-tasking environment.

To start multiple Natural subtasks, use JCL to run the supplied Natural module, TESTTASK, in batch (as documented in **Step 2: Verify Natural Subtask Support**, page 39, *Construct Spectrum and SDK Installation Guide for Mainframes*). If your Natural nucleus is re-entrant, TESTTASK will successfully start Natural subtask sessions that will execute the TESTSTSK program, which will then write trace information to workfile 1 showing the execution status of the subtasks. Otherwise, the job that runs TESTTASK will not end and will have to be manually cancelled.

Log Utilities

Construct Spectrum supplies several utilities for archiving and deleting log data. Most of the parameters apply to all log archive utilities.

Spectrum Log Utilities

The following Spectrum log utilities are supplied with Construct Spectrum:

Utility	Description
BSBLARCP	<p>Allows the Spectrum Log data to be archived to a work file and optionally deleted from the Spectrum Log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Input End Date	Indicates the last LOG date to be archived.
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).
Delete After Archive	Indicates whether to delete log records after they are archived.
BSBLRESP	<p>Restores data to the Spectrum Log file. It uses the entire log data created by the BSBLARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p>
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).

Construct Spectrum Control Record Log Utilities

The following Control record utilities are supplied with Construct Spectrum:

Utility	Description
BSCTARCP	<p>Allows the Spectrum Control Record log data to be archived to a work file and, optionally, deleted from the Spectrum Control Record log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Input End Date	Indicates the last LOG date to be archived.
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).
Delete After Archive	Indicates whether to delete log records after they are archived.
BSCTRESP	<p>Restores data to the Spectrum Control Record log file. It uses the entire log data created by the BSCTARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p>
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).

Domain Log Utilities

The following domain log utilities are supplied with Construct Spectrum:

Utility	Description
BSDOARCP	<p>Allows the Domain log data to be archived to a work file and, optionally, deleted from the Domain log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Input End Date	Indicates the last LOG date to be archived.
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).
Delete After Archive	Indicates whether to delete log records after they are archived.
BSDORESP	<p>Restores data to the Domain log file. It uses the entire log data created by the BSDOARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p>
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).

Spectrum Group Log Utilities

The following group log utilities are supplied with Construct Spectrum:

Utility	Description
BSGRARCP	<p>Allows the Spectrum Group log data to be archived to a work file and, optionally, deleted from the Spectrum Group log, based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Input End Date	Indicates the last LOG date to be archived.
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).
Delete After Archive	Indicates whether to delete log records after they are archived.
BSGRRESP	<p>Restores data to the Spectrum Group log file. It uses the entire log data created by the BSGRARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p>
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> • To display full details of all data being logged, enter “F” (full). • To show only the main log information, enter “B” (brief).

Application Service Definition Log Utilities

The following Application Service Definition utilities are supplied with Construct Spectrum:

Utility	Description
BSIFARCP	<p>Allows the Application Service Definition log data to be archived to a work file and, optionally, deleted from the Application Service Definition log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Report type	<p>Indicates which report type to display.</p> <ul style="list-style-type: none"> To include information related to the interface and method data, enter “F” (full). To display only the log for the application service header information, enter “B” (brief).
Delete After Archive	<p>Indicates whether to delete log records after they are archived.</p>
BSIFRESP	<p>Restores data to the Application Service Definition log file. It uses the entire log data created by the BSIFARCP utility. It also generates a log record of the restoration process.</p> <p>This utility has the following input field:</p>
Report type	<p>Indicates which report type to display.</p> <ul style="list-style-type: none"> To include information related to the interface and method data, enter “F” (full). To display only the log for the application service header information, enter “B” (brief).

Spectrum Steplib Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description
BSSDARCP	<p>Allows the Spectrum Steplib log data to be archived to a work file and, optionally, deleted from the Spectrum Steplib log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Input End Date	Indicates the last LOG date to be archived.
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> To display full details of all data being logged, enter “F” (full). To show only the main log information, enter “B” (brief).
Delete After Archive	Indicates whether to delete log records after they are archived.
BSSDRESP	<p>Restores data to the Broker Steplib log file. It uses the entire log data created by the BSSDARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p>
Full Report	<p>Indicates which details to display.</p> <ul style="list-style-type: none"> To display full details of all data being logged, enter “F” (full). To show only the main log information, enter “B” (brief).

User and Group Log Utilities

The following utilities are supplied with Construct Spectrum:

Utility	Description
BSUSARCP	<p>Allows the User and Group log data to be archived to a work file and, optionally, deleted from the User and Group log based on a date. It also generates a log record of the archive process.</p> <p>This utility has the following input fields:</p>
Report type	<p>Indicates which report type to display.</p> <ul style="list-style-type: none"> To include information related to the interface and method data, enter “F” (full). To display only the log for the application service header information, enter “B” (brief).
Delete After Archive	<p>Indicates whether to delete log records after they are archived.</p>
BSUSRESP	<p>Restores data to the User and Group log file. It uses the entire log data created by the BSUSARCP utility. It also generates a log record of the restore process.</p> <p>This utility has the following input field:</p>
Report type	<p>Indicates which report type to display.</p> <ul style="list-style-type: none"> To include information related to the interface and method data, enter “F” (full). To display only the log for the application service header information, enter “B” (brief).

INDEX

Numerics

- 1:V fields
 - example of a PDA, 233
 - example of calling the Allocate method, 233
 - example of code to specify an array, 233
 - example of code to specify number of occurrences, 233
 - example of instantiating the PDA, 233
 - example of obtaining the bounds of an array, 234
 - example of passing an array, 234
 - example of using the same data area, 234
- 1:V overrides
 - Edit 1
 - V Overrides window, 108
- 1:V variables
 - Subprogram proxies
 - 1:V variable considerations, 108

A

- ABO interface
 - customizing, 98
- ABO project
 - components, 91
 - creating, 87
- ABO wizard
 - using, 92
- Active server page (ASP) script
 - definition of, 255
- ActiveX business object (ABO)
 - definition of, 255
 - with Microsoft IIS, 35
- ActiveX DLL
 - definition of, 255
- Adding
 - methods
 - application service definitions, 113
 - user exits
 - Subprogram-Proxy model, 106, 109
- Altered characters
 - Translations program, 181
- Application library
 - definition of, 255
- Application objects
 - troubleshooting, 185
- Application service definition
 - definition of, 255
- Application service definitions
 - accessing, 112
 - adding a method, 113
 - Maintain Application Service Definitions panel, 112
 - methods, 111
- Application services
 - creating and selecting server components, 236
 - definition of, 255
 - external parameters, 237
 - global data areas, 236
 - parameter data areas, 237
 - subprogram behavior, 237
 - subprogram interface, 236
 - terminal I/O, 236
 - timing issues, 237
- Applications
 - client/server
 - deploying, 191
 - Construct Spectrum
 - client/server, 28
 - web, 28
 - creating without using client framework
 - setting up server components, 236

Architecture

- Construct Spectrum diagram, 29
- definition of, 255

- ASCII character set translation of, 31

B

BDT class

- definition of, 256

BDT controller

- calling conversion routines, 126
- convert from display, 128
- convert to display, 127
- converting in-place, 128
- creating sample values, 129
- error information properties, 131
 - ErrorCode, 131
 - ErrorLen, 131
 - ErrorMsg, 131
 - ErrorPos, 131
- example of code, 131
- syntax example, 125
- using Natural formats, 130

BDT controller class

- definition of, 256

BDT controller object

- definition of, 256

BDT modifier

- definition of, 256

BDT procedure

- definition of, 256

BDT_DATE

- Predict keyword, 137

BDTs (business data types)

- overview, 122

Block handling

- default methods, 116
- overriding, 116
- specifying on the server, 118

Broker ACI calls

- wrapping, 278

Browse command handler

- definition of, 255

Browse data cache

- definition of, 255

Browse dialog

- definition of, 256

Browse process

- definition of, 256

Browse subprogram proxy

- number of occurrences returned, 108
- specifying number of occurrences, 106

BSBLARCP

- log utility, 292

BSBLRESP

- log utility, 292

BSCTARCP

- log utility
 - Control record, 293

BSCTRESP

- log utility
 - Control record, 293

BSDOARCP

- log utility
 - domain, 294

BSDORESP

- log utility
 - domain, 294

BSGRARCP

- log utility
 - group, 295

BSGRRESP

- log utility
 - group, 295

BSIFARCP

- log utility
 - Application Service Definition, 296

BSIFRESP

- log utility
 - Application Service Definition, 296

BSSDARCP

- log utility
 - steplib, 297

BSSDRESP

- log utility
 - steplib, 297

BSUSARCP

- log utility
 - user and group, 298

- BSUSRESP
 - log utility
 - user and group, 298
 - Business data type (BDT)
 - definition of, 256
 - Business data type (BDT) objects
 - BDTController, 141
 - BDTConversion, 141
 - diagram of properties and methods, 142
 - Business data types
 - setting up in Predict, 40
 - Business data types (BDTs)
 - benefits of using, 123
 - client framework, 124–125
 - composition
 - conversion routine, 124
 - modifiers, 124
 - name, 124
 - creating
 - modifiers, 138
 - name, 138
 - Natural formats supported, 138
 - returning appropriate variant types, 139
 - customizing and creating, 138
 - handling runtime errors, 141
 - one conversion routine with multiple BDTs, 148
 - overriding, 149
 - overview, 122
 - placing the conversion routine, 148
 - referencing
 - in your application, 149
 - Predict, 150
 - registering, 145
 - example of code, 145
 - retrieving error information, 149
 - supplied with Construct Spectrum
 - Alpha, 133
 - Boolean, 133
 - Currency, 135
 - Date, 136
 - Numeric, 134
 - Time, 134
 - used by client framework
 - diagram, 126
 - using modifiers, 129
 - Visual Basic, 123
 - diagram, 123
 - Business object
 - definition of, 257
 - setting up in Predict, 42
 - Business-Object-Super-Model
 - before using, 73
 - application environment, 75
 - default values in Predict, 73
 - model defaults, 73
 - naming conventions, 74
 - generating packages, 76
 - general package parameters, 78
 - specific package parameters, 79
 - standard parameters, 77
 - overview, 72
 - when to use, 72
 - Business-Object-Super-Model wizard
 - definition of, 257
- ## C
- CallNat method
 - syntax, 220
 - CALLNAT simulation
 - Spectrum Dispatch Client, 33
 - CallSystem method
 - syntax, 221
 - Cardinality
 - definition of, 257
 - Character set
 - determining, 290
 - Character UI
 - on mainframe server, 30
 - Checklists
 - application creation, 38
 - Child model
 - definition of, 257
 - Client applications
 - definition of, 257
 - developing
 - code to call subprogram, 252
 - creating the user interface, 251
 - running applications, 253
 - Client calls
 - simulate for debugging, 175
 - Client framework
 - definition of, 257

- Client/Server application
 - creating using Construct Spectrum tools, 17
 - creating without Construct Spectrum, 17
 - Code
 - preserving customizations to generated code, 58
 - protecting
 - using implied user exits, 58
 - using the cst
 - PRESERVE tag, 58
 - Code block
 - definition of, 257
 - Command block
 - definition of, 257
 - Command handler list
 - definition of, 258
 - Command handlers
 - definition of, 257
 - Command ID
 - definition of, 258
 - Communication errors
 - handling, 162
 - possible origins, 162
 - retrieving information, 163
 - severe, 163
 - Complex redefine
 - definition of, 258
 - Components
 - framework
 - ABO project, 91
 - Compression
 - definition of, 258
 - Configuration editor
 - invoking, 50
 - Configuration Profiles tab
 - Configuration editor, 50
 - Construct Spectrum
 - courses, 20
 - definition of, 258
 - glossary of terms, 255
 - Construct Spectrum Add-In
 - definition of, 258
 - Construct Spectrum Administration subsystem
 - definition of, 258
 - Construct Spectrum applications
 - creating without using client framework
 - setting up server components, 236
 - Construct Spectrum SDK Reference
 - layout, 14
 - Conventions
 - used in this documentation, 18
 - Conversation Factory
 - transferring data between two platforms, 289
 - Conversion routines
 - creating, 143
 - handling errors, 131
 - properties and methods, 143, 154
 - Courses
 - Construct Spectrum, 20
 - Creating
 - applications, 195
 - assign values to fields in parameter data areas, 195
 - example of code to write to data areas, 195
 - check success of CALLNAT, 196
 - example of code to check CallNat, 196
 - create parameter data area instances, 195
 - example of code declaring variables, 195
 - example of creating a data area, 195
 - use the CallNat method on the client, 196
 - example of code to use the CallNat method, 196
 - domain for your application, 39
 - Customizing
 - ABO
 - using user exits, 100
 - properties generated for the ABO, 98
- ## D
- Data encryption/de-encryption
 - Construct Spectrum applications, 31
 - Data Sizes tab
 - Diagnostics window, 179

- Data translation, 31
 - system functions, 32
- Database record
 - definition of, 258
- DBID
 - definition of, 258
- Debug data
 - Debug Library field, 173
 - generating, 167
 - writing to the source area
 - example of code in subprograms, 170
 - example of results from IF statement, 170
 - example of results without IF statement, 171
- Debugging
 - client/server applications, 162
 - list of error sources, 162
 - runtime errors, 167
 - traditional tools, 163–164
 - communication errors, 167
 - returning information, 32
- Debugging tools
 - client/server
 - Diagnostics window, 177
 - Translations program, 181
 - definition of, 259
 - INPUT statement, 174
 - simulating client calls, 175
 - traditional
 - DISPLAY statements, 163
 - INPUT statements, 163
 - Natural Debugging facility, 163
 - WRITE statements, 163
- Defaulting from Predict
 - business object description, 42
 - GUI controls, 40
 - hold field, 41
 - primary key, 41
- DEFINE PRINTER statement
 - using, 170
- Defining
 - domain for your application, 45
 - security for your application, 47
 - steplib chain for your application, 43
- Dependent object
 - definition of, 259
- Deploying
 - client/server applications, 191
 - collect files for installation, 191
 - create the executable file, 191
 - install the client application, 192
 - run the application, 192
- Deployment
 - definition of, 259
- Descriptions
 - defaulting for business objects, 42
- Determining
 - character set, 290
- Development environments
 - Construct Spectrum Add-Ins
 - Visual Basic, 26
 - Construct Spectrum Administration subsystem, 24
 - Construct Spectrum options
 - Add-Ins menu, 26
 - Construct Windows interface, 25
 - creating a web application, 27
 - integrated tools, 23
 - using an HTML editor, 27
- Development process
 - overview, 36
- Diagnostics window
 - Data Sizes tab, 179
 - description, 177
 - Initialize Data tab, 180
 - Subprogram Proxy tab, 178
 - summary of diagnostic information, 177
 - using, 178
- Dialog
 - definition of, 259
- Dispatch service
 - definition of, 259
- Dispatch service data
 - on mainframe server, 31
- Dispatcher
 - definition of, 259
- Dispatcher objects
 - troubleshooting, 186
- Distributing
 - client/server applications, 191
 - collect files for installation, 191
 - create the executable file, 191
 - install the client application, 192
 - run the application, 192

Documentation
related, 18

Domain
definition of, 259

Domains
defining, 45
defining security, 47

Double-byte character set (DBCS)
definition of, 259

Download data
definition of, 259

E

EBCDIC character set
translation of, 31

EDIT command
Natural
view generated debug members, 173

Enabling trace options in subprogram
proxy, 107

Encapsulation
definition of, 259

Encrypt and decrypt data
system functions, 32

Encryption
definition of, 259

Encryption/de-encryption of data
Construct Spectrum applications, 31

Entire Broker
encapsulation of calls, 33
on mainframe server, 31
on Windows platform, 33
with Microsoft IIS, 35

Entire Broker service settings
definition of, 260

Entire Broker stub
definition of, 260

Error categories
communication, 165
debugging client/server applications,
165
runtime, 165
Spectrum system messages, 165

Error handling
system functions, 32

Errors.bas
description, 91

ETBCB data area
description, 285

Event
definition of, 260

Externally-creatable object
definition of, 260

F

Field
definition of, 260

Field headings
setting up in Predict, 39

FieldRef property
diagram of creating two data areas, 231
diagram of fields defined as objects to
CallNat, 230
diagram of fields passed to CallNat, 228
diagram of using, 231
example, 231
example of code to pass individual
fields, 228
example of using the CUSTA Natural
data area, 232
example of Visual Basic code, 232
syntax, 230

File volume information
specifying in Predict, 42

FNR
definition of, 260

Foreign key
definition of, 260

Form
definition of, 260

Framework components
ABO project, 91

G

Generate
definition of, 260

Generated module
definition of, 261

Generating
 debug data, 167
 saving parameter and debug data, 167
 Generate Trace Code field, 172
 package modules
 Generation subsystem, 82
 subprogram proxy, 105–106
 Generation data cache
 definition of, 261
 Global data areas
 application services, 236
 Global settings
 configuration profile, 50
 Globals.bas
 description, 91
 Grid
 definition of, 261
 Grid control
 definition of, 261
 Group
 definition of, 261
 Grouping related business objects, 45
 GUI
 definition of, 261
 GUI control override
 definition of, 261
 GUI controls
 setting up in Predict, 40
 GUI dialog
 on Windows platform, 34

H

Hold field default
 setting up in Predict, 41
 Hook
 definition of, 261
 Host
 definition of, 261
 HTML fragment
 definition of, 261
 HTML template
 definition of, 261
 HTML Template wizard
 definition of, 261
 http request
 definition of, 262

I

Initialize Data tab
 Diagnostics window, 180
 INPUT statement
 use as debugging tool, 174
 Instantiation
 definition of, 262
 Internationalization
 definition of, 262
 Internet/intranet
 supported browsers, 35
 Invoke Proxy function
 accessing, 176
 Invoke Proxy panel
 description, 176
 Invoking
 Configuration editor, 50

J

Job control language (JCL)
 definition of, 262

K

Keys
 defaulting primary, 41
 Keyword
 definition of, 262
 Knowledge
 assumed, 14

L

Level 1 block optimization
 description, 213
 diagram of client and server as sender
 and receiver, 215
 directional attributes
 example, 214
 list of directional attributes, 213
 Level 1 data block
 definition of, 262
 Level 1 data block optimization
 definition of, 262
 Libraries
 adding to your steplib chain, 44

- Library image file (LIF)
 - definition of, 262
- Library image files
 - simulated PDAs, 33
- LIF definitions module
 - definition of, 262
- LIFDefinitions.bas
 - description, 91
- LIST command
 - Natural
 - view generated debug members, 173
- Localization
 - definition of, 262
- Log utilities
 - supplied with Construct Spectrum, 292
- Lookups
 - definition of, 262
- M**
- Maintain Domains Table panel
 - description, 46
- Maintain Steplib Table panel
 - description, 44
- Maintain User Table panel
 - accessing, 173
- Maintenance dialog
 - definition of, 262
- Maintenance subprogram proxy
 - level 1 blocks sent for default methods, 116
- Mapper function
 - description, 130
- MDI child
 - definition of, 263
- MDI frame
 - definition of, 263
- MDI parent
 - definition of, 263
- Menu
 - definition of, 263
- Menu bar
 - definition of, 263
- Message handling
 - system functions, 32
- Message protocol
 - MESSAGE(A1), 273
 - MSG-NR(N4), 273
 - REPLICA-ID(A32), 273
 - RESPONSE-CODE(N4), 273
 - SIGNATURE(A6), 273
 - SPECTRUM-SERVICE(A32), 273
 - SYSERR-LIBRARY(A8), 273
- Metadata
 - definition of, 263
- Method
 - definition of, 263
- Methods
 - Abort, 224
 - adding, 113
 - application service definitions, 113
 - updating application service definitions, 113
 - updating LIFs, 114
 - Allocate, 201
 - Commit, 224
 - GetField, 204
 - Initialize, 201
 - Reset, 205
 - SetField, 205
 - StartTransaction, 224
- Middleware
 - relationship with Construct Spectrum, 22
- Model
 - definition of, 263
- Module
 - definition of, 263
- Multi-level security
 - definition of, 264
- Multiple-document interface (MDI)
 - definition of, 264
- Multi-tasking verification utility
 - description, 291

N

Natural Construct nucleus
definition of, 264

Natural data area
simulation, 198

- Allocate method, 201
- application object properties or methods, 201
- creating NaturalDataArea objects, 202
- data area definitions, 198
- definition, 198
- diagram of components, 198
- diagram of objects in data area simulation, 200
- example of a data area definition, 208
- example of code for data area definition, 199
- example of code for redefining, 207
- example of code to declare and initialize the application object, 201
- example of code using structure name as a qualifier, 206
- example of reading arrays with the GetField method, 208
- example of reading occurrences of the Item array, 209
- example of specifying a field with occurrences, 208
- example of using redefined fields, 207
- Initialize method, 201
- LIFDirectory property, 201
- list of features in data area definitions, 199
- MainLibrary property, 201
- NaturalDataArea class, 202–203
- NaturalDataArea object, 206
- NaturalFieldDef class, 210
- simulation objects, 200
- syntax for Allocate method, 202

Natural Debugging facility
definition of, 264

Natural security
user information, 173

Natural source areas
writing information to, 236

Natural subprograms
example of creating, 238–239
PDA, 238
on mainframe server, 30

NaturalDataArea object
troubleshooting, 185

Navigation bar
definition of, 264

Node
definition of, 264

Nodes
marking for refresh, 70
removing from cache, 70

Nucleus
definition of, 264

O

Object
definition of, 264

Object factory
definition of, 264

Object Factory wizard
definition of, 264

Object library
definition of, 264

OLE
definition of, 265

OLE automation server
definition of, 265

Options window
customizing the ABO
description, 99

Overflow condition
definition of, 265

Overriding
1:V variables, 108
domain steplib chain, 115

P

Package
definition of, 265

Page handler
definition of, 265

Page Handler wizard
definition of, 265

Parameter alignment problems
diagnosing, 177

- Parameter and debug data
 - accessing the Maintain User Table panel, 173
 - using, 172
- Parameter data areas
 - application services
 - data size limitations, 237
 - example of creating, 238
 - simulation by Spectrum Dispatch Client, 33
- Parameters
 - externalizing, 237
- Parent model
 - definition of, 265
- Parse area
 - definition of, 265
- Partner products
 - Construct Spectrum, 22
- Ping
 - definition of, 265
- Platform
 - definition of, 265
- Predict data dictionary
 - relationship with Construct Spectrum, 22
- Predict set up tasks
 - business data types, 40
 - default business object description, 42
 - default GUI controls, 40
 - default hold field, 41
 - default primary key, 41
 - field headings, 39
 - file volume information, 42
 - verification rules, 40, 42
- Prerequisites
 - assumed knowledge, 14
- Preserving characters
 - Translation program, 181
- Primary keys
 - defaulting from Predict, 41
- Printable characters
 - Translation program, 181
- Programming languages
 - incorporating with Construct Spectrum, 23
- Project
 - definition of, 265

- Project group
 - definition of, 265
- Properties
 - CheckFieldSpec, 203, 209
 - Decimals, 210
 - DefinedRank, 210
 - Definition, 203, 209
 - Field, 203
 - FieldDef, 204, 209
 - FieldDefs, 204, 210
 - FieldRef, 204
 - Format, 210
 - FormatLength, 210
 - FromIndex, 211
 - Length, 211
 - Level, 211
 - LevelTypeTrail, 211
 - LibraryImageFile, 204
 - LIFDirectory, 201
 - MainLibrary, 201
 - Name, 204, 212
 - PackedData, 204
 - PackedDataLength, 205, 210
 - Rank, 212
 - Redefined, 212
 - Structure, 212
 - ThruIndex, 211–212
 - TransactionActive, 224
- Property
 - definition of, 265

R

- Regenerate/preserve status
 - definition of, 266
- Remote call
 - definition of, 266
- Replacement tag
 - definition of, 266
- Reports
 - using, 63
- RequestProperty properties
 - Spectrum Dispatch Client, 186
- Resource
 - definition of, 266
- Run
 - definition of, 266

- Running
 - applications, 253
- Runtime errors
 - listing, 165
 - results, 175
- S**
- Security
 - defining for a domain, 47
- Security cache
 - definition of, 266
- Security services
 - on mainframe server, 31
- Sending
 - responses back to client, 272
- Server
 - components
 - setting up for communication with client, 236
 - definition of, 266
- Server application
 - definition of, 266
- Service
 - definition of, 266
- Service exit
 - definition of, 266
- Service log
 - definition of, 266
- Set up checklists
 - see* Checklists, 38
- Setting
 - trace options, 167
- Setting up
 - security for your application, 43
- Settings for Profile tab
 - Configuration editor, 52
- Shutdown
 - definition of, 266
- Software development kit (SDK)
 - definition of, 266
- SPAETB data area
 - description, 280
- Specifying
 - block handling on the server, 118
 - general package parameters
 - Business-Object-Super-Model, 78
 - overrides, 118
 - specific package parameters
 - Business-Object-Super-Model, 79, 81
 - standard parameters
 - Business-Object-Super-Model, 77
- Specifying defaults
 - hold key, 41
 - primary key, 41
- Spectrum administration
 - on mainframe server, 31
- Spectrum client/server application
 - definition of, 266
- Spectrum Control record
 - definition of, 267
- Spectrum Dispatch Client
 - advanced features, 228
 - 1 to V fields, 233
 - FieldRef property, 228
 - application service definitions, 215
 - example in a library image file, 216
 - CALLNAT simulation, 33
 - client/server communication, 213
 - application service, 197
 - components, 197
 - data area simulation, 198
 - level 1 block optimization, 213
 - client/server communication components
 - definitions, 197
 - dispatch service definitions, 197
 - Dispatcher objects, 197
 - creating applications
 - See* creating applications, 195
 - data area simulation components, 197
 - data area allocator, 197
 - data area definitions, 197
 - data area objects, 197
 - database transaction control, 34
 - Dispatcher objects and dispatch service definitions, 218
 - compression and encryption, 223
 - database transaction control, 224
 - diagram of Dispatcher objects, 218
 - diagram of timeout functionality, 222
 - example of code to create Dispatcher objects, 218
 - example of implementing level 1 block optimization, 220
 - example of resuming a call, 223
 - list of error types, 225
 - remote subprogram invocation, 220

- service selection, 219
 - timeout, retry, and resume handling, 221
 - tracing, 224
 - user identification and authentication
 - application properties, 226
 - encapsulation of Entire Broker calls, 33
 - functions
 - client/server communication, 197
 - Natural data area simulation, 197
 - initializing
 - Project Properties dialog, 251
 - library image files and the steplib chain, 227
 - syntax of the steplib definition, 227
 - overview, 194
 - properties, 185
 - with Microsoft IIS, 35
- Spectrum Dispatch Client (SDC)
 - definition of, 267
 - on Windows platform, 33
- Spectrum dispatch service
 - definition of, 267
 - on mainframe server, 31
 - running online, 174
- Spectrum security service
 - definition of, 267
- Spectrum security services
 - on mainframe server, 31
- Spectrum Service Manager
 - definition of, 267
- Spectrum service settings
 - definition of, 267
- Spectrum web application
 - definition of, 267
- Spectrum web framework
 - definition of, 267
- Spectrum XML Cache Viewer
 - overview, 68
 - refreshing, 70
- SPSTLATE utility, 290
- SPUETB subprogram
 - wrapping Broker ACI calls, 278
- SPUREPLY subprogram
 - sending responses back to client, 272
- Status bar
 - definition of, 267
- Steplib chain
 - defining, 43
 - definition of, 267
- Sub Main procedure
 - definition of, 267
- Subprogram proxies
 - generating using model, 240
 - invoking online, 175
 - setting trace code option, 107
 - Spectrum dispatch service, 31
- Subprogram proxy
 - definition of, 267
 - methods generated, 111
 - on mainframe server, 30
 - overview
 - application service definition methods
 - See* Application service definitions, 111
 - prerequisites
 - Construct Spectrum Administration subsystem data files, 111
 - generating subprograms and object PDAs, 111
 - versioning, 120
 - security implications, 120
- Subprogram Proxy tab
 - Diagnostics window, 178
- Subprogram-Proxy model
 - adding user exits, 109
 - overview, 104
 - standard parameters, 106
 - using, 105
- Subprograms
 - behavior, 237
 - interfacing, 236
- Super model
 - definition of, 268

T

- Target module
 - definition of, 268
- Target object
 - definition of, 268
- Target subprogram
 - definition of, 268

- Template parser
 - definition of, 268
- Timing issues
 - application services, 237
- Toolbar
 - definition of, 268
- Toolbar button
 - definition of, 268
- Toolkit
 - definition of, 268
- Tools
 - debugging
 - client/server applications, 163
- Trace options
 - definition of, 268
 - setting, 167
 - Subprogram Proxy Trace Options window, 168
- Trace-Option(1)
 - description, 168
- Trace-Option(2)
 - description, 172
 - Generate Trace Code field, 172
 - valid values, 172
- Transferring
 - data between two platforms, 289
- Translating
 - character sets, 290
- Translation Mappings window
 - description, 182
- Translation of character sets, 31
- Translations program
 - ASCII/EBCDIC, 181
 - character sets, 181
 - altered, 181
 - preserved, 181
 - printable, 181
 - translation tables, 181
- Troubleshooting
 - Construct Spectrum Add-In, 184
 - Construct Spectrum dispatch client properties, 185
- Type library
 - definition of, 268

U

- Updating
 - application service definitions, 113
 - LIF files, 114
- Upload data
 - definition of, 268
- User exits
 - customizing the ABO, 100
- User interface
 - creating, 251
 - example form, 251
- Utilities
 - log, 292
 - multi-tasking verification, 291
 - sending responses back to client
 - SPUREPLY subprogram, 272
 - transferring data between two platforms
 - Conversation Factory, 289
 - translating character sets
 - SPSTLATE subprogram, 290
 - wrapping Broker ACI calls
 - SPUETB subprogram, 278
- Utility.bas
 - description, 91

V

- Variant
 - definition of, 268
- VB-Client-Server-Super-Model
 - definition of, 268
- Verification rules
 - definition of, 269
 - setting up in Predict, 40, 42
- Visual Basic browse object
 - definition of, 269
- Visual Basic business object
 - definition of, 269
 - on Windows platform, 34
- Visual Basic maintenance object
 - definition of, 269
- Volume information
 - see* File volume information

W

Web application
with Microsoft IIS, 35

Web application ASP
definition of, 269

Web class
definition of, 269

Web Super wizard
definition of, 269

Wildcard
definition of, 269

Wrapping
Broker ACI calls, 278

X

XML extract
definition of, 270