GemStone

GemBuilder for C

July 1996

GemStone

Version 5.0

IMPORTANT NOTICE

This manual and the information contained in it are furnished for informational use only and are subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this manual or in the information contained in it. The manual, or any part of it, may not be reproduced, displayed, photocopied, transmitted or otherwise copied in any form or by any means now known or later developed, such as electronic, optical or mechanical means, without written authorization from GemStone Systems, Inc. Any unauthorized copying may be a violation of law.

The software installed in accordance with this manual is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Copyright 1996 by GemStone Systems, Inc. All rights reserved.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

GemStone is a registered trademark of GemStone Systems, Inc.

Preface

About This Manual

This manual describes GemBuilder for C — a set of C functions that provide a bridge between your application's C code and the application's database controlled by GemStone®. These GemBuilder functions provide your C program with complete access to a GemStone database of objects, and to a virtual machine on which to execute GemStone Smalltalk code.

Assumptions

To make use of the information in this manual, you must be familiar with the GemStone Smalltalk programming language, as described in the *GemStone Programming Guide* manual.

In addition, you must know the C programming language, as described in Kernighan and Ritchie's *The C Programming Language* (Prentice Hall, 1978).

Finally, you should be familiar with your C compiler, as described in its user documentation.

This manual assumes that the GemStone database system has been correctly installed on your host computer as described in the *GemStone System*

Administration Guide, and that your system meets the requirements listed in the *GemStone* or *GemBuilder for C Installation Guide*.

How This Manual is Organized

Chapter 1, "Introduction," describes the GemBuilder functions in general, and how they are used in application development with GemStone.

Chapter 2, "Building Applications With GemBuilder for C," introduces the two versions of GemBuilder and explains how to build applications that bind to GemBuilder at run time or build time.

Chapter 3, "Writing C Functions to be Called from GemStone," describes how to implement "user action" routines that can be called from GemStone Smalltalk methods.

Chapter 4, "The Mechanics of Compiling and Linking," describes how to compile and link your C applications and user actions, and how to install them in a GemStone environment prior to execution.

Chapter 5, "GemBuilder C Functions — A Reference Guide," provides a detailed description of each GemBuilder function, including: synopsis (syntax), parameters, return value, a general description of what the function does, one or more examples of its use, and a list of related GemBuilder functions.

Conventions

In examples, system output is shown in `monospace` typeface, while commands you type are shown in **bold** typeface. Place holders that are meant to be replaced with real values are shown in *italic* typeface. For example:

```
% startstone myStone
startstone[Info]:
 GEMSTONE_SYS_CONF=/user1/gemstone/data/system.conf
 GEMSTONE_EXE_CONF=/user1/gemstone/data/system.conf
...
```

Other Useful Documents

For more information about the GemStone data management system and its development tools, see the *GemStone Programming Guide* and the *GemStone Kernel Reference*. The *GemStone Programming Guide* offers a tutorial approach to

GemStone Smalltalk, GemStone's object-oriented language. The *GemStone Kernel Reference* includes a full protocol for each of the Smalltalk kernel classes.

Technical Support

GemStone provides several sources for product information and support. GemStone product manuals provide extensive documentation, and should always be your first source of information. GemStone Technical Support engineers will refer you to these documents when applicable. However, you may need to contact Technical Support for the following reasons:

- Your technical question is not answered in the documentation.
- You receive an error message that directs you to contact GemStone Technical Support.
- You want to report a bug.
- You want to submit a feature request.

Questions concerning product availability, pricing, keyfiles, or future features should be directed to your GemStone account manager.

When contacting GemStone Technical Support, please be prepared to provide the following information:

- Your name, company name, and GemStone license number,
- the GemStone product and version you are using,
- the hardware platform and operating system you are using,
- a description of the problem or request,
- exact error message(s) received, if any.

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

For non-emergency requests, you should contact Technical Support by email, Web form, or facsimile. You will receive confirmation of your request, and a request assignment number for tracking. Replies will be sent by email whenever possible, regardless of how they were received.

Email:support@gemstone.com
The preferred method of contact. Please do not send files larger than 100K (for

example, core dumps) to this address. A special address for large files will be provided on request.

World Wide Web:http://www.gemstone.com

Technical Support is located under Services. A Help Request Form is available for request submissions. This form requires a password, which is free of charge but must be requested by completing the Registration Form, found in the same location. Allow 24 hours for your registration to be recorded and a password assigned.

Facsimile:(503) 629-8556

When you send a fax to Technical Support, you should also leave a voicemail message to make sure your fax will be picked up as soon as possible.

We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production database that is non-operational.

Telephone:(800) 243-4772 or (503) 690-3503

Emergency requests will be handled by the first available engineer. If you are reporting an emergency and you receive a recorded message, do not use the voicemail option. Transfer your call to the operator, who will take a message and immediately contact an engineer.

Non-emergency requests received by telephone will be placed in the normal support queue for evaluation and response.

24x7 Emergency Technical Support

GemStone offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact GemStone 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. Contact your GemStone account manager for more details.

Contents

Chapter 1. Introduction

1.1 GemBuilder Application Overview 1-1
Deciding Where to Do the Work. 1-3
Representing GemStone Objects in C 1-3
Smalltalk Access to Objects 1-3
Calling C Functions from Smalltalk Methods 1-4
The GemBuilder Functions. 1-4
1.2 Session Control 1-5
Starting and Stopping GemBuilder 1-5
Remote Login Setup. 1-5
Logging In and Out 1-6
Transaction Management. 1-6
Committing a Transaction 1-6
Aborting a Transaction 1-7
Controlling Transactions Manually 1-7
1.3 Representing Objects in C 1-8
GemStone-defined Object Mnemonics 1-8
Converting Between Special Objects and C Values 1-9
Byte Swizzling of Binary Floating-Point Values 1-11

1.4 Manipulating Objects in GemStone 1-12
Sending Messages to GemStone Objects 1-12
Executing Code in GemStone 1-13
Interrupting GemStone Execution 1-14
Modification of Classes. 1-15
1.5 Manipulating Objects Through Structural Access 1-16
Direct Access to Metadata 1-17
Byte Objects 1-17
Pointer Objects. 1-18
Non-sequenceable Collections (NSC Objects) 1-20
1.6 Creating Objects 1-21
1.7 Fetching and Storing Objects 1-22
Efficient Fetching and Storing with Object Traversal 1-22
How Object Traversal Works 1-23
The Object Traversal Functions 1-24
Efficient Fetching And Storing with Path Access 1-24
Path Representations 1-25
1.8 Nonblocking Functions 1-25
1.9 Operating System Considerations 1-27
Interrupt Handling in your GemBuilder Application 1-27
Executing Host File Access Methods 1-28
1.10 Error Handling and Recovery 1-28
Polling for Errors 1-29
Error Jump Buffers 1-29
The Call Stack 1-30
GemStone System Errors. 1-30
1.11 Garbage Collection 1-31
1.12 Preparing to Execute GemStone Applications 1-32
Kerberos 1-32
GemStone Environment Variables 1-32

Chapter 2. Building Applications With GemBuilder for C

2.1 GciRpc and GciLnk. 2-1
Use GciRpc for Debugging. 2-2
Use GciLnk for Performance. 2-2
Multiple GemStone Sessions. 2-2

2.2 The GemBuilder Shared Libraries. 2-2
2.3 Binding to GemBuilder at Run Time 2-3
Building the Application 2-3
Searching for the Library 2-4
How UNIX Matches Search Names with Shared Library Files 2-4
How Windows NT Matches Search Names with DLL Files . 2-4
Build-time Binding 2-5

Chapter 3. Writing C Functions to be Called from GemStone

3.1 Shared User Action Libraries 3-1
3.2 How User Actions Work 3-2
3.3 Developing User Actions. 3-3
Write the User Action Functions. 3-3
Create a User Action Library. 3-4
The `gciua.hf` Header File 3-4
The Initialization and Shutdown Functions 3-4
Compiling and Linking Shared Libraries 3-6
Using Existing User Actions in a User Action Library. 3-6
Using Third-party C Code With a User Action Library 3-6
Loading User Actions. 3-6
Loading User Action Libraries At Run Time 3-7
Specifying the User Action Library 3-8
Creating User Actions in Your C Application 3-8
Verify That Required User Actions Have Been Installed . . . 3-9
Write the Code That Calls Your User Actions 3-9
Remote User Actions 3-10
Limit on Circular Calls Among User Actions and Smalltalk . 3-10
Debug the User Action 3-10
3.4 Executing User Actions. 3-10
Choosing Between Session and Application User Actions 3-10
Running User Actions with Applications. 3-12
With an RPC Application 3-12
With a Linked Application 3-13
Running User Actions with Gems. 3-14
Running User Actions with Applications and Gems 3-15

Chapter 4. The Mechanics of Compiling and Linking

4.1 Development Environment and Standard Libraries 4-2
4.2 Compiling C Source Code for GemStone 4-2
The C Compiler 4-2
Compilation Options 4-2
Compilation Command Lines 4-3
4.3 Linking C Object Code with GemStone 4-4
Run-time and Build-time Binding of Shared Libraries 4-4
Risk of Database Corruption. 4-4
GemStone Link Files 4-5
The Linker 4-6
Link Options. 4-6
Command Line Assumptions 4-6
Linking Applications That Bind to GemBuilder at Run Time 4-7
Linking Applications That Bind to GemBuilder at Build Time 4-7
Linking User Actions into Shared Libraries 4-8

Chapter 5. GemBuilder C Functions — A Reference Guide

5.1 Function Summary Tables. 5-1
5.2 GemBuilder Include Files 5-10
5.3 GemBuilder Data Types. 5-11
The Structure for Representing the Date and Time 5-12
The Error Report Structure. 5-12
The Object Information Structure 5-13
The Object Report Structure 5-14
The Object Report Header Structure 5-15
The User Action Information Structure. 5-17
5.4 Structural Access Functions 5-18
5.5 Unix Interrupt Handling. 5-18
5.6 Reserved Prefixes. 5-18
5.7 GemBuilder Function and Macro Reference. 5-19
GciAbort 5-20
GciAddOopToNsc 5-22
GciAddOopsToNsc 5-23
GciAddSaveObjsToReadSet 5-24

GciAlteredObjs 5-25
GCI_ALIGN 5-27
GciAppendBytes 5-28
GciAppendChars 5-29
GciAppendOops 5-30
GciBegin 5-31
GCI_BOOL_TO_OOP 5-32
GciCallInProgress 5-33
GciCheckAuth 5-34
GCI_CHR_TO_OOP 5-36
GciClampedTrav 5-37
GciClampedTraverseObjs 5-40
GciClassMethodForClass 5-42
GciClassNamedSize 5-44
GciClearStack 5-45
GciCommit 5-48
GciContinue 5-49
GciContinueWith 5-51
GciCreateByteObj 5-53
GciCreateOopObj 5-55
GciCTimeToDateTime 5-57
GciDateTimeToCTime 5-58
GciDbgEstablish 5-59
GciDirtyObjsInit 5-61
GciDirtySaveObjs 5-62
GciEnableSignaledErrors 5-64
GciErr 5-66
GciExecute 5-68
GciExecuteFromContext 5-70
GciExecuteStr 5-72
GciExecuteStrFromContext 5-74
GciExecuteStrTrav 5-76
GciFetchByte 5-79
GciFetchBytes 5-81
GciFetchChars 5-84
GciFetchClass 5-85
GciFetchDateTime 5-87
GciFetchNamedOop 5-88
GciFetchNamedOops 5-90

GciFetchNamedSize 5-92
GciFetchNameOfClass 5-93
GciFetchObjImpl 5-94
GciFetchObjectInfo 5-95
GciFetchObjInfo 5-97
GciFetchOop 5-99
GciFetchOops 5-101
GciFetchPaths 5-103
GciFetchSize 5-108
GciFetchVaryingOop 5-110
GciFetchVaryingOops 5-113
GciFetchVaryingSize 5-115
GciFindObjRep 5-117
GciFltToOop 5-119
GciGetFreeOop 5-120
GciGetFreeOops 5-122
GciGetSessionId 5-124
GciHandleError 5-125
GciHardBreak 5-127
GciInit 5-128
GciInitAppName 5-129
GciInstMethodForClass 5-130
GciInstallUserAction 5-132
GciInUserAction 5-133
GciIsKindOf 5-134
GciIsKindOfClass 5-135
GciIsRemote 5-136
GCI_IS_REPORT_CLAMPED 5-137
GciIsSubclassOf 5-138
GciIsSubclassOfClass 5-139
GciIvNameToIdx 5-140
GciLoadUserActionLibrary 5-142
GciLogin 5-144
GciLogout 5-146
GCI_LONG_IS_SMALL_INT 5-147
GciLongToOop 5-148
GCI_LONG_TO_OOP 5-150
GciMoreTraversal 5-152
GciNbAbort 5-155

GciNbBegin 5-156
GciNbClampedTrav 5-157
GciNbClampedTraverseObjs 5-158
GciNbCommit 5-160
GciNbContinue 5-161
GciNbContinueWith 5-162
GciNbEnd 5-163
GciNbExecute 5-165
GciNbExecuteStr 5-167
GciNbExecuteStrFromContext 5-169
GciNbExecuteStrTrav 5-171
GciNbMoreTraversal 5-173
GciNbPerform 5-175
GciNbPerformNoDebug 5-177
GciNbPerformTrav 5-179
GciNbStoreTrav 5-181
GciNbTraverseObjs 5-183
GciNewByteObj 5-185
GciNewCharObj 5-186
GciNewDateTime 5-187
GciNewOop 5-188
GciNewOops 5-189
GciNewOopUsingObjRep 5-191
GciNewString 5-194
GciNewSymbol 5-195
GciObjExists 5-196
GciObjInCollection 5-197
GciObjRepSize 5-198
GCI_OOP_IS_BOOL 5-200
GCI_OOP_IS_SMALL_INT 5-201
GCI_OOP_IS_SPECIAL 5-202
GciOopToBool 5-203
GCI_OOP_TO_BOOL 5-204
GciOopToChr 5-205
GCI_OOP_TO_CHR 5-206
GciOopToFlt 5-207
GciOopToLong 5-209
GCI_OOP_TO_LONG 5-211
GciPathToStr 5-213

GciPerform 5-216
GciPerformNoDebug 5-218
GciPerformSym 5-220
GciPerformTrav 5-222
GciPerformTraverse 5-224
GciPollForSignal 5-227
GciPopErrJump 5-229
GciProcessDeferredUpdates 5-231
GciPushErrHandler 5-232
GciPushErrJump 5-233
GciRaiseException 5-237
GciReleaseAllOops 5-238
GciReleaseOops 5-239
GciRemoveOopFromNsc 5-241
GciRemoveOopsFromNsc 5-243
GciReplaceOops 5-245
GciReplaceVaryingOops 5-247
GciResolveSymbol 5-248
GciResolveSymbolObj 5-249
GciRtlIsLoaded 5-250
GciRtlLoad 5-251
GciRtlUnload 5-253
GciSaveObjs 5-254
GciSendMsg 5-255
GciSessionIsRemote 5-257
GciSetErrJump 5-258
GciSetNet 5-260
GciSetSessionId 5-263
GciShutdown 5-264
GciSoftBreak 5-265
GciStoreByte 5-266
GciStoreBytes 5-268
GciStoreBytesInstanceOf 5-270
GciStoreChars 5-272
GciStoreIdxOop 5-274
GciStoreIdxOops 5-276
GciStoreNamedOop 5-278
GciStoreNamedOops 5-280
GciStoreOop 5-282

GciStoreOops 5-284
GciStorePaths 5-287
GciStoreTrav 5-292
GciStrKeyValueDictAt 5-296
GciStrKeyValueDictAtObj 5-297
GciStrKeyValueDictAtObjPut 5-298
GciStrKeyValueDictAtPut 5-299
GciStrToPath 5-300
GciSymDictAt 5-303
GciSymDictAtObj 5-304
GciSymDictAtObjPut 5-305
GciSymDictAtPut 5-306
GciTraverseObjs 5-307
GciUnsignedLongToOop 5-313
GciUserActionInit 5-314
GCI_VALUE_BUFF 5-315
GciVersion 5-317

Appendix A. Reserved OOPs

Appendix B. Network Resource String Syntax

B.1 Overview B-1
B.2 Defaults B-2
B.3 Notation B-3
B.4 Syntax B-4

Appendix C. Linking to Static User Action Code

C.1 Creating the Custom Gem. C-1
C.2 Deploying Static User Actions for Custom Gems. C-2
How GemStone Starts Gem Processes C-2
Starting a Private Custom Gem Under Unix C-3
C.3 Name Conflicts with Dynamic User Actions C-4

Tables

Table 5.1. Functions for Controlling Sessions and Transactions 5-1
Table 5.2. Functions for Handling Errors and Interrupts and for Debugging. . 5-3
Table 5.3. Functions for Compiling and Executing Smalltalk Code in the Database 5-4
Table 5.4. Functions for Accessing Symbol Dictionaries. 5-5
Table 5.5. Functions for Creating and Initializing Objects. 5-5
Table 5.6. Functions and Macros for Converting Objects and Values 5-6
Table 5.7. Object Traversal and Path Functions and Macros 5-7
Table 5.8. Structural Access Functions. 5-8
Table 5.9. Object Implementation Restrictions on Instance Variables 5-17
Table 5.10. Differences in Reported Object Size 5-109

Figures

Figure 1.1. The Role of GemBuilder in Application Development 1-2
Figure 1.2. Object Traversal and Paths. 1-23
Figure 3.1. Access to Application and Session User Actions 3-12
Figure 3.2. Application User Actions and RPC Applications in GemStone Processes. 3-12
Figure 3.3. Session User Actions and Linked Applications in GemStone Processes. 3-13
Figure 3.4. Session User Actions and RPC Gems in GemStone Processes . . . 3-14
Figure 3.5. RPC Applications and Gems with User Actions in GemStone Processes. 3-15
Figure 3.6. Application and Session User Actions in GemStone Processes . . 3-16

Index

Chapter 1 Introduction

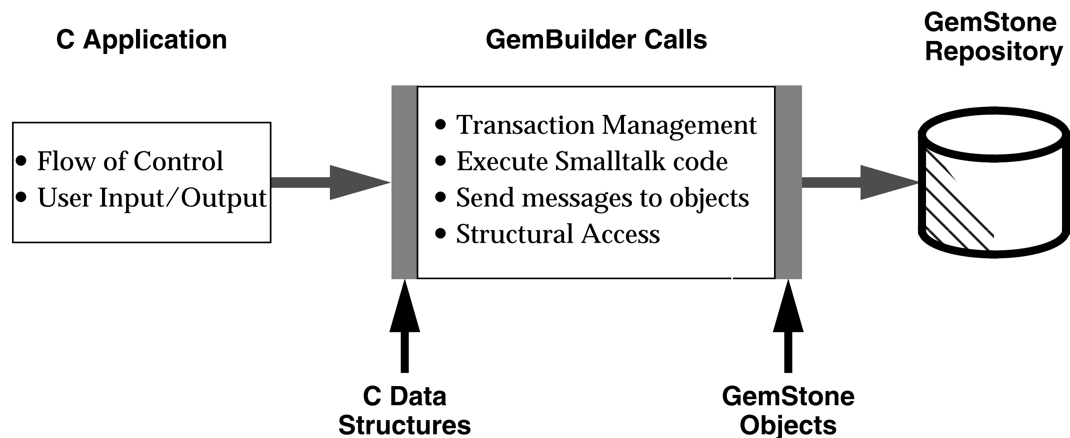
GemBuilder for C is a set of C functions that provide your C application with complete access to a GemStone repository and its programming language, Smalltalk[1]. The GemStone object server contains your schema (class definitions) and objects (instances of those classes), while your C program provides the user interface for your GemStone application. The GemBuilder functions allow your C program to access the GemStone repository either through structural access (the C model) or by sending messages (the Smalltalk model). Both of these approaches are discussed in detail later in this chapter.

1.1 GemBuilder Application Overview

Figure 1.1 illustrates the role of GemBuilder in developing a GemStone application. In effect, developing your GemStone application consists of two separate efforts: creating Smalltalk classes and methods, and writing C code.

1. GemStone embeds a variety of the Smalltalk language within the repository. It is separate from but similar to other varieties of Smalltalk that are sold commercially. Smalltalk serves as the data definition and data manipulation language for GemStone, and provides the repository with its ability to identify, access, and manipulate objects internally. When this manual mentions Smalltalk, it generally is referring to GemStone's internal language.

Figure 1.1 The Role of GemBuilder in Application Development



We recommend the following steps for developing your hybrid application:

Step 1. Define the application's external interface.

Any GemBuilder application must manage its user interface through custom modules written in C.

Step 2. Decide where to perform the work.

Applications that are a hybrid of C functions and Smalltalk classes pose interesting problems to the designer: Where is the best place to perform the application's work? Is it better to import the representation of an object into your C program and perform the work there, or to send a message which invokes a Smalltalk method? In the next section, we'll examine this question in more detail.

Step 3. Implement and debug the application.

After you've developed a satisfactory design, you can implement and test the C-based functions using familiar techniques and tools (editor, C compiler, link editor, debugger). For information about implementing applications, see Chapter 2, "Building Applications With GemBuilder for C."

Step 4. Compile and link the application.

For instructions about compiling and linking your application, please see Chapter 4, "The Mechanics of Compiling and Linking." For full details, see your C compiler user documentation.

Deciding Where to Do the Work

As mentioned above, you will need to decide how much of the application's work to perform in C functions and how much in Smalltalk methods. The following paragraphs discuss both approaches.

Representing GemStone Objects in C

You may choose to implement C functions that access GemStone objects for manipulation in your C program. In such cases, a representation of each object must be imported from GemStone into your C program before the C function is executed. GemBuilder provides functions for importing objects from GemStone to your C program, creating new GemStone objects, directly accessing and modifying the internal contents of objects, and exporting objects to the GemStone repository.

Of course, if you import an object to your C program and modify it, or if you create a new object within your C program, your application must export the new or modified object to GemStone before it can commit the changes to the repository.

Here are some advantages of using GemBuilder structural access functions to modify objects:

- It may be more efficient to perform a function in C than in Smalltalk.
- The function may need to be closely linked with I/O functions for the user interface.
- The function may already exist in a standard library. In this case, the data must be transported from GemStone to that function.

The section "Manipulating Objects Through Structural Access" on page 1-16 defines exactly how objects are represented in C as address space, and defines the GemBuilder functions for exchanging these structures between GemStone and C.

Smalltalk Access to Objects

In many cases, you will choose to perform your GemStone work directly in Smalltalk. GemBuilder provides C functions for defining and compiling Smalltalk methods for a class, and for sending a message to an object (invoking a Smalltalk method). Here are some advantages of writing a function directly in Smalltalk:

- The integrity of the data encapsulation provided by the object metaphor is preserved.
- Functions in Smalltalk are more easily shared among multiple applications.

- Functions in Smalltalk may be easier to implement. There is no need to worry about moving objects between C and Smalltalk or about space management.
- The overhead of transporting objects between C and Smalltalk is avoided.
- Classes or methods may already exist which exhibit behavior similar to the desired behavior. Thus, less effort will be required to implement a new function in Smalltalk.

The section “Manipulating Objects in GemStone” on page 1-12 defines the GemBuilder functions that allow C applications to send Smalltalk messages to objects and execute Smalltalk code.

Calling C Functions from Smalltalk Methods

Even though you may choose to perform your GemStone work in Smalltalk, you may find that you need to access some functions written in C. GemBuilder allows you to link your user-written C functions to a GemStone session process, and subsequently call those functions from Smalltalk. For example, operations that are computationally intensive or are external to GemStone can be written as C functions and called from within a Smalltalk method (whose high-level structure and control is written in Smalltalk). This is similar to the concept of “user-defined primitives” offered by other object-oriented systems. Here are some advantages of calling C functions from Smalltalk:

- For computationally intensive portions of a GemStone operation, C functions may execute faster than the same functions written in Smalltalk.
- Operating system services, or services of other software systems, can be accessed without the overhead of spawning a subprocess. In addition, using C functions to access such services provides greater flexibility for passing arguments and returning results.

Chapter 3, “Writing C Functions to be Called from GemStone,” describes how to implement “user action” routines that can be called from Smalltalk methods, and how to link those routines into a GemBuilder application or a Gem (GemStone session) process.

The GemBuilder Functions

The remainder of this chapter introduces you to many of the GemBuilder C functions.

- First, we’ll look at functions used in managing GemStone sessions: logging into (and out of) GemStone, switching between multiple sessions, and committing and aborting transactions.

- Next, we'll look at functions that allow your C program to manipulate objects by sending Smalltalk messages or executing Smalltalk code fragments.
- Finally, we'll examine those functions that perform "structural access" upon the representation of objects within your C program.

1.2 Session Control

All interactions with the GemStone repository monitor occur within the scope of a user's GemStone session, which may encapsulate one or more individual transactions. GemBuilder provides functions for obtaining and managing GemStone repository sessions, such as logging in and logging out, committing and aborting transactions, and connecting to a different session.

Starting and Stopping GemBuilder

The functions **GciInitAppName** and **GciInit** initialize GemBuilder. When it is used, your application should call **GciInitAppName** before calling **GciInit**. Your C application must not call any other GemBuilder functions until it calls **GciInit**.

The function **GciShutdown** logs out all sessions that are connected to the Gem and deactivates GemBuilder. Your C application should call **GciShutdown** before exiting, in order to guarantee that the process deallocates its resources.

Remote Login Setup

There are three ways to prepare for remote login to a GemStone repository:

1. First, you use a netldi that is running in guest mode, attached to the Stone process. Guest mode provides easy access in situations where it is not considered necessary to authenticate users in the network environment before permitting them to log in.
2. Second, you can use the **kerberos** system for authenticating users. See your kerberos documentation and man pages for details.
3. Otherwise, you need to have a `.netrc` file in your `$HOME` directory. This file contains remote login data: the name of your host machine, your login name, and your host machine password, in the following format:

   `machine` ***host_machine_name*** `username` ***name*** `password` ***passwd***

   If you will be using more than one host machine, you will need a separate entry in this file for each machine, with each entry on its own line.

You may also wish to set the GEM_RPCGCI_TIMEOUT configuration parameter in the GemStone configuration file you use when starting a remote Gem. This parameter sets a timeout limit for the remote Gem; if the Gem remains inactive too long, GemStone logs out the session and terminates the Gem process. See the *GemStone System Administration Guide* for more details.

Logging In and Out

Before your C application can perform any useful repository work, it must create a session with the GemStone system by calling **GciLogin**. That function uses the network parameters initialized by **GciSetNet**.

If your application calls **GciLogin** again after you are already logged in, GemBuilder will create an additional, independent, GemStone session for you. Multiple sessions can be attached to the same GemStone repository, or they can be attached to different repositories. The maximum number of sessions that may be logged in at one time depends upon your version of GemStone and the terms of your license agreement.

From the point of view of GemBuilder, only a single session is active at any one time. It is known as the *current session*. Any time you execute code that communicates with the repository, it talks to the current session only. Other sessions are unaffected.

Each session is assigned a number by GemBuilder as it is created. Your application can call **GciGetSessionId** to inquire about the number of the current session, or **GciSetSessionId** to make another session the current one. Your application is responsible for treating each session distinctly.

An application can terminate a session by calling **GciLogout**. After that call returns, the current session no longer exists.

Transaction Management

Committing a Transaction

The GemStone repository proceeds from one stable state to the next by continuously committing transactions. In Smalltalk, the message `System commitTransaction` attempts to commit changes to the repository. Similarly, when your C application calls the function **GciCommit**, GemStone will attempt to commit any changes to objects occurring within the current session.

A session within a transaction views the repository as it existed when the transaction started. By the time you are ready to commit a transaction, other

sessions or users may have changed the state of the repository through intervening commit operations. Your application can call **GciAlteredObjs** to determine which objects must be reread from the repository in order to make its view current. Then, to reread those objects, use whatever kind of GemBuilder fetch or traversal functions best suits your needs.

If an attempt to commit fails, your application must call **GciAbort** to discard the transaction. If it does not do so, subsequent calls to **GciCommit** will not succeed.

As mentioned earlier, if your C code has created any new objects or has modified any objects whose representation you have imported, those objects must be exported to the GemStone repository in their new state before the transaction is committed. This ensures that the committed repository properly reflects the intended state.

Aborting a Transaction

By calling **GciAbort**, an application can discard from its current session all the changes to persistent objects that were made since the last successful commit or since the beginning of the session (whichever is later). This has exactly the same effect as sending the Smalltalk message

```
System abortTransaction.
```

After the application aborts a transaction, it must reread any object whose state has changed.

Controlling Transactions Manually

Under automatic transaction control, a transaction is started when a user logs in to the repository. The transaction then continues until it is either committed or aborted. The call to **GciAbort** or **GciCommit** automatically starts a new transaction when it finishes processing the previous one. Thus, the user is always operating within a transaction.

Automatic transaction control is the default control mode in GemStone. However, there is some overhead associated with transactions that an application can avoid by changing the transaction mode to manual:

```
GciExecuteStr(
   "System transactionMode: #manualBegin", OOP_NIL);
```

The transaction mode can also be returned to the automatic default:

```
GciExecuteStr(
   "System transactionMode: #autoBegin", OOP_NIL);
```

In manual mode, the application starts a new transaction manually by calling the **GciBegin** function. The **GciAbort** and **GciCommit** functions complete the current transaction, but do not start a new transaction. Thus, they leave the user session operating outside of a transaction, without its attendant overhead. The session views the repository as it was when the last transaction was completed, or when the mode was last reset, whichever is later.

Since automatic transaction control is the default, a transaction is always started when a user logs in. To operate outside a transaction initially, an application must first set the mode to manual, and then either abort or commit the transaction.

1.3 Representing Objects in C

An important feature of the GemStone data model is its ability to preserve an object's identity distinct from its state. Within GemStone, each object is identified by a unique 32-bit object-oriented pointer, or OOP. Whenever your C program attempts to access or modify the state of a GemStone object, GemStone uses its OOP to identify it. Both the OOP and a representation of the object's state may be imported into an application's C address space.

Within your C program, object identity is represented in variables of type **OopType** (object-oriented pointer). The GemBuilder include file `gci.ht` defines type **OopType**, along with other types used by GemBuilder functions. For more information, see "GemBuilder Include Files" on page 5-10.

GemStone-defined Object Mnemonics

The GemBuilder include file `gcioop.ht` defines C mnemonics for all of the kernel classes in the GemStone repository, as well as the GemStone objects *nil*, *true*, and *false*, and the GemStone error dictionary.

In addition to the predefined objects mentioned above, the GemBuilder include file `gcioop.ht` also defines the C mnemonic *OOP_ILLEGAL*. That mnemonic represents a value that will never be used to represent any object in the repository. You can thus initialize the state of an OOP variable to OOP_ILLEGAL, and test later in your program to see if that variable contains valid information.

NOTE:
Bear in mind that your C program can only use predefined OOPs, or OOPs that it has received from the GemStone. Your C program cannot create new OOPs directly — it must ask GemStone to create new OOPs for it.

Converting Between Special Objects and C Values

Some Smalltalk classes encode their objects' states directly in their OOPs:

- SmallInteger objects (for example, the number 5)
- AbstractCharacter and its subclasses, JISCharacter and Character (for example, the letter 'b')
- Boolean values (true and false)
- Instances of class UndefinedObject (such as *nil*)

The following GemBuilder functions and macros allow conversion between Character, SmallInteger, or Boolean objects and the equivalent C values:

GCI_BOOL_TO_OOP — (MACRO) Convert a C Boolean value to a GemStone Boolean object.

GCI_CHR_TO_OOP — (MACRO) Convert a C character value to a GemStone Character object.

GciLongToOop — Find a GemStone object that corresponds to a C long integer.

GCI_LONG_TO_OOP — (MACRO) Find a GemStone object that corresponds to a C long integer.

GciOopToBool — Convert a Boolean object to a C Boolean value.

GCI_OOP_TO_BOOL — (MACRO) Convert a Boolean object to a C Boolean value.

GciOopToChr — Convert a Character object to a C character value.

GCI_OOP_TO_CHR — (MACRO) Convert a Character object to a C character value.

GciOopToLong — Convert a Gemstone object to a C long integer value.

GCI_OOP_TO_LONG — (MACRO) Convert a GemStone object to a C long integer value.

GciUnsignedLongToOop — Find a GemStone object that corresponds to a C unsigned long integer.

In addition, the following functions allow conversion between Float objects and their equivalent C values. Although a Float's OOP does not encode its state, these functions are listed here for your convenience.

GciFltToOop — Convert a C double value to a Float object.

GciOopToFlt — Convert a Float object to a C double value.

The following macros are for testing OOPs:

GCI_LONG_IS_SMALL_INT — (MACRO) Determine whether or not a long can be translated into a SmallInteger.

GCI_OOP_IS_BOOL — (MACRO) Determine whether or not a GemStone object represents a Boolean value.

GCI_OOP_IS_SMALL_INT — (MACRO) Determine whether or not a GemStone object represents a SmallInteger.

GCI_OOP_IS_SPECIAL — (MACRO) Determine whether or not a GemStone object has a special representation.

The GemBuilder include file `gcioop.ht` uses the C mnemonics OOP_TRUE, OOP_FALSE, and OOP_NIL to represent the GemStone objects *true*, *false*, and *nil*, respectively.

In Example 1.1, assume that you have defined a Smalltalk class called Address that represents a mailing address. If the class has five instance variables, the OOPs of one instance of Address can be imported into a C array called *address*. Example 1.1 assumes that this is done in the “intervening code”. Finally, assume that the fifth instance variable represents the zip code of the address.

The fifth element of *address* is the OOP of the SmallInteger object that represents the zip code, not the zip code itself. Example 1.1 imports the value of the zip code object to the C variable *zip*.

Example 1.1

```
OopType address[5];
long zip;

/* Intervening code goes here, in place of this comment */

zip = GciOopToLong (address[4]);

/* zip now contains a long integer which has the same
 value as the GemStone object represented by address[4] */
```

Byte Swizzling of Binary Floating-Point Values

If an application is running on a different machine than its Gem, the byte ordering of binary floating-point values may differ on the two machines. To ensure the correct interpretation of floating values when they are transferred between such machines, the bytes need to be reordered (*swizzled*) to match the machine to which they are transferred. GemBuilder handles all necessary byte swizzling for an application automatically and transparently.

In GemStone, a binary float is an instance of class Float (eight bytes) or SmallFloat (four bytes). The size of binary float objects is fixed by GemStone and cannot be changed. The programmer must supply all the bytes for a binary floating object when creating or storing it.

The following GemBuilder functions provide automatic byte swizzling for binary floats:

GciClampedTraverseObjs — Traverse an array of objects, subject to clamps.

GciCreateByteObj — Create a new byte-format object.

GciFetchObjInfo — Fetch information and values from an object.

GciMoreTraversal — Continue object traversal, reusing a given buffer.

GciNbClampedTraverseObjs — Traverse an array of objects, subject to clamps (nonblocking).

GciNbMoreTraversal — Continue object traversal, reusing a given buffer (nonblocking).

GciNbStoreTrav — Store multiple traversal buffer values in objects (nonblocking).

GciNbTraverseObjs — Traverse an array of GemStone objects (nonblocking).

GciNewOopUsingObjRep — Create a new GemStone object from an existing object report.

GciPerformTraverse — First send a message to a GemStone object, then traverse the result of the message.

GciStoreBytesInstanceOf — Store multiple bytes in a byte object.

GciStoreTrav — Store multiple traversal buffer values in objects.

GciTraverseObjs — Traverse an array of GemStone objects.

The following GemBuilder functions raise an error if you pass a binary float object to them:

GciAppendBytes — Append bytes to a byte object.

GciStoreByte — Store one byte in a byte object.

GciStoreBytes — Store multiple bytes in a byte object.

GciStoreChars — Store multiple ASCII characters in a byte object.

The **GciFetchBytes** function does not raise an error if you pass a binary float object to it, but it also does not provide automatic byte swizzling. It is intended primarily for use with other kinds of byte objects, such as strings. If you wish to use it with binary floats, you must perform your own byte swizzling as needed.

1.4 Manipulating Objects in GemStone

GemBuilder provides functions that allow C applications to execute Smalltalk code in the repository and to send messages directly to GemStone objects. This section describes these functions in more detail.

Sending Messages to GemStone Objects

GemBuilder provides two functions, **GciSendMsg** and **GciPerform**, that send a message to a GemStone object. When GemStone receives a message, it invokes and executes the method associated with that message. Thus, the code execution occurs in the repository, not in the application.

Example 1.2 illustrates differences in syntax for these functions. However, each statement would have the same effect when executed: to place *someValue* at *someKey* location within *someDict* object.

Example 1.2

```
OopType someDict, someKey, someValue, someArgList[2];
someArgList[0] = someKey;
someArgList[1] = someValue;

/* Intervening code goes here, in place of this comment */

/* Two statements that have the same effect when executed */
oop = GciSendMsg(someDict, 4, "at:", someKey, "put:",
  someValue);
oop = GciPerform(someDict, "at:put:", someArgList, 2);
```

Each function has its own advantages over the other. The **GciSendMsg** syntax lists the GemStone message elements explicitly, which may make your C code more readable (and therefore easier to maintain). On the other hand, **GciPerform** does not need to piece together the message elements for GemStone, and hence it runs somewhat faster. Nevertheless, your application may need to do the same work itself in order to use **GciPerform**, in which case **GciSendMsg** is the better choice, since it does this work automatically and reliably. Where these tradeoffs are important to your application, your choice may well be determined by how much reuse the program can make of the argument list for **GciPerform**.

Executing Code in GemStone

Your C application can execute Smalltalk code by calling any of the following GemBuilder functions:

GciExecute — Execute a Smalltalk expression contained in a String object.

GciExecuteFromContext — Execute a Smalltalk expression contained in a String object as if it were a message sent to another object.

GciExecuteStr — Execute a Smalltalk expression contained in a C string.

GciExecuteStrFromContext — Execute a Smalltalk expression contained in a C string as if it were a message sent to an object.

The GemBuilder function **GciExecuteStr** allows your application to send a C string containing Smalltalk code to GemStone for compilation and execution. The Smalltalk code may be a message expression, a statement, or a series of statements; in sum, any self-contained unit of code that you could execute within a Topaz **PrintIt** command.

GemStone uses the specified symbol list argument to bind any symbols contained in the Smalltalk source. If the symbol list is OOP_NIL, GemStone uses the symbol list associated with the currently logged-in user. Example 1.3 demonstrates the use of this GemBuilder function.

Example 1.3

```
/*
Pass the String to GemStone for compilation and execution.
If it succeeds, return the OOP "objSize" (the size of the
object).
*/
objSize = GciExecuteStr(" ^ myObject size ", OOP_NIL);
```

Your Smalltalk code has the same format as a method, and may include temporaries. In addition, although the circumflex (^) character is used in the above example to return a value after GemStone has executed Smalltalk code (`myObject size`), the circumflex is not required. GemStone returns the result of the last Smalltalk statement executed.

The other functions work similarly, with variations. Before you call **GciExecute** or **GciExecuteFromContext**, you must create or modify a GemStone String object to contain the Smalltalk text to be executed. The **GciExecuteFromContext** and **GciExecuteStrFromContext** functions execute the Smalltalk code within the context (scope) of a specified GemStone object, which implies that the code can access the object's instance variables.

Interrupting GemStone Execution

GemBuilder provides two ways for your application to handle repository interrupts:

- A *soft break* interrupts the Smalltalk virtual machine only. The only GemBuilder functions that can recognize a soft break are **GciSendMsg**, **GciPerform**, **GciContinue**, **GciExecute**, **GciExecuteFromContext**, **GciExecuteStr**, and **GciExecuteStrFromContext**.
- A *hard break* interrupts the Gem process itself, and is not trappable through Smalltalk exceptions.

Issuing a soft break may be desirable if, for example, your application sends a message to an object (via **GciSendMsg** or **GciPerform**), and for some reason the invoked Smalltalk method enters an infinite loop.

In order for GemBuilder functions in your program to recognize interrupts, your program usually needs an interrupt routine that can call the functions **GciSoftBreak** and **GciHardBreak**. Since GemBuilder generally does not relinquish control to an application until it has finished its processing, soft and hard breaks are then initiated from an interrupt service routine. Alternatively, if you are calling the non-blocking GemBuilder functions, you can service interrupts directly within your event loop, while awaiting the completion of a function.

If GemStone is executing when it receives the break, it replies with an error message. If it is not executing, it ignores the break.

Modification of Classes

Some class definitions are more flexible than others. With respect to modification, classes fall into three categories:

kernel classes — Predefined kernel classes cannot be modified. You can, however, create a subclass of a kernel class and redefine your subclass's behavior.

invariant classes — Once a class has been fully developed, it is normally invariant. Class invariance does not imply that it is impervious to all change. You can add or remove methods, method categories, class variables, or pool variables to any class except a predefined kernel class. You can also create instances of an invariant class.

modifiable classes — You can also create specially modifiable classes, a feature that can be useful (for example) while you are defining schema or implementing the classes. You can modify these classes in the same ways as invariant classes, but you can also add or remove named instance variables or change constraints on instance variables. However, you cannot create an instance of a modifiable class. To create an instance, you must first change the class to invariant.

The GemStone Behavior class provides several methods for changing the characteristics of modifiable classes. Use only these predefined methods — *do not use structural access to modify classes.*

1.5 Manipulating Objects Through Structural Access

As mentioned earlier in this chapter, GemBuilder provides a set of C functions that enable you to do the following:

- Import objects from GemStone to your C program
- Create new GemStone objects
- Directly access and modify the internal contents of objects through their C representations
- Export objects from your C program to the GemStone repository

You may need to use GemBuilder's "structural access" functions for either of two reasons:

- Speed

  Because they call on GemStone's internal object manager without using the Smalltalk virtual machine, the structural access functions provide the most efficient possible access to individual objects.

- Generality

  If your C application must handle GemStone objects that it did not create, using the structural access functions may be the only way you can be sure that the components of those objects will be accessible to the application. A user might, for example, define a subclass of Array in which at: and at:put: were disallowed or given new meanings. In that case, your C application could not rely on the standard GemStone kernel class methods to read and manipulate the contents of such a collection.

Despite their advantages, you should use these structural access functions *only* if you've determined that Smalltalk message-passing won't do the job at hand. GemBuilder's structural access functions violate the principles of abstract data types and encapsulation, and they bypass the consistency checks encoded in the Smalltalk kernel class methods. If your C application unwisely alters the structure of a GemStone object (by, for example, storing bytes directly into a floating-point number), the object will behave badly and your application will break.

For the same reason, do not use structural access to change the characteristics of modifiable classes. Use **GciSendMsg** to invoke the Smalltalk methods defined under class Behavior for this specific purpose.

For security reasons, the GemStone object AllUsers cannot be modified using structural access. If you attempt to do so, GemStone raises the RT_ERR_OBJECT_PROTECTED error.

Direct Access to Metadata

Your C program can use GemBuilder's structural access functions to request certain data about an object:

- Class

  Each object is an instance of some class. The class defines the behavior of its instances. To find an object's class, call **GciFetchClass**.

- Format

  GemStone represents the state of an object in one of four different implementations (formats): byte, pointer, NSC (non-sequenceable collection), or special. (These implementations are described in greater detail in the following section.) To find an object's implementation, call GciFetchObjImpl.

- Size

  The function **GciFetchNamedSize** returns the number of named instance variables in an object, while **GciFetchVaryingSize** returns the number of unnamed instance variables in an object. **GciFetchSize** returns the object's complete size (the sum of its named and unnamed variables).

  The result of **GciFetchSize** depends on the object's implementation ("format"). For byte objects (such as instances of String or Float), **GciFetchSize** returns the number of bytes in the object's representation. For pointer and NSC objects, this function returns the number of OOPs that represent the object. For "special" objects (such as *nil*, or instances of SmallInteger, Character, and Boolean), the size is always 0.

Byte Objects

GemStone byte objects (for example, instances of class String or Symbol) can be manipulated in C as arrays of characters. The following GemBuilder functions enable your C program to store into, or fetch from, GemStone byte objects such as Strings:

GciAppendBytes — Append bytes to a byte object.

GciAppendChars — Append a C string to a byte object.

GciFetchByte — Fetch one byte from an indexed byte object.

GciFetchBytes — Fetch multiple bytes from an indexed byte object.

GciFetchChars — Fetch multiple ASCII characters from an indexed byte object.

GciStoreByte — Store one byte in a byte object.

GciStoreBytes — Store multiple bytes in a byte object.

GciStoreChars — Store multiple ASCII characters in a byte object.

Although instances of Float are implemented within GemStone as byte objects, use the functions GciOopToFlt and GciFltToOop to convert between Float objects and their equivalent C values.

Assume that the C variable *suppId* contains an OOP representing an object of class String. Example 1.4 imports that String into the C variable *suppName*:

Example 1.4

```
long                size;
OopType             suppId;
char                suppName[MAXLEN + 1];

/* Intervening code goes here, in place of this comment */

size = GciFetchSize(suppId);
GciFetchBytes (suppId, 1L, suppName, size);
suppName[size] = '\0';
/* suppName now contains the GemStone object referenced
  by suppId */
```

Pointer Objects

In your C program, a GemStone pointer object is represented as an array of OOPs. The order of the OOPs within the GemStone pointer object is preserved in the C array. GemStone represents the following kinds of objects as arrays of OOPs:

Objects with Named Instance Variables

Any object with one or more named instance variables is represented as an array of OOPs. You can determine the positional mapping of instance variables to indexes within the OOP array by calling the GemBuilder function **GciIvNameToIdx**. The following GemBuilder functions allow your C program to store into, or fetch from, GemStone pointer objects with named instance variables:

GciFetchNamedOop — Fetch the OOP of one of an object's named instance variables.

GciFetchNamedOops — Fetch the OOPs of one or more of an object's named instance variables.

GciStoreNamedOop — Store one OOP into an object's named instance variable.

GciStoreNamedOops — Store one or more OOPs into an object's named instance variables.

Indexable Objects

Any indexable object not implemented as a byte object is represented as an array of OOPs. The following GemBuilder functions allow your C program to store into, or fetch from, indexable pointer objects:

GciFetchVaryingOop — Fetch the OOP of one unnamed instance variable from an indexed pointer object or NSC.

GciFetchVaryingOops — Fetch the OOPs of one or more unnamed instance variables from an indexed pointer object or NSC.

GciStoreIdxOop — Store one OOP in a pointer object's unnamed instance variable.

GciStoreIdxOops — Store one or more OOPs in a pointer object's unnamed instance variables.

In each of the following functions, if the indexable object contains named instance variables, pointers to the named instance variables precede pointers to the indexed instance variables.

GciFetchOop — Fetch the OOP of one instance variable of an object.

GciFetchOops — Fetch the OOPs of one or more instance variables of an object.

GciStoreOop — Store one OOP into an object's instance variable.

GciStoreOops — Store one or more OOPs into an object's instance variables.

Assume that the C variable *currSup* contains an OOP representing an object of class Supplier (which defines seven named instance variables). Example 1.5 imports the state of the Supplier object (that is, the OOPs of its component instance variables) into the C variable *instVar*:

Example 1.5

```
OopType currSup;
OopType instVar[7];

/* Intervening code goes here, in place of this comment */

GciFetchNamedOops (currSup, 1L, instVar, 7);
/* instVar now contains the OOPs of the seven instance
  variables of the GemStone object referenced by currSup
*/
```

Non-sequenceable Collections (NSC Objects)

In addition to byte objects and pointer objects, GemStone exports objects implemented as non-sequenceable collections (NSCs). NSC objects (for example, instances of class Bag, Set, and Dictionary) reference other objects in a manner similar to pointer objects, except that the notion of order is not preserved when objects are added to or removed from the collection.

The following GemBuilder functions allow your C program to store into, or fetch from, GemStone NSC objects:

GciAddOopToNsc — Add an OOP to the unordered variables of a non-sequenceable collection.

GciAddOopsToNsc — Add multiple OOPs to the unordered variables of a non-sequenceable collection.

GciFetchOop — Fetch the OOP of one instance variable of an object.

GciFetchOops — Fetch the OOPs of one or more instance variables of an object.

GciRemoveOopFromNsc — Remove an OOP from an NSC.

GciRemoveOopsFromNsc — Remove one or more OOPs from an NSC.

GciReplaceVaryingOops — Replace all unnamed instance variables in an NSC object.

Note that GemStone preserves the position of objects in an NSC only until the NSC is modified, or until the session is terminated (whichever comes first). Although you may use the functions **GciFetchOops** or **GciFetchOop** (defined for pointer objects) to retrieve the OOPs of an NSC's elements, you must use one of the

GciAddOopToNsc functions to modify an NSC. (The **GciStoreOop** functions cannot be used with NSCs.)

Assume that the C variable *mySuppSet* contains an OOP representing an object of class SupplierSet (a large set of Supplier objects). Example 1.6 exports the contents of the C variable *newSupp* (a Supplier object) into that SupplierSet:

Example 1.6

```
OopType mySuppSet;
OopType newSupp;

/* Intervening code goes here, in place of this comment */

GciAddOopToNsc (mySuppSet, newSupp);
/* The GemStone Set referenced by mySuppSet now contains
      the OOP of the object newSupp */
```

1.6 Creating Objects

The following GemBuilder functions allow your C program to create instances of Smalltalk classes:

GciNewOop — Create a new GemStone object.

GciNewOops — Create multiple new GemStone objects.

GciNewOopUsingObjRep — Create a new GemStone object from an existing object report.

Your C application may also create a new object by executing some Smalltalk code that creates new objects as a side-effect.

Once your application has created a new object, it can export the object to the repository by performing the following steps:

Step 1. Modify a previously committed object in the repository so that it references the new object. This may be accomplished with a call to one of the **GciStore...** functions, or by sending a Smalltalk message with the new object as an argument, where the invoked method changes a committed object to reference the new object.

Step 2. Give the new object some meaningful state.

Step 3. Commit a transaction. (As mentioned earlier in this chapter, your C program must first export the object to the GemStone repository before attempting to commit the transaction.)

1.7 Fetching and Storing Objects

Efficient Fetching and Storing with Object Traversal

The functions described in the preceding sections allow your C program to import and export the components of a single GemStone object. When your application needs to obtain information about multiple objects in the repository, it can minimize the number of network calls by using GemBuilder's object traversal functions.

NOTE:
If you are using GciLnk (the "linkable" GemBuilder*), object traversal will be of little benefit to you. For details, see "GciRpc and GciLnk" on page 2-1.*

Suppose, for example, that you had created a GemStone Employee class like the one in Example 1.7:

Example 1.7

```
Object subclass: 'Employee'
      instVarNames: #( 'name' 'empNum' 'jobTitle'
                              'department' 'address'
                              'favoriteTune')
      classVars: #()
      poolDictionaries: #()
      inDictionary: UserGlobals
      constraints: #[ #[#name,   Name],
                              #[#empNum, SmallInteger],
                              #[#jobTitle, String],
                              #[#department, Department],
                              #[#address, Address],
                              #[#favoriteTune, String] ]
      isInvariant: false.
```

Notice the constraints on Employee's instance variables. Department, name, and address are all constrained to contain instances of other complex classes.

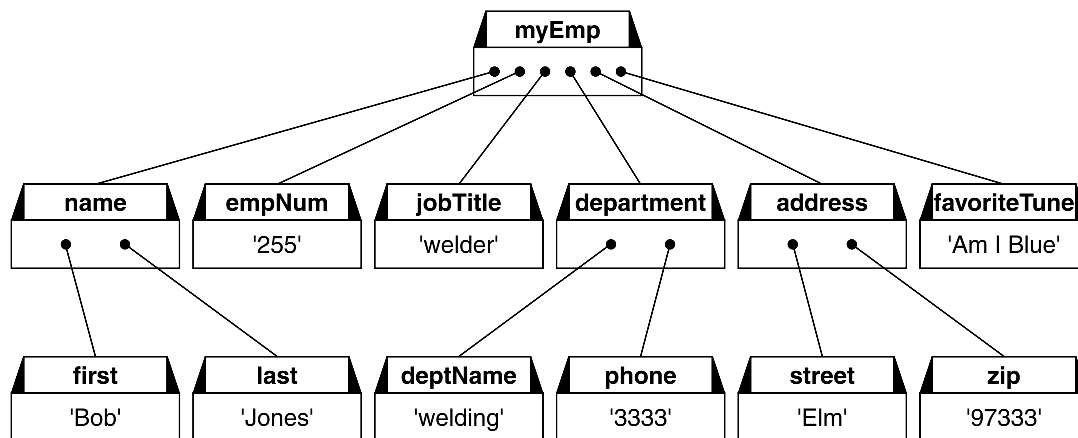
Now imagine that you needed to write C code to make a two-column display of job titles and favorite tunes. By using GemBuilder's "object traversal" functions, you can minimize the number of network fetches and avoid running the Smalltalk virtual machine.

How Object Traversal Works

To understand the object traversal mechanism, think of each GemStone pointer object as the root of a tree (for now, ignore the possibility of objects containing themselves). The branches at the first level go to the object's instance variables, which in turn are connected to their own instance variables, and so on.

Figure 1.2 illustrates a piece of the tree formed by an instance of Employee.

Figure 1.2 Object Traversal and Paths



In a single call, GemStone's internal object traversal function walks such a tree post-depth-first to some specified level, building up a "traversal buffer" that is an array of "object reports" describing the classes of the objects encountered and the values of their contents. It then returns that traversal buffer to your application for selective extraction and processing of the contents.

Thus, to make your list of job titles and favorite tunes with the smallest possible amount of network traffic per employee processed, you could ask GemStone to traverse each employee to two levels (the first level is the Employee object itself

and the second level is that object's instance variables). You could then pick out the object reports describing *jobTitle* and *favoriteTune*, and extract the values stored by those reports (*welder* and *Am I Blue* respectively).

This approach would minimize network traffic to a single round trip.

One further optimization is possible: instead of fetching each employee and traversing it individually to level two, you could ask GemStone to begin traversal at the *collection* of employees and to descend three levels. That way, you would get information about the whole collection of employees with just a single call over the network.

The Object Traversal Functions

The function **GciTraverseObjs** traverses object trees rooted at a collection of one or more GemStone objects, gathering object reports on the specified objects into a traversal buffer.

Each object report provides information about an object's identity (its OOP), class, size (the number of instance variables, named plus unnamed), segment, implementation (byte, pointer, NSC, or special), and the values stored in its instance variables.

When the amount of information obtained in a traversal exceeds the amount of available memory, your application can break the traversal into manageable amounts of information by issuing repeated calls to **GciMoreTraversal**. Generally speaking, an application can continue to call **GciMoreTraversal** until it has obtained all requested information.

Your application can call **GciFindObjRep** to scan a traversal buffer for an individual object report. Before it allocates memory for a copy of the object report, your program can call **GciObjRepSize** to obtain the size of the report.

The function **GciStoreTrav** allows you to store values into any number of existing GemStone objects in a single network round trip. That function takes a traversal buffer of object reports as its argument.

Efficient Fetching And Storing with Path Access

As you've seen, object traversal is a powerful tool for fetching information about multiple objects efficiently. But writing the code for parsing traversal buffers and object reports may not always be simple. And even if you can afford the memory for importing unwanted information, the processing time spent in parsing that information into object reports may be unacceptable.

Consider the Employee object illustrated in the Figure 1.2. If your job were to extract a list of job titles and favorite tunes from a set of such Employees, it would be reasonable to use GemBuilder's object traversal functions (as described above) to get the needed information. The time spent in building up object reports for the unwanted portions would probably be negligible. Suppose, however, that there were an additional 200 instance variables in each Employee. Then the time used in processing wasted object reports would far exceed the time spent in useful work.

Therefore, GemBuilder provides a set of path access functions that can fetch or store multiple objects at selected positions in an object tree with a single call across the network, bringing only the desired information back. The function **GciFetchPaths** lets you fetch selected components from a large set of objects with only a single network round trip. Similarly, your program can call **GciStorePaths** to store new values into disparate locations within a large number of GemStone objects.

Path Representations

There are two ways of representing a path describing the position of an object to be fetched from within an object tree.

In Smalltalk, the path leading to an Employee's zip code would be expressed as `'address.zip'`, and the path leading to his favorite tune would simply be `'favoriteTune'`.

By contrast, the path access functions also use a non-symbolic form of these paths in which each step along the path is represented by an integral offset from the beginning of an object. The path to an Employee's zip code would be represented by an array containing the integers 5 and 2. The first element, 5, is the offset of the address instance variable, and the second element, 2, is the offset of zip within an Address. The path to an Employee's favorite tune would be represented by a one-element array containing the integer 6.

Your program can call the **GciPathToStr** and **GciStrToPath** functions to convert between the two path representations.

1.8 Nonblocking Functions

Under most circumstances, when an application calls a GemBuilder function, the operation that the function specifies is completed before the function returns control to the application. That is, the GemBuilder function *blocks* the application from proceeding until the operation is finished. This effect guarantees a strict sequence of execution.

Nevertheless, in most cases a GemBuilder function calls upon GemStone (that is, the Gem) to perform some work. If the Gem and the application are running in different processes, especially on different machines, blocking implies that only one process can accomplish work at a time. GemBuilder's *nonblocking functions* were designed to take advantage of the opportunity for concurrent execution in separate Gem and application processes.

The results of performing an operation through a blocking function or through its nonblocking twin are always the same. The difference is that the nonblocking function does not wait for the operation to complete before it returns control to the session. Since the results of the operation are probably not ready when a nonblocking function returns, all nonblocking functions but one (**GciNbEnd**) return void.

While a nonblocking operation is in progress an application can do any kind of work that does not require GemBuilder. In fact, it can also call a limited set of GemBuilder functions, listed as follows:

```
GciCallInProgress
GciErr
GciGetSessionId
GciHardBreak
GciNbEnd
GciSetSessionId
GciShutdown
GciSoftBreak
```

If the application first changes sessions, and that session has no nonblocking operation in progress, then the application can call any GemBuilder function, including a nonblocking function. GemBuilder supports one repository request at a time, per session. However, nonblocking functions do not implement threads, meaning that you cannot have multiple concurrent repository requests in progress within a single session. If an application calls any GemBuilder function besides those listed here while a nonblocking operation is in progress in the current session, the error GCI_ERR_OP_IN_PROGRESS is generated.

Once a nonblocking operation is in progress, an application must call **GciNbEnd** at least once to determine the operation's status. Repeated calls are made if necessary, until the operation is complete. When it is complete, **GciNbEnd** hands the application a pointer to the result of the operation, the same value that the corresponding blocking call would have returned directly.

Nonblocking functions are not truly nonblocking if they are called from a linkable GemBuilder session, because the Gem and GemBuilder are part of the same process. However, those functions can still be used in linkable sessions. If they

are, **GciNbEnd** must still be called at least once per nonblocking call, and it always indicates that the operation is complete.

All error handling features are supported while nonblocking functions are used. Errors may be signalled either when the nonblocking function is called or later when **GciNbEnd** is called.

1.9 Operating System Considerations

Like your C application, GemBuilder for C is, in itself, a body of C code. Some aspects of the interface must interact with the surrounding operating system. The purpose of this section is to point out a few places where you must code with caution in order to avoid conflicts.

Interrupt Handling in your GemBuilder Application

Under Unix, it is important that interrupts be enabled when your code calls GemBuilder functions. Disabling interrupts has the effect of disabling much of the error handling within GemBuilder. Because signal handlers usually disable interrupts during their execution, the implication of this rule is that your application's signal handling code should not call GemBuilder functions. Three GemBuilder functions are exceptions to this rule: **GciCallInProgress**, **GciSoftBreak**, and **GciHardBreak** may be called from within an interrupt handler.

When you initialize the GciLnk version of GemBuilder on Unix platforms, **GciInit** establishes its own handler for SIGIO interrupts. The handler is installed only once and never de-installed. **GciInit** handles the following signals, treating them as fatal:

SIGQUIT	SIGILL	SIGABRT	SIGBUS
SIGSYS	SIGTERM	SIGXFSZ	SIGXCPU
SIGSEV	SIGEMT	SIGLOST	

The handler for fatal signals is not called during a spinlocks critical region. **GciInit** handles SIGCHLD and SIGFPE and treats them as fatal, unless they already have a signal handler installed. **GciInit** ignores SIGPIPE, SIGHUP, and SIGDANGER, unless they already have a signal handler installed.

You can establish your own handler for SIGIO if you like, but the application needs to coordinate the Unix signal handlers installed by the application, GemBuilder, and any third-party software compiled into the application.

Observe this precaution: When you call **GciInit** and then call **signal**() to establish your own handler, save the return value of the **signal**() call. This is the address of the GemBuilder SIGIO handler. Then, at any time during the execution of your own SIGIO handler, make sure the GemBuilder handler is invoked. This assures proper and predictable behavior of subsequent GemBuilder calls.

If you install your handler and then call **GciInit**, the GemBuilder handler chains the invocation of your handler with its own.

If all of your application's handlers are installed only once and make an effort to chain to the previously installed handler, the order of their installation should not be significant. If you have third-party signal handlers that do not chain, however, then you should delay the call to **GciInit** until after those handlers are installed. Routinely installing and de-installing a handler is discouraged, because it makes chaining of handlers impossible.

Executing Host File Access Methods

If you use **GciPerform**, **GciSendMsg**, or any of the **GciExecute**... functions to execute a Smalltalk host file access method (as listed below), and you do not supply a full file pathname as part of the method argument, the default directory for the Smalltalk method depends on the version of GemBuilder that you are running. With GciLnk, the default directory is the directory in which the Gem (GemStone session) process was started. With GciRpc, the default directory is the `home` directory of the host user account, or the `#dir` specification of the network resource string. The Smalltalk methods that are affected include `System (C) | performOnServer:` and the file accessing methods implemented in `GsFile`. See the file I/O information in the *GemStone Programming Guide.*

1.10 Error Handling and Recovery

Your C program is responsible for processing any errors generated by GemBuilder function calls. GemStone errors and message creation information are handled by a GemStone language dictionary known as its *error category.* The OOP of this dictionary is available as a mnemonic, `OOP_GEMSTONE_ERROR_CAT` , that is defined in GemBuilder include file `gcioop.ht`.

The GemBuilder include file `gcierr.ht` documents and defines mnemonics for all GemStone errors. Search the file for the mnemonic name or error number to locate an error in the file. The errors are divided into five groups: compiler, run-time (virtual machine), aborting, fatal, and event.

GemBuilder provides functions that allow you to poll for errors or to use error jump buffers. The following paragraphs describe both of these techniques.

Polling for Errors

Each call to GemBuilder can potentially fail for a number of reasons. Your program can call **GciErr** to determine whether the previous GemBuilder call resulted in an error. If so, **GciErr** will obtain full information about the error. If an error occurs while Smalltalk code is executing (in response to **GciPerform**, **GciSendMsg**, or one of the **GciExecute...** functions), your program may be able to continue Smalltalk execution by calling **GciContinue**.

Error Jump Buffers

When your program makes 3 or more GemBuilder calls in sequence, jump buffers provide significantly faster performance than polling for errors.

When your C program calls **setjmp**, the context of the current C environment is saved in a jump buffer designated by your program. Just as your C program can take advantage of the C runtime library's **setjmp**/**longjmp** error-handling mechanism, it can call **GciPushErrJump** to push the jump buffer onto an error jump stack maintained by GemBuilder. The corresponding function **GciPopErrJump** pops buffers from the stack.

When an error occurs during a GemBuilder call, the GemBuilder function causes a **longjmp** to the buffer currently at the top of GemBuilder's error jump stack, and pops that buffer from the stack. At that time, the previous environment is restored.

For functions with local error recovery, your program can call **GciSetErrJump** to temporarily disable the error handling mechanism (and to re-enable error handling afterwards).

Whenever the jump stack is empty, the application must use **GciErr** to poll for any GemBuilder errors.

The **setjmp** and **longjmp** functions are described in your C runtime library documentation.

GciPushErrHandler also pushes a jump buffer onto the stack of GemBuilder jump buffers. The block of code to which a longjmp on this jump buffer will go must call **GciHandleError** instead of **GciErr**. Its jump buffer must be of type `GCI_SIG_JMP_BUF_TYPE` , and it must be used with the **GCI_SETJMP** and **GCI_LONGJMP** macros instead of **setjmp** and **longjmp**.

For linkable GemBuilder applications, the combination of **GciHandleError** and **GciPushErrHandler** offer performance gains over **GciErr** and **GciPushErrJump**. (See page 5-125 for an example of this function's use.)

To determine whether the previous GemBuilder function call resulted in an error, your application program can call **GciHandleError**. If an error has occurred, this function provides information about the error and about the state of the GemStone system. The `gcierr.ht` include file lists the various errors that may be returned by GemBuilder functions.

WARNING:
If you use a long jump call, especially in an interrupt handler, you should be sure that you do not jump around GemBuilder calls; stay within your own code. GemBuilder functions, once entered, must be allowed to complete. Otherwise, GemBuilder could be left in an inconsistent state that can cause your application to fail.

The Call Stack

The Smalltalk virtual machine creates and maintains a *call stack* that provides information about the state of execution of the current Smalltalk expression or sequence of expressions. The call stack includes an ordered list of activation records related to the methods and blocks that are currently being executed. The virtual machine ordinarily clears the call stack before each new expression is executed.

If a soft break or an unexpected error occurs, the virtual machine suspends execution, creates a Process object, and raises an error. The Process object represents both the Smalltalk call stack when execution was suspended and any information that the virtual machine needs to resume execution. If there was no fatal error, your program can call **GciContinue** to resume execution. Call **GciClearStack** instead if there was a fatal error, or if you do not want your program to resume the suspended execution.

GemStone System Errors

If your application receives a GemStone system error while linked with GciLnk, relink your application with GciRpc and run it again with an uncorrupted copy of your repository. Your GemStone system administrator can refer to the repository backup and recovery procedures in the *GemStone System Administration Guide*.

If the error can be reproduced, contact GemStone Customer Support. Otherwise, the error is in your application, and you need to debug your application before using GciLnk again.

1.11 Garbage Collection

GemStone performs automatic garbage collection for all GemStone objects, whether persistent or not. You can find a full discussion of removal of persistent objects in the chapter "Reclaiming Storage" in the *GemStone Programming Guide*.

Before removing any objects, the GemStone garbage collector checks the *export set* in the user session's workspace. Any object in this set is considered to be marked for *saving*. In this context, saving does not mean "writing to disk". Instead, objects are saved from garbage collection and retained for use by the application.

The garbage collector does not remove objects that are in the export set, or objects that are referenced by a persistent object. It also does not remove any additional objects that they refer to, or more objects that those additional objects refer to, and so on.[1] The export set is used to avoid having an unreferenced object disappear from your session before you can create your own reference to it.

If left unguarded, new or temporary objects could be vulnerable to garbage collection, since no other object might reference them. On the other hand, persistent objects must have a reference already when you import them into an application.

A C application can create new objects by using any of the **GciNew...** or **GciCreate...** functions, or by calling the **GciStoreTrav** function with the GCI_STORE_TRAV_CREATE flag. New objects may also be returned from calls to the **GciErr** and **GciHandleError** functions. The application can also send messages to existing GemStone objects and execute code in GemStone by using any of the **GciSendMsg**, **GciPerform...**, or **GciExecute...** functions.

All of these functions return their results to the C application in the form of one or more OOPs (objects), through either return values or output parameters. To protect these result objects from premature garbage collection, GemBuilder *automatically* adds all of them to the export set. However, it adds them only when the application calls the functions; it does not add them when a user action calls the functions.

GemBuilder does not automatically add other objects to the export set. It specifically does not add objects that it imports from the repository. The danger of premature removal (by other sessions) of an object that is already persistent is

1. Mathematically speaking, the set of saved objects is the transitive closure over references of the union of the export set with all objects that are referenced by at least one persistent object.

likely to be very small. And if the set becomes larger than is really necessary, garbage tends to accumulate in the repository, and GemStone slows down.

By default, GemBuilder thus strikes a reasonable balance between the need for performance and the preservation of objects needed by an application. The application can improve performance further by calling the **GciRelease...** functions at appropriate times, to reduce the export set's size and permit garbage collection of obsolete temporaries. The application should also be careful to call the **GciSaveObjs** function when it needs to be sure to retain an object that is not already in the export set.

1.12 Preparing to Execute GemStone Applications

The following information includes the requirements and recommendations for preparing your environment to execute C applications for GemStone. Your application may have additional requirements, such as environment variables that it uses.

Kerberos

Kerberos provides user authentication for services such as process creation. Both Kerberos and the GemStone internal interface to it are part of every GemStone shared library. If you are using Kerberos with your application, it is not necessary to include any Kerberos header files in the application or to take any special action when compiling or linking. Your application can use Kerberos for user authentication through a network resource string and a netldi, as explained in the *GemStone System Administration Guide*.

GemStone Environment Variables

Anyone who runs a GemStone application or process is responsible for setting the following environment variables:

GEMSTONE — A full pathname to your GemStone installation directory.

PATH — Add the GemStone `bin` directory to your path.

The following environment variables influence the behavior of GemStone and GemBuilder. You may wish to supply values or defaults for them when you or your users run your application or a Gem.

GEMSTONE_EXE_CONF — (not for RPC applications) A full path to a special GemStone configuration file for an executable, if any. See the *GemStone System Administration Guide* for details.

GEMSTONE_SYS_CONF — (not for RPC applications) A full path to a special GemStone configuration file for your system, if any. See the *GemStone System Administration Guide* for details.

GEMSTONE_NRS_ALL — A network resource string — a means for identifying certain GemStone file and process information. It can identify Kerberos as the means by which to perform user authentication, and the name of the script to run to start an RPC Gem. See the *GemStone System Administration Guide* for details.

GEMSTONE_LANG — A means of identifying the language to use for messages. Under Unix, it is a full pathname to a file containing message text. Under Windows NT, it is the name of the language, such as “german” or “spanish”. See the *GemStone System Administration Guide* for more details.

GCIUSER_DLL — (Windows NT only) A name or path that identifies a DLL file containing user actions.

Chapter 2

Building Applications With GemBuilder for C

This chapter explains how to use GemBuilder to build your C application. Two versions of GemBuilder for C are available to you: GciLnk (the linked version) and GciRpc (the RPC version).

2.1 GciRpc and GciLnk

With GciRpc, your application exists in a process separate from the Gem. The two processes communicate through remote procedure calls. With GciLnk, your application and default Gem (the GemStone session) exist as a single process. Your application is expected to provide the main entry point. You can also run RPC Gems when you use GciLnk.

With GciRpc, because networking software is used for the remote procedure call to the Gem process, there's a fixed overhead (many milliseconds) associated with each GemBuilder call, independent of whatever object access is performed or Smalltalk code is executed.

The function **GciIsRemote** reports whether your application was linked with GciRpc — the "remote procedure call" version of GemBuilder — or GciLnk. The following paragraphs explain some of the differences between these two versions of GemBuilder.

Use GciRpc for Debugging

When debugging a new application, you must use GciRpc. You should use GciLnk *only* after your application has been properly debugged.

When using an RPC Gem, you usually achieve the best performance by using functions such as **GciTraverseObjs**, **GciStoreTrav**, and **GciFetchPaths**. Those functions are designed to reduce the number of network round-trips through remote procedure calls.

Use GciLnk for Performance

You can use the linked, single-Gem configuration to enhance performance significantly. With GciLnk, a GemBuilder function call is a machine-instruction procedure call (with overhead measured in microseconds) rather than a remote call over the network to a different process.

WARNING!

Before using GciLnk, debug your C code in a process that does not include a Gem! *For more information, see section "Risk of Database Corruption" on page 4-4.*

With GciLnk, you usually achieve the best performance by using the simple **GciFetch**... and **GciStore**... functions instead of the complex object traversal functions. This makes the application easier to write.

However, you can also run RPC Gems under GciLnk, when you login to GemStone multiple times. The complex traversal functions should perform better in those sessions.

Multiple GemStone Sessions

If your application will be running multiple GemStone sessions simultaneously, or if you will need to run your application and the GemStone session on separate machines, then you will need to use either the GciRpc (remote procedure call) version of GemBuilder, or a non-default login session from GciLnk.

2.2 The GemBuilder Shared Libraries

The two versions of GemBuilder are provided as a set of shared libraries. A *shared library* is a collection of object modules that can be bound to an executable, usually at run time. On Windows NT, shared libraries are known as *dynamically linked*

libraries (DLLs). The contents of a shared library are not copied into the executable. Instead, the library's main function loads all of its functions. Only one copy is loaded into memory, even if multiple client processes use the library at the same time. Thus, they "share" the library.

On Unix the GemBuilder library files `gcilnk50.*` ands `gcirpc50.*` reside in `$GEMSTONE/lib`. On Windows NT, the link files `gcilnk.lib` and `gcirpc.lib` are in `$GEMSTONE/lib`, but the DLL files themselves, `gcilw50.dll`, `gcirw50.dll`, and `gsw50.dll`, are in `$GEMSTONE/bin`.

2.3 Binding to GemBuilder at Run Time

Shared libraries are generally bound to their application at run time. The binding is done by code that is part of the application. If that code is not executed, the shared library is not loaded. With this type of binding, applications can decide at run time which GemBuilder library to use. They can also unbind at run time and rebind to the same or different shared libraries. The code is free to handle a run-time-bind error however it sees fit.

Building the Application

To build an application that run-time-binds to GemBuilder:

1. Include `gcirtl.hf` (not `gci.hf`) in the C source code.

   However, applications are free to use their own run-time-bind interface instead of `gcirtl`, which is meant to be used from C. For example, a Smalltalk application would use the mechanism provided by the Smalltalk vendor to call a shared library.

2. Call `GciRtlLoad(useRpc, ...)` to load the RPC GemBuilder (if `useRpc`) or the linked GemBuilder (if not `useRpc`).

   Call `GciRtlLoad` before any other GemBuilder calls. Call `GciRtlUnload` to unload the current version of GemBuilder.

3. Link with `gcirtlobj.o`, not one of the GemBuilder libraries (`libgcirpc50.*` and `libgcilnk50.*`).

   Chapter 4, "The Mechanics of Compiling and Linking," tells how to compile and link your application.

Searching for the Library

At run time the `gcirtl` code searches for the GemBuilder library in the following places:

1. Any directories specified by the application with **GciRtlLoad**.
2. The `$GEMSTONE/lib` directory.
3. The normal operating system search, as described in the following sections.

How UNIX Matches Search Names with Shared Library Files

The UNIX operating system loader searches the following directories for matching file names, in this order:

1. Any path specified by an environment variable:

   LD_LIBRARY_PATH Solaris
   LIBPATH AIX
   SHLIB_PATH HP-UX

   On HP-UX, the executable must have a bit set or it will not use SHLIB_PATH. Use the **chatr** command to set this bit on an existing executable.

2. Any path recorded in the executable when it was built.
3. The global directory `/usr/lib`. HP-UX also has `/usr/local/lib` and `/usr/contrib/lib`, but the executable must be built in a certain way for it to search these directories. See your HP-UX documentation.

How Windows NT Matches Search Names with DLL Files

Windows NT treats a search name as a path to the DLL file it is looking for. However, the name as given may be an incomplete path, and Windows NT must complete it to find the file.

If the search name already has a file extension (even if it is a null extension, "."), then Windows NT retains the extension. Otherwise, it appends ".dll" as the file extension for the search name. For example, "*pgm*" has no extension and defaults to "*pgm.dll*", while "*pgm.*" has a null extension and is used as given.

If the search name (with extension) is a full path, Windows NT looks for a file at that full path. Otherwise, it searches a predetermined list of directories for a file of the given file name (with extension). The search succeeds with the first directory that has a file whose pathname matches the search name. It fails if no directory has a match.

Windows NT searches the following directories for matching file names, in this order:

1. The directory from which the current executable was loaded (where the executable file for the current process is located).
2. The current directory of the current process.
3. The 32-bit Windows system directory, `SYSTEM32`.
4. The 16-bit Windows system directory, `SYSTEM`.
5. The Windows installation directory.
6. The directories that are listed in the PATH environment variable, in order.

The three Windows directories are rarely (if ever) used for anything relating to GemStone. For more information on locating and loading DLLs, consult your Windows NT documentation for the LoadLibrary C function.

Build-time Binding

Selecting the GemBuilder library at run time, using the `gcirtl` interface, is the preferred behavior for GemBuilder applications. It is possible, however, to bind shared libraries to your application at build time.

In build-time binding, the linker is told what shared library is to be used by the target that is being built. The linker records information about the shared library in the target. When the target is loaded by the operating system, it automatically attempts to load the shared library that was build-time-bound.

The operating system generates an error if it cannot load shared libraries that have been build-time-bound. You must specify which GemBuilder library to use, so the application must commit to using either the RPC or LNK library.

To build an application that build-time-binds to GemBuilder:

1. Include `gci.hf` in the C source code.
2. Link with the `-lgcilnk50` or `-lgcirpc50` flag, to specify one of the GemBuilder libraries.

See Chapter 4, “The Mechanics of Compiling and Linking,” for information on linking command lines.

GemBuilder finds build-time-bound shared libraries by using the normal operating system searches described in “Searching for the Library” on page 2-4. If a library cannot be found, the operating system returns an error.

Chapter

3

Writing C Functions to be Called from GemStone

For certain operations, you may choose to write a C function rather than to perform the work in GemStone. For example, operations that are computationally intensive or are external to GemStone can be written as C functions and called from within a Smalltalk method (whose high-level structure and control is written in Smalltalk). This approach is similar to the concept of "user-defined primitives" offered by some other object-oriented systems.

This chapter describes how to implement C user action functions that can be called from GemStone, and how to call those functions from a GemBuilder application or a Gem (GemStone session) process.

3.1 Shared User Action Libraries

Although user actions can be linked directly into an application, they are usually placed in shared libraries so they can be loaded dynamically. The contents of a library are not copied into the executable. Instead, the library's main function loads all of its user actions. Only one copy is loaded into memory, even if multiple client processes use the library at the same time. See Chapter 2, "Building Applications With GemBuilder for C," for more information.

User action libraries are used in two ways: They can be *application user actions*, which are loaded by the application process, or *session user actions*, which are loaded by the session process. The operation that is used to load the library determines which type it is, not any quality of the library itself. Application and Gem executables can load any library.

Application user actions are the traditional GemStone user actions. They are used by the application for communication with the Gem or for an interactive interface to the user.

Session user actions add new functionality to the Gem, something like the traditional custom Gem. The difference here is that you only need one Gem, which can customize itself at run time. It loads the appropriate libraries for the code it is running. The decisions are made automatically within GemStone Smalltalk, rather than requiring the users to decide what Gem they need before they start their session.

3.2 How User Actions Work

Here's a quick overview of the sequence of events when a user action function is executed:

1. The Gem or your C application program initiates GemStone Smalltalk execution by calling one of the following functions: **GciExecute**, **GciExecuteStr**, **GciExecuteStrFromContext**, **GciSendMsg**, **GciPerform**, or **GciContinue**.
2. Your GemStone Smalltalk code invokes a user action function (written in C) by sending a message of the form:
   `System userAction:` *aSymbol args*

   The *args* arguments are passed to the C user action function named *aSymbol*. (You must have already initialized that function before logging in to GemStone. See "Loading User Actions" on page 3-6.)
3. The C user action function can call any GemBuilder functions and any C functions provided in the application or the libraries loaded by the application (for application user actions), or provided in the libraries loaded by the Gem (for session user actions).

   Specifically, the C user action function can call GemBuilder's structural access functions (**GciFetch**... and **GciStore**..., etc.) to read or modify, respectively, any objects that were passed as arguments to the user action.

If a GemBuilder or other GemStone error is encountered during execution of the user action, control is returned to the Gem or your GemBuilder application as if the error had occurred during the call to **GciExecute** (or whichever GemBuilder function executed the GemStone Smalltalk code in step 1).

4. The C user action function must return an **OopType** as the function result, and must return control directly to the Smalltalk method from which it was called.

NOTE:
Results are unpredictable if the C function uses **GCI_LONGJMP** *instead of returning control to the GemStone Smalltalk virtual machine.*

3.3 Developing User Actions

For your GemStone application to take advantage of user action functions, you do the following:

Step 1. Determine which operations to perform in C user action functions rather than in Smalltalk. Then write the user action functions.

Step 2. Create a user action library to package the functions.

Step 3. Provide the code to load the user action library.

- If the application is to load the library, add the loading code to your application.
- If the session is to load the library, use the GemStone Smalltalk method `System | loadUserActionLibrary:` for loading.

Step 4. Write the Smalltalk code that calls your user action. Commit it to your GemStone repository.

Step 5. Debug your user action.

The following sections describe each of these steps.

Write the User Action Functions

Writing a C function to be installed as a user action that can be called from Smalltalk is generally no different than writing any other C function. The only special guideline is that your C application should treat all argument and result objects of a user action as temporary objects. That is, OOPs should not be saved in static C variables for use by a subsequent invocation of the user action or by another C function, unless you save them explicitly by calling **GciSaveObjs**.

To make a newly created result object a permanent part of the GemStone repository, the user action should store the OOP of the new object into one of the argument objects known to be permanent, or return the OOP of that result object as the function result. After the user action returns, the permanence of the new object is determined by normal Smalltalk object semantics.

Create a User Action Library

Whether you have one user action or many, the way in which you prepare and package the source code for execution has significant effects upon what uses you can make of user actions at run time. It is important to visualize your intended execution configurations as you design the way in which you package your user actions.

To build a user action library:

1. Include `gciua.hf` in your C source code.
2. Define the initialization and shutdown functions.
3. Compile with shared library switches.
4. Link with `gciualib.o` and shared library switches.
5. Install the library in the `$GEMSTONE/ualib` directory.

The `gciua.hf` Header File

User action libraries must always include the `gciua.hf` file, rather than the `gci.hf` file. Using the wrong file causes unpredictable results.

The Initialization and Shutdown Functions

A user action library must define the initialization function **GciUserActionInit** and the shutdown function **GciUserActionShutdown**.

Defining the Initialization Function

Example 3.1 shows how the initialization function **GciUserActionInit** is defined, using the macro **GCIUSER_ACTION_INIT_DEF**. This macro must call the **GCI_DECLARE_ACTION** macro once for each function in the set of user actions.

Example 3.1

```
GCIUSER_ACTION_INIT_DEF()
{
 GCI_DECLARE_ACTION("doLogin", doLogin, 1);
 GCI_DECLARE_ACTION("doLogout", doLogout, 1);
             .
             .
}
```

The **GCI_DECLARE_ACTION** macro associates the Smalltalk name of the user action function *userActionName* (a C string) with the C address of that function, *userActionFunction*, and declares the number or arguments that the function takes. A call to the macro looks similar to this:

`GCI_DECLARE_ACTION("` *userActionName*`",` *userActionFunction*`, 1)`

The macro expands to a block of C statements that install the user action into a table of such functions that GemBuilder maintains. Once a user action is installed, it can be called from GemStone.

The name of the user action, "*userActionName*", is a case-sensitive, null-terminated string that corresponds to the symbolic name by which the function is called from Smalltalk. The name is significant to 31 characters. It is recommended that the name of the user action be the same as the C source code name for the function, *userActionFunction*.

The last argument to the **GCI_DECLARE_ACTION** macro indicates how many arguments the C function accepts. This value should correspond to the number of arguments specified in the Smalltalk message. When it is 0, the function argument is void. Similarly, a value of 1 means one argument. The maximum number of arguments is 8. Each argument is of type **OopType**.

Your user action library may call **GCI_DECLARE_ACTION** repeatedly to install multiple C functions. Each invocation of **GCI_DECLARE_ACTION** must specify a unique *userActionName*. However, the same *userActionFunction* argument may be used in multiple calls to **GCI_DECLARE_ACTION**.

Defining the Shutdown Function

The shutdown function **GciUserActionShutdown** is defined by the **GCIUSER_ACTION_SHUTDOWN_DEF** macro. **GciUserActionShutdown** is called when the user action library is unloaded. It is provided so the user action library can clean up any system resources it has allocated. Do not make

GemBuilder C calls from this function, because the session may no longer exist. In fact, **GciUserActionShutdown** can be left empty. Example 3.2 shows a shutdown definition that does nothing but report that it has been called.

Example 3.2

```
GCIUSER_ACTION_SHUTDOWN_DEF()
{
 /* Nothing needs to be done. */
 fprintf(stderr, "GciUserActionShutdown called.\n");
}
```

Compiling and Linking Shared Libraries

Shared user actions are compiled for and linked into a shared library. See Chapter 4, “The Mechanics of Compiling and Linking,” for instructions.

Be sure to check the output from your link program carefully. Linking with shared libraries does not require that all entry points be resolved at link time. Those that are outside of each shared library await resolution until application execution time, or even until function invocation time. You may not find out about incorrect external references until run time.

Using Existing User Actions in a User Action Library

With slight modifications, existing user action code can be used in a user action library. You need to include `gciua.hf` instead of `gci.hf`. Define a **GciUserActionShutdown**, and a **GciUserActionInit**, if it is not already present. Compile, link, and install according to the instructions for user action libraries.

Using Third-party C Code With a User Action Library

Third-party C code has to reside in the same process as the C user action code. Link the third-party code into the user action library itself, and then you can call that code. It doesn’t matter where you call it from.

Loading User Actions

GemBuilder does not support the loading of any default user action library. Applications and Gems must include code that specifically loads the libraries they require.

Loading User Action Libraries At Run Time

Dynamic run-time loading of user action libraries requires some planning to avoid name conflicts. If an executable tries to load a library with the same name as a library that has already been loaded, the operation fails.

When user actions are installed in a process, they are given a name by which GemBuilder refers to them. These names must be unique. If a user action that was already loaded has the same name as one of the user actions in the library the executable is attempting to load, the load operation fails. On the other hand, if the two libraries contain functions with the same implementation but different names, the operation succeeds.

NOTE:
For backward compatibility, an exception to this behavior is made for name conflicts between application user actions and static user actions in custom Gems. The application is allowed to load its user action library, but the static Gem user action is always used. The application user action with the conflicting name is ignored.

Application User Actions

If the application is to load a user action library, implement an application feature to load it. The GemStone interfaces provide a way to load user actions from your application.

- GemBuilder for C applications: the **GciLoadUserActionLibrary** call
- Smalltalk applications using GemBuilder for Smalltalk:

  `GBSM loadUserActionLibrary:` *ualib*
- Topaz applications: the **loadua** command

The application must load application user actions after it initializes GemBuilder (**GciInit**) and before the user logs into GemStone (**GciLogin**). If the application attempts to install user actions after logging in, an error is returned.

Session User Actions

A linked or RPC Gem process can install and execute its own user action libraries. To cause the Gem to do this, use the `System | loadUserActionLibrary:` method in your GemStone Smalltalk application code. A session user action library stays loaded until the session logs out.

The session must load its user actions after the user logs into GemStone (**GciLogin**). At that time, any application user actions are already loaded. If a

session tries to load a library that the application has already defined, it gets an error. The loading code can be written to handle the error appropriately. Two sessions can load the same user action library without conflict.

Specifying the User Action Library

When writing scripts or committing to the database, you can specify the user action library as a full path or a simple base name. Always use the base name when you need portability. The code that GemBuilder uses to load a user action library expands the base name *ua* to a valid shared library name for the current platform:

- Solaris (Sun): `lib`*ua*`.so`
- AIX (IBM): `lib`*ua*`.o`
- HP-UX (HP): `lib`*ua*`.sl`
- Windows NT: *ua*`.dll`

and searches for the file in the following places in the specified order:

1. The current directory of the application or Gem.
2. The directory the executable is in, if it can be determined.
3. The `$GEMSTONE/ualib` directory.
4. The normal operating system search, as described in “Searching for the Library” on page 2-4.

Creating User Actions in Your C Application

Loading user action libraries at run time is the preferred behavior for GemBuilder applications. For application user actions, however, you have the option to create the user actions directly in your C application, not as part of a library. When you implement user actions this way, include `gcirtl.hf` or `gci.hf` in your C source code, instead of `gciua.hf`. The **GciUserActionInit** and **GciUserActionShutdown** functions are not required, but the application must call the **GCI_DECLARE_ACTION** macro once for each function in the set of user actions.

After your application has successfully logged in to GemStone (via **GciLogin**), it may not call **GCI_DECLARE_ACTION**. If your application attempts to install user actions after logging in, an error will be returned.

Verify That Required User Actions Have Been Installed

After logging in to GemStone, your application can test for the presence of specific user actions by sending the following Smalltalk message:

```
System hasUserAction: aSymbol
```

This method returns *true* if your C application has loaded the user action named *aSymbol*, *false* otherwise.

For a list of all the currently available user actions, send this message:

```
System userActionReport
```

Write the Code That Calls Your User Actions

Once your application or Gem has a way to access the user action library, your GemStone Smalltalk code invokes a user action function by sending a message to the GemStone system. The message can take one of the following forms:

```
System userAction: aSymbol
System userAction: aSymbol with: arg1 [with: arg2] ...
System userAction: aSymbol withArgs: anArrayOfUpTo8Args
```

You can use the *with* keyword from zero to seven times in a message. The *aSymbol* argument is the name of the user action function, significant to 31 characters. Each method returns the function result.

Notice that these methods allow you to pass up to eight arguments to the C user action function. If you need to pass more than eight objects to a user action, you can create a Collection (for example, an instance of Array), store the objects into the Collection, and then pass the Collection as a single argument object to the C user action function:

```
| myArray |
myArray := Array new: 10.

"populate myArray, then send the following message"

System userAction: #doSomething with: myArray.
```

NOTE:
You can also call a user action function directly from your C code, as you would any other C funtion.

Remote User Actions

The user action code that is called can be remote (on a different machine) from the Gem that invokes this method.

Limit on Circular Calls Among User Actions and Smalltalk

From Smalltalk you can invoke a user action, and within the user action you can do a **GciSend**, **GciPerform**, or **GciExecute**, that may in turn invoke another user action. This kind of circular function calling is limited in that no more than 47 user actions may be active at any one time on the current Smalltalk stack. If the limit is exceeded, GemStone raises an error.

Debug the User Action

Even if you intend to use your library only as session user actions, test them first as application user actions with an RPC Gem. As with applications, never debug user actions with linked versions.

WARNING!
Debug your C code in a process that does not include a Gem!
For more information, see section "Risk of Database Corruption" on page 4-4.

Use the instructions for user actions in Chapter 4, "The Mechanics of Compiling and Linking," to compile and link the user action library. Then load the user actions from the RPC version of your application or Topaz. To load from Topaz, use the **loadua** command.

3.4 Executing User Actions

User actions can be executed either in the GemBuilder application (client) process or in a Gem (server) process, or in both.

Choosing Between Session and Application User Actions

The distinction between application user actions that execute in the application and session user actions that execute in the Gem is interesting primarily when the two processes are running remotely, or when the application has more than one Gem process.

Remote Application and Gem Processes

When the application and Gem run on different machines and the Gem calls an application user action, the call is made over the network. Computation is done by the application where the application user action is running, and the result is returned across the network. Using a session user action eliminates this network traffic.

On the other hand, for overall efficiency you also need to consider which machine is more suitable for execution of the user action. For example, assume that your application acquires data from somewhere and wishes to store it in GemStone. You could write a user action to create GemStone objects from the data and then store the objects. It might make more sense to execute the user action in the application process rather than transport the raw data to the Gem.

Alternatively, assume there is a GemStone object that could require processing before the application could use it, like a matrix on which you need to perform a Fast Fourier Transform (FFT). If the Gem runs on a more powerful machine than the client, you may wish to run an FFT user action in the Gem process and send the result to your application.

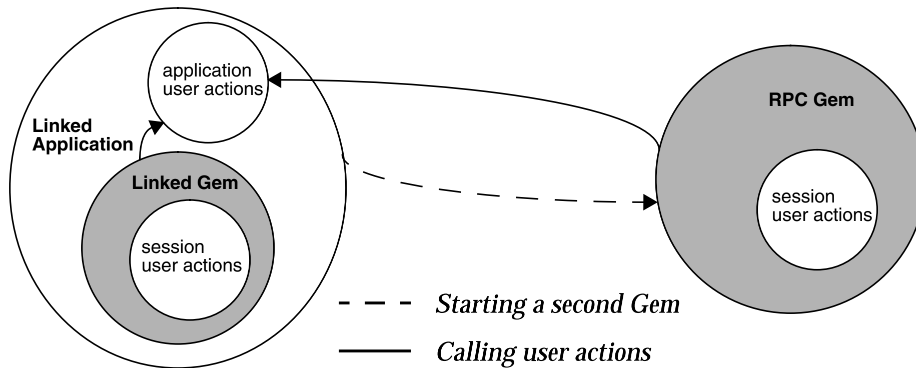
Applications With Multiple Gems

In most situations, session user actions are preferable, because the Gem does not have to make calls to the application. In the case of a linked application, however, an application user action is just as efficient for the linked Gem, because the Gem and application run as one process. Using an application user action guarantees that if any new sessions are created, they will have access to the same user action functions as the first session.

Every Gem can access its own session user actions and the application user actions loaded by its application. A Gem cannot access another Gem's session user actions, however, even when the Gems belong to the same application.

Although a linked application and its first Gem run in the same process, that process can have session and application user actions, as in Figure 3.1. Application user actions, loaded by the application's loading function, are accessible to all the Gems. Session user actions in the same process, loaded by the `System | loadUserActionLibrary:` method, are not accessible to the RPC Gem. Conversely, the RPC Gem's user actions are not accessible to the linked Gem.

Figure 3.1 Access to Application and Session User Actions



The following sections discuss the various possible configurations in detail.

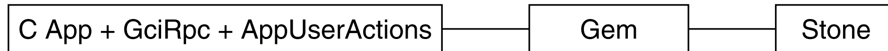
Running User Actions with Applications

User actions can be executed in the user application process under two configurations of GemStone processes. The configurations differ depending upon whether the application is linked or RPC.

With an RPC Application

Figure 3.2 illustrates how various architectural components are distributed among three GemStone processes when a set of user actions executes with an RPC application.

Figure 3.2 Application User Actions and RPC Applications in GemStone Processes



In this configuration, the application runs in a separate process from any Gem. Each time the application calls a GemBuilder C function, the function uses remote procedure calls to communicate with a Gem. The remote procedure calls are used whether the Gem is running on the same machine as the application, or on another machine across the network.

The user actions run in the same process as the application. If they call GemBuilder functions, those functions also use remote procedure calls to communicate with the Gem.

In this configuration, all your code executes as a GemStone client (on the application side). It can thus execute on any GemStone client platform; it is not restricted to GemStone server platforms. Care should be taken in coding to minimize remote procedure call overhead and to avoid excessive transportation of GemStone data across the network. The following list enumerates some of the conditions in which you may find occasion to use this configuration:

- The application and/or the user action needs to be debugged or tested.
- The user action depends on facilities or implement capabilities for the application environment. Screen managament, GUI operations, and control of specialized hardware are possibilities.
- The application acquires data from somewhere and wishes to store it in GemStone. The user action creates the requisite GemStone objects from the data and then commits them to the repository.

NOTE:
*You can run RPC Topazas the C application in this configuration for debugging to perform unit testing of user action libraries. Apply a source-level debugger to the Topaz executable, load the libraries with the Topaz **loadua** command, then call the user actions directly from GemStone Smalltalk.*

With a Linked Application

Figure 3.3 illustrates how various architectural components are distributed between two GemStone processes when a set of user actions executes with a linked application.

Figure 3.3 Session User Actions and Linked Applications in GemStone Processes

In this configuration, the application, the user actions, and one Gem all run in the same process (on the same machine). All function calls, from the application to GemBuilder and between GemBuilder and the Gem, are resolved by ordinary C-language linkage, not by remote procedure calls.

Since a Gem is required for each GemStone session, the first session uses the (linked) Gem that runs in your application process. This Gem has the advantages that it does not incur the overhead of remote procedure calls, and may not incur as much network traffic. It has the disadvantage that it must run in the same process as the Gem, so that work cannot be distributed between separate client and server

processes. Since the application cannot continue processing while the Gem is at work, the non-blocking GemBuilder functions provide no benefit here.

If a linked application user logs in to GemStone more than once, GemStone creates a new RPC Gem process for each new session. (These sessions would be additions to the configuration of Figure 3.3.) If one of these sessions invokes a user action, the user action executes in the same process as the application. If the user action then calls a GemBuilder function, that call is serviced by the linked Gem, not by the Gem from which the user action was invoked.

In this configuration, your code executes only on GemStone server platforms. It cannot execute on client-only platforms because a Gem is part of the same process. The occasions for using this configuration are much the same as those for running user actions with an RPC application, except that you should not use this one for debugging.

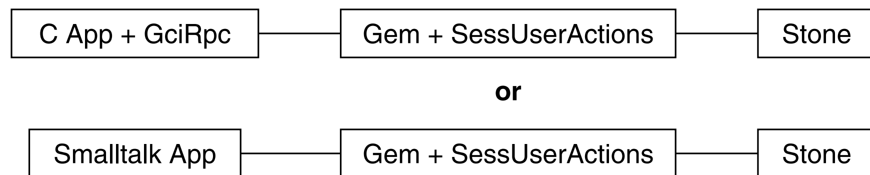
WARNING!
Debug your user actions in a process that does not include a Gem!
For more information, see "Risk of Database Corruption" on page 4-4.

Running User Actions with Gems

Just as with applications, there are two forms of Gems: linked and RPC. The linked Gem is embedded in the `gcilnk` library and is only used with linked applications.

Figure 3.4 illustrates how various architectural components are distributed among three GemStone processes when a set of user actions executes with an RPC Gem.

Figure 3.4 Session User Actions and RPC Gems in GemStone Processes



An RPC Gem executes in a separate process that can install and execute its own user actions. The RPC Gem is RPC because it communicates by means of remote procedure calls, through an RPC GemBuilder, with an application in another process.

However, it is also a separate C program. The Gem itself also uses GemBuilder directly, to interact with the database. That is the reason why the RPC Gem is linked with the `gcilnk` library. The user action in this configuration executes in

the same process as the Gem, with the GemBuilder that does *not* use remote procedure calls.

WARNING!
Debug your user actions in a process that does not include a Gem!
For more information, see *"Risk of Database Corruption" on page 4-4.*

The following list enumerates some of the conditions in which you may find occasion to use this configuration:

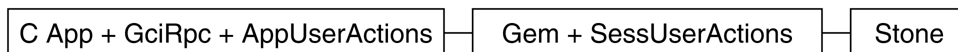
- You wish to execute the user action from a Smalltalk application using GemBuilder for Smalltalk. This configuration is required for that purpose.
- You wish the user action to be available to all or many other C applications.
- The user action is called frequently from GemStone. This configuration eliminates network traffic between GemBuilder and GemStone.
- The user action makes many calls to GemBuilder. This configuration avoids remote procedure call overhead.
- You have a GemStone object or objects that you wish to process first, and your application needs the result. The processing may be substantial. Your GemStone server machine may be more powerful than your client machine and could do it more quickly, or it might have specialized software the user action needs. Also, the result might be smaller and could reduce network traffic.

  For example, the user action might retrieve a data matrix and a filter from GemStone, perform a Fast Fourier Transform, and send the result to the application.

Running User Actions with Applications and Gems

Figure 3.5 illustrates how various architectural components are distributed among three GemStone processes when one set of user actions executes with an RPC application and another set of user actions executes with an RPC Gem.

Figure 3.5 RPC Applications and Gems with User Actions in GemStone Processes



This configuration is a combination of previous configurations. The application and the Gem run in separate processes. User actions in the first set execute in the application process, and user actions in the second set execute in the Gem process.

When user actions are installed in a process, they are given a name by which GemBuilder refers to them. If a user action in the application has the same name as a user action in the Gem, then the one in the Gem is always used, and the one in the application is ignored.

The two types of user actions could also exist in one linked process, as shown in Figure 3.6.

Figure 3.6 Application and Session User Actions in GemStone Processes

In this configuration, the user actions can be loaded as either application or session user actions; it would be the same from the point of view of the linked Gem. Application user actions would be just as efficient as session user actions, because they are part of the Gem process. If a linked application user logs in to GemStone more than once, GemStone creates a new RPC Gem process for each new session, additions to the configuration of Figure 3.6. The RPC Gems do not have access to the linked Gem's session user actions. So it is generally better to load them as application user actions, just in case.

Chapter 4

The Mechanics of Compiling and Linking

This chapter describes how to compile and link your C applications and user actions.

The focus is directly on operations for each compiling or linking alternative on each GemStone server platform[1]. It is assumed that you already know which alternatives you want to use, and why, and when. Those topics are part of the application design and code implementation, which are described in other chapters of this manual.

All operations are illustrated as though you are issuing commands at a command-line prompt. You may choose to take advantage of your system's programming aids, such as the Unix **make** utility and predefined environment variables, to simplify compilation and linking. Whatever you choose, be sure that you designate options and operations that are equivalent to those shown here.

NOTE:
Much of the material in this chapter is system-specific and, therefore, subject to change by compiler vendors and hardware manufacturers. Please check your GemStone Release Notes and vendor publications for possible updates.

1. Please see your *GemBuilder for C Release Notes* for analogous information on GemStone client platforms.

4.1 Development Environment and Standard Libraries

For simplicity, set the GEMSTONE environment variable to your GemStone installation directory. The command lines shown in this chapter assume that this has been done. No other environment variables are required to find the GemStone C libraries.

GemStone requires linking with certain architecture-specific "standard" C libraries on some platforms.[1] The order in which these libraries are specified can be significant; be sure to retain the ordering given in the command lines to follow in this section.

The environment of the supported Unix platforms is System V. On these platforms, the `/usr/bin` directory should be present in the PATH environment variable. If `/usr/ucb` is also present in PATH, then it should come after `/usr/bin`. The System V "standard" C libraries (*not* Berkeley) should be used in linking.

4.2 Compiling C Source Code for GemStone

The following information includes the requirements and recommendations for compiling C applications or user actions for GemStone. Your C code may have additional requirements, such as compile options or environment variables.

The C Compiler

Use a required or recommended C compiler to compile your GemStone C application. See the *GemStone Release Notes* for your platform for supported compilers.

Compilation Options

When you compile, specify each directory that is to be searched for include files separately by repeating the `-I` option. At a minimum, you should specify the GemStone `include` directory.

The `-c` option inhibits the "load and go" operation, so compilation ends when the compiler has produced an object file.

1. The *socket* library in particular contains operating system calls that support TCP/IP sockets. The functions for this purpose sometimes also require functions that are found in yet other system libraries.

For information on most options, please consult your compiler documentation.

Compilation Command Lines

Simple example command lines for compiling C source code on each platform follow.

The first command line for each platform illustrates how to compile a simple application program named *appl*, whose source contains one code file, *appl*.c. Its result is one object file, *appl*.o.

The second command line for each platform illustrates how to compile the user action file *useract*.c. For simplicity, this file is assumed to be a library containing both the source code for one set of user actions and the implementation of the function that installs them all with GemStone. Its result is one object file, *useract*.o. Some platforms require the object files that are put in a user action library to be compiled with special flags.

If you have multiple application or user action files, they should all be compiled under these same basic conditions.

Solaris (Sun):

```
$ cc -c -I$GEMSTONE/include     appl.c
$ cc -c -K PIC -I$GEMSTONE/include     useract.c
```

The -K PIC switch is required for user action library code.

HP-UX (HP):

```
$ cc -c -Aa -I$GEMSTONE/include     appl.c
$ cc -c -Aa +z -I$GEMSTONE/include     useract.c
```

The -Aa switch is required; it designates the ANSI C mode. The +z switch is required for user action library code.

AIX (IBM):

```
$ cc -c -I$GEMSTONE/include    appl.c
$ cc -c -I$GEMSTONE/include    useract.c
```

Windows NT:

```
C:\> cl -c -I%GEMSTONE%\include -DWIN32     appl.c
C:\> cl -c -I%GEMSTONE%\include -DWIN32     useract.c
```

The -DWIN32 switch is required.

4.3 Linking C Object Code with GemStone

The following information includes the requirements and recommendations for linking C applications or user actions with GemStone. Your code may have additional requirements, such as link options or libraries.

Run-time and Build-time Binding of Shared Libraries

GemStone uses shared libraries that can be bound to an executable at build time or run time. The semantics of linking differ, depending on which way a library is used.

When you choose run-time binding, the binding is done by code that is part of the application. The same application can use either the RPC or linked GemBuilder libraries with this type of binding.

When you choose build-time binding, the linker is told what shared library is to be used by the target that is being built. The linker records information about the shared library in the target. When the target is loaded by the operating system it automatically attempts to load the shared library that was build-time-bound. You must specify which GemBuilder library to use, so the application must commit to using either the RPC or linked library.

Linking with shared libraries does not require that all entry points be resolved at link time. Those that are outside of each shared library await resolution until application execution time, or even until function invocation time.

NOTE:
When you link a user action shared library, be aware of the dangers of incorrect unresolved external references. If you misspell a function call, you may not find out about it until run-time, when your process dies with an unresolved external reference error. Be sure to check your link program's output carefully.

Risk of Database Corruption

WARNING!
Debug your C code in a process that does not include a Gem!

Do not login to GemStone in a linked application or run a Gem with your user actions until your C code has been properly debugged!

When your C code executes in the same process as a Gem, it shares the same address space as the GemStone database buffers and object caches that are part of the Gem. If that C code has not yet been debugged, there is a danger that it might use a C pointer erroneously. Such an error could overwrite the Gem code or its data, with unpredictable and disastrous results. It is conceivable that such corruption of the Gem could lead it to perform undesired GemStone operations that might then leave your database irretrievably corrupt. The only remedy then is to restore the database from a backup.

There are three circumstances under which this risk arises:

- You are running your linked application and you have logged into GemStone.
- You are running any linked application and you are executing one of your user actions from the application.
- You are running any Gem, even a remote Gem, and you are executing one of your user actions from the Gem.

To avoid the risk, you must run your C code in some process that does not include a Gem. If the Gem is in a separate process, it has a separate address space that your C code should not be able to access. Use the RPC version of an application, and run any user actions from the application.

If you login to a linked session, GemStone records an entry to that effect in the Gem's system log file, `gemsys.log`. Check that log file to assure yourself that you are using a correct configuration for debugging.

GemStone Link Files

The following files can be found in the GemStone `lib` directory.

`gcirtlobj.o` (Unix)
Used when run-time-binding GemBuilder.

`gcirtl.lib` (Windows NT)
Used when run-time-binding GemBuilder.

`gciualib.o` (Unix)
Used when building a user action library.

`gciualib.obj` (Windows NT)
Used when building a user action library.

`libgcilnk50`.* (Unix)
GemBuilder LNK library. File extension varies by platform.

`gcilnk.lib (Windows NT)`
GemBuilder LNK library

`libgcirpc50.*` (Unix)
GemBuilder RPC library. File extension varies by platform.

`gcirpc.lib (Windows NT)`
GemBuilder RPC library

The Linker

On Unix, use the same C compiler to link your GemStone C code as you used to compile it.

On Windows NT, use the `link` program that comes with the same C compiler that you used to compile your GemStone C code.

Link Options

The `-o` option designates the path of the executable file produced by the link operation.

Be sure to employ at the appropriate times the link option that designates symbolic debugging (often `-g`).

For information on most options, please consult your linker (compiler) documentation.

Command Line Assumptions

Simple example command lines for linking object code on each platform follow. Two sets of application command lines are given, one for linking run-time-bound applications and one for linking build-time-bound RPC or linked applications.

Each command line illustrates how to link a simple application program named *appl* with one application object file, *appl*`.o`. Its result is one executable file, *appl* or *appl*`.exe`, depending on your platform.

User action lines illustrate how to link one user action object file *useract*`.o` with GemStone libraries to produce a user action library.

If you have multiple application or user action files, they should all be linked under the same basic conditions.

Linking Applications That Bind to GemBuilder at Run Time

Solaris (Sun):

```
$ cc -o  appl  appl.o $GEMSTONE/lib/gcirtlobj.o
   -lm -lsocket -lnsl
```

HP-UX (HP):

```
$ cc -z -o   appl  appl.o $GEMSTONE/lib/gcirtlobj.o -lm
```

The `-z` switch is highly recommended.

AIX (IBM):

```
$ cc -o  appl  appl.o $GEMSTONE/lib/gcirtlobj.o -lm
```

Windows NT:

```
% link  appl.obj %GEMSTONE%\lib\gcirtl.lib
```

Linking Applications That Bind to GemBuilder at Build Time

The following command lines include the shared library switch that determines the version of GemBuilder to which the application is linked. In `gci*50` and `gci*.lib`, for `*` substitute:

`lnk` for the GemBuilder LNK library

`rpc` for the GemBuilder RPC library

Solaris (Sun):

```
$ cc -o  appl  appl.o -L$GEMSTONE/lib -lgci*50
   -lm -lsocket -lnsl
```

HP-UX (HP):

```
$ cc -z -o   appl  appl.o -L$GEMSTONE/lib -lgci*50 -lm
```

The `-z` switch is highly recommended.

AIX (IBM):

```
$ cc -o  appl  appl.o -L$GEMSTONE/lib -lgci*50 -lm
```

Windows NT:

```
% link  appl.obj %GEMSTONE%\lib\gci*.lib
```

Linking User Actions into Shared Libraries

Solaris (Sun):

```
$ ld -G -z text -B symbolic -o libua.so -h libua.so useract.o
   $GEMSTONE/lib/gciualib.o -ldl -lm -lsocket -lnsl -lc
```

HP-UX (HP):

```
$ ld -b -z -o libua.sl +e GciUserActionLibraryMain
      useract.o $GEMSTONE/lib/gciualib.o -ldld -lm
```

The `-z` switch is highly recommended.

AIX (IBM):

```
$ cc -bM:SRE -o libua.o -e GciUserActionLibraryMain
      useract.o $GEMSTONE/lib/gciualib.o -lm
```

Windows NT:

```
% link -dll -entry:_DllMainCRTStartup@12 -out:useract.dll
      useract.obj %GEMSTONE%\lib\gciualib.obj
```

Chapter 5

GemBuilder C Functions — A Reference Guide

This chapter describes the GemBuilder functions that may be called by your C application program.

5.1 Function Summary Tables

Tables 5.1 through 5.7 summarize the GemBuilder C functions and the services that they provide to your application.

Table 5.1 Functions for Controlling Sessions and Transactions

GciAbort	Abort the current transaction.
GciAddSaveObjsToReadSet	Add all objects in the export set to the read set of the current transaction.
GciAlteredObjs	Within a given array of cached objects, find those that have changed in the database.
GciBegin	Begin a new transaction.
GciCheckAuth	Gather the current authorizations for an array of database objects.
GciCommit	Write the current transaction to the database.
GciDirtyObjsInit	Begin tracking which objects in the session workspace change.

Table 5.1 Functions for Controlling Sessions and Transactions

Function	Description
GciDirtySaveObjs	Find all objects in the export set that have changed since the last changes were found.
GciGetSessionId	Find the ID number of the current user session.
GciHardBreak	Interrupt GemStone and abort the current transaction.
GciInit	Initialize GemBuilder.
GciInitAppName	Override the default application configuration file name.
GciInstallUserAction	Associate a C function with a Smalltalk user action.
GciIsRemote	Determine whether the application is running linked or remotely.
GciLoadUserActionLibrary	Load an application user action library.
GciLogin	Start a user session.
GciLogout	End the current user session.
GciNbAbort	Abort the current transaction (nonblocking).
GciNbBegin	Begin a new transaction (nonblocking).
GciNbCommit	Write the current transaction to the database (nonblocking).
GciProcessDeferredUpdates	Process deferred updates to objects that do not allow direct structural update.
GciReleaseAllOops	Mark all imported GemStone OOPs as eligible for garbage collection.
GciReleaseOops	Mark an array of GemStone OOPs as eligible for garbage collection.
GciRtlIsLoaded	Report whether a GemBuilder library is loaded.
GciRtlLoad	Load a GemBuilder library.
GciRtlUnload	Unload a GemBuilder library.
GciSaveObjs	Mark an array of OOPs as ineligible for garbage collection.
GciSessionIsRemote	Determine whether or not the current session is using a Gem on another machine.
GciSetNet	Set network parameters for connecting the user to the Gem and Stone processes.
GciSetSessionId	Set an active session to be the current one.
GciShutdown	Logout from all sessions and deactivate GemBuilder.
GciUserActionInit	Declare user actions for GemStone.

Table 5.2 Functions for Handling Errors and Interrupts and for Debugging

GciCallInProgress	Determine if a GemBuilder call is currently in progress.
GciClearStack	Clear the Smalltalk call stack.
GciContinue	Continue code execution in GemStone after an error.
GciContinueWith	Continue code execution in GemStone after an error.
GciDbgEstablish	Specify the debugging function for GemBuilder to execute before most calls to GemBuilder functions.
GciEnableSignaledErrors	Establish or remove GemBuilder visibility to signaled errors from GemStone.
GciErr	Prepare a report describing the most recent GemBuilder error.
GciHandleError	Check the previous GemBuilder call for an error.
GciInUserAction	Determine whether or not the current process is executing a user action.
GciNbContinue	Continue code execution in GemStone after an error (nonblocking).
GciNbContinueWith	Continue code execution in GemStone after an error (nonblocking).
GciPollForSignal	Poll GemStone for signal errors without executing any Smalltalk methods.
GciPopErrJump	Discard a previously saved error jump buffer.
GciPushErrHandler	Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.
GciPushErrJump	Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.
GciRaiseException	Signal an error, synchronously, within a user action.
GciSetErrJump	Enable or disable the current error handler.
GciSoftBreak	Interrupt the execution of Smalltalk code, but permit it to be restarted.

Table 5.3 Functions for Compiling and Executing Smalltalk Code in the Database

Function	Description
GciClassMethodForClass	Compile a class method for a class.
GciExecute	Execute a Smalltalk expression contained in a String object.
GciExecuteFromContext	Execute a Smalltalk expression contained in a String object as if it were a message sent to another object.
GciExecuteStr	Execute a Smalltalk expression contained in a C string.
GciExecuteStrFromContext	Execute a Smalltalk expression contained in a C string as if it were a message sent to an object.
GciInstMethodForClass	Compile an instance method for a class.
GciNbExecute	Execute a Smalltalk expression contained in a String object (nonblocking).
GciNbExecuteStr	Execute a Smalltalk expression contained in a C string (nonblocking).
GciNbExecuteStrFromContext	Execute a Smalltalk expression contained in a C string as if it were a message sent to an object (nonblocking).
GciNbPerform	Send a message to a GemStone object (nonblocking).
GciNbPerformNoDebug	Send a message to a GemStone object, and temporarily disable debugging (nonblocking).
GciPerform	Send a message to a GemStone object.
GciPerformNoDebug	Send a message to a GemStone object, and temporarily disable debugging.
GciPerformSym	Send a message to a GemStone object, using a String object as a selector.
GciPerformTraverse	First send a message to a GemStone object, then traverse the result of the message.
GciSendMsg	Send a message to a GemStone object.

Table 5.4 Functions for Accessing Symbol Dictionaries

GciResolveSymbol	Find the OOP of the object to which a symbol name refers, in the context of the current session's user profile.
GciResolveSymbolObj	Find the OOP of the object to which a symbol object refers, in the context of the current session's user profile.
GciSymDictAt	Find the value in a symbol dictionary at the corresponding string key.
GciSymDictAtObj	Find the value in a symbol dictionary at the corresponding object key.
GciSymDictAtObjPut	Store a value into a symbol dictionary at the corresponding object key.
GciSymDictAtPut	Store a value into a symbol dictionary at the corresponding string key.
GciTraverseObjs	Find the value in a symbol KeyValue dictionary at the corresponding string key.
GciStrKeyValueDictAtObj	Find the value in a symbol KeyValue dictionary at the corresponding object key.
GciStrKeyValueDictAtObj Put	Store a value into a symbol KeyValue dictionary at the corresponding object key.
GciStrKeyValueDictAtPut	Store a value into a symbol KeyValue dictionary at the corresponding string key.

Table 5.5 Functions for Creating and Initializing Objects

GciCreateByteObj	Create a new byte-format object.
GciCreateOopObj	Create a new pointer-format object.
GciGetFreeOop	Allocate an OOP.
GciGetFreeOops	Allocate multiple OOPs.
GciNewByteObj	Create and initialize a new byte object.
GciNewCharObj	Create and initialize a new character object.
GciNewDateTime	Create and initialize a new date-time object.
GciNewOop	Create a new GemStone object.
GciNewOops	Create multiple new GemStone objects.
GciNewOopUsingObjRep	Create a new GemStone object from an existing object report.
GciNewString	Create a new String object from a C character string.
GciNewSymbol	Create a new Symbol object from a C character string.

Table 5.6 Functions and Macros for Converting Objects and Values

GCI_BOOL_TO_OOP	(MACRO) Convert a C Boolean value to a GemStone Boolean object.
GCI_CHR_TO_OOP	(MACRO) Convert a C character value to a GemStone Character object.
GciCTimeToDateTime	Convert a C date-time representation to GemStone's.
GciDateTimeToCTime	Convert a GemStone date-time representation to C's.
GciFetchDateTime	Convert the contents of a DateTime object and place the results in a C structure.
GciFltToOop	Convert a C double value to a Float object.
GCI_LONG_IS_SMALL_INT	(MACRO) Determine whether or not a long can be translated into a SmallInteger.
GciLongToOop	Find a GemStone object that corresponds to a C long integer.
GCI_LONG_TO_OOP	(MACRO) Find a GemStone object that corresponds to a C long integer.
GCI_OOP_IS_BOOL	(MACRO) Determine whether or not a GemStone object represents a Boolean value.
GCI_OOP_IS_SMALL_INT	(MACRO) Determine whether or not a GemStone object represents a SmallInteger.
GCI_OOP_IS_SPECIAL	(MACRO) Determine whether or not a GemStone object has a special representation.
GciOopToBool	Convert a Boolean object to a C Boolean value.
GCI_OOP_TO_BOOL	(MACRO) Convert a Boolean object to a C Boolean value.
GciOopToChr	Convert a Character object to a C character value.
GCI_OOP_TO_CHR	(MACRO) Convert a Character object to a C character value.
GciOopToFlt	Convert a Float object to a C double value.
GciOopToLong	Convert a Gemstone object to a C long integer value.
GCI_OOP_TO_LONG	(MACRO) Convert a GemStone object to a C long integer value.
GciUnsignedLongToOop	Find a GemStone object that corresponds to a C unsigned long integer.

Table 5.7 Object Traversal and Path Functions and Macros

GCI_ALIGN	(MACRO) Align an address to a word boundary.
GciClampedTraverseObjs	Traverse an array of objects, subject to clamps.
GciExecuteStrTrav	Execute a string and traverse the result of the execution.
GciFetchPaths	Fetch selected multiple OOPs from an object tree.
GciFindObjRep	Fetch an object report in a traversal buffer.
GCI_IS_REPORT_CLAMPED	(MACRO) Determine whether or not an object was clamped during traversal.
GciMoreTraversal	Continue object traversal, reusing a given buffer.
GciNbClampedTraverseObjs	Traverse an array of objects, subject to clamps (nonblocking).
GciNbExecuteStrTrav	Execute a string and traverse the result of the execution (nonblocking).
GciNbMoreTraversal	Continue object traversal, reusing a given buffer (nonblocking).
GciNbStoreTrav	Store multiple traversal buffer values in objects (nonblocking).
GciNbTraverseObjs	Traverse an array of GemStone objects (nonblocking).
GciObjRepSize	Find the number of bytes in an object report.
GciPathToStr	Convert a path representation from numeric to string.
GciPerformTraverse	First send a message to a GemStone object, then traverse the result of the message.
GciStorePaths	Store selected multiple OOPs into an object tree.
GciStoreTrav	Store multiple traversal buffer values in objects.
GciStrToPath	Convert a path representation from string to numeric.
GciTraverseObjs	Traverse an array of GemStone objects.
GCI_VALUE_BUFF	(MACRO) Find a pointer to the value buffer of an object report.
GciVersion	Return a string that describes the GemBuilder version.

Table 5.8 Structural Access Functions

GciAddOopToNsc	Add an OOP to the unordered variables of a non-sequenceable collection.
GciAddOopsToNsc	Add multiple OOPs to the unordered variables of a non-sequenceable collection.
GciAppendBytes	Append bytes to a byte object.
GciAppendChars	Append a C string to a byte object.
GciAppendOops	Append OOPs to the unnamed variables of a collection.
GciClassNamedSize	Find the number of named instance variables in a class.
GciFetchByte	Fetch one byte from an indexed byte object.
GciFetchBytes	Fetch multiple bytes from an indexed byte object.
GciFetchChars	Fetch multiple ASCII characters from an indexed byte object.
GciFetchClass	Fetch the class of an object.
GciFetchNamedOop	Fetch the OOP of one of an object's named instance variables.
GciFetchNamedOops	Fetch the OOPs of one or more of an object's named instance variables.
GciFetchNamedSize	Fetch the number of named instance variables in an object.
GciFetchNameOfClass	Fetch the class name object for a given class.
GciFetchObjImpl	Fetch the implementation of an object.
GciFetchObjInfo	Fetch information and values from an object.
GciFetchOop	Fetch the OOP of one instance variable of an object.
GciFetchOops	Fetch the OOPs of one or more instance variables of an object.
GciFetchSize	Fetch the size of an object.
GciFetchVaryingOop	Fetch the OOP of one unnamed instance variable from an indexed pointer object or NSC.
GciFetchVaryingOops	Fetch the OOPs of one or more unnamed instance variables from an indexed pointer object or NSC.
GciFetchVaryingSize	Fetch the number of unnamed instance variables in a pointer object or NSC.
GciIsKindOf	Determine whether or not an object is some kind of a given class or class history.
GciIsKindOfClass	Determine whether or not an object is some kind of a given class.
GciIsSubclassOf	Determine whether or not a class is a subclass of a given class or class history.
GciIsSubclassOfClass	Determine whether or not a class is a subclass of a given class.

Table 5.8 Structural Access Functions

GciIvNameToIdx	Fetch the index of an instance variable name.
GciObjExists	Determine whether or not a GemStone object exists.
GciObjInCollection	Determine whether or not a GemStone object is in a Collection.
GciRemoveOopFromNsc	Remove an OOP from an NSC.
GciRemoveOopsFromNsc	Remove one or more OOPs from an NSC.
GciReplaceOops	Replace all instance variables in a GemStone object.
GciReplaceVaryingOops	Replace all unnamed instance variables in an NSC object.
GciStoreByte	Store one byte in a byte object.
GciStoreBytes	Store multiple bytes in a byte object.
GciStoreChars	Store multiple ASCII characters in a byte object.
GciStoreIdxOop	Store one OOP in a pointer object’s unnamed instance variable.
GciStoreIdxOops	Store one or more OOPs in a pointer object’s unnamed instance variables.
GciStoreNamedOop	Store one OOP into an object’s named instance variable.
GciStoreNamedOops	Store one or more OOPs into an object’s named instance variables.
GciStoreOop	Store one OOP into an object’s instance variable.
GciStoreOops	Store one or more OOPs into an object’s instance variables.

5.2 GemBuilder Include Files

The following include files are provided for use with GemBuilder C functions. These files are in the `$GEMSTONE/include` directory. For distinctions between the run-time and build-time files, see "Binding to GemBuilder at Run Time" on page 2-3.

You can include these files in your code. The first three are mutually exclusive.

`gcirtl.hf`	Forward references to the GemBuilder functions, to be included in code that will bind to GemBuilder at run time. For modules that define user actions, use `gciua.hf` instead of this file.
`gci.hf`	Forward references to the GemBuilder functions, to be included in code that will bind to GemBuilder at build time. For modules that define user actions, use `gciua.hf` instead of this file.
`gciua.hf`	Used instead of `gcirtl.hf` or `gci.hf` in modules that define user actions.
`gcifloat.hf`	Macros, constants and functions for accessing the bodies of instances of GemStone classes Float and SmallFloat. Optional for code that includes `gci.hf` and `gciua.hf`, not used with `gcirtl.hf`.

You do not include these files explicitly; they are listed here for your information.

`flag.ht`	Contains host-specific C definitions for compilation.
`gci.ht`	Defines C types for use by GemBuilder functions. See "GemBuilder Data Types" on page 5-11.
`gcicmn.ht`	Defines common C types and macros used by `gcirtl.hf`, `gci.hf`, and `gciua.hf`.
`gcierr.ht`	Defines mnemonics for all GemStone errors.
`gcioc.ht`	Defines C mnemonics for sizes and offsets into objects.
`gcioop.ht`	Defines C mnemonics for predefined GemStone objects. See Appendix A, "Reserved OOPs," for a list of constants defined in this file.
`gcirtl.ht`	Defines C types specific to shared libraries for use by GemBuilder functions. Used by `gcirtl.hf`.
`gcirtlm.hf`	Macros used by `gcirtl.hf`.

`gciuser.hf` Defines a macro to be used to install user actions. Include `gciua.hf` instead of this file.

`staticua.hf` Included in modules that define static user actions.

`version.ht` Defines C mnemonics for version-dependent strings.

5.3 GemBuilder Data Types

The following C types are used by GemBuilder functions. The file `gci.ht` defines each of the GemBuilder types (shown in capital letters below). That file is in the `$GEMSTONE/include` directory.

jmp_buf Jump buffer, defined in the `setjmp.h` file.

ArraySizeType An unsigned int.

BoolType An int.

ByteType An unsigned 8-bit integer.

OopType Object-oriented pointer, a signed 32-bit integer.

GciClampedTravArgsSType
A structure for clamped traversal arguments.

GciDateTimeSType
A structure for representing GemStone dates and times.

GciDbgFuncType
The type of C function called by **GciDbgEstablish.**

GciErrSType A GemStone error report (see “The Error Report Structure” on page 5-12).

GciObjInfoSType
A GemStone object information report (see “The Object Information Structure” on page 5-13).

GciObjRepHdrSType
An object report header (see “The Object Report Header Structure” on page 5-15).

GciObjRepSType
An object report (see “The Object Report Structure” on page 5-14).

GciSessionIdType
A signed 32-bit integer.

GciUserActionSType
A structure for describing a user action (see “The User Action Information Structure” on page 5-17.

The Structure for Representing the Date and Time

GemBuilder includes some functions to facilitate access to objects of type DateTime. (These functions also make use of the C representation for time, **time_t**.)

The structured type **GciDateTimeSType**, which provides a C representation of an instance of class DateTime, contains the following fields:

```
typedef struct {
    long year;
    long dayOfYear;
    long seconds;
} GciDateTimeSType;
```

The `year` value must be less than 1,000,000.

In addition, a C mnemonic supports representation of DateTime objects.

```
#define GCI_SECONDS_PER_DAY    86400
/* conversion constant */
```

NOTE:
The OOP of the Smalltalk DateTime class is OOP_CLASS_DATE_TIME.

The Error Report Structure

An error report is a C structured type named **GciErrSType**. This structure contains the following fields:

OopType *category*
The OOP of the GemStone error dictionary (OOP_GEMSTONE_ERROR_CAT).

long *number*
The GemStone error number (a positive integer).

OopType *context*
The OOP of a Process that provides the state of the virtual machine for use in debugging. This Process can be used as the

argument to GciContinue or GciClearStack. If the virtual machine was not running, then *context* is OOP_NIL. If you are not interested in debugging or in continuing from an error, or if the error is not in the runtime error category, your program can ignore this value.

char *message*[GCI_ERR_STR_SIZE + 1]
The null-terminated string which contains the text of the error message. In this release, GCI_ERR_STR_SIZE is defined to be 300.

OopType *args*[GCI_MAX_ERR_ARGS]
An optional array of error arguments. In this release, GCI_MAX_ERR_ARGS is defined to be 10.

long *argCount*
The number of arguments in the args array.

BoolType *fatal*
TRUE if this error is fatal, FALSE otherwise.

The arguments (*args*) are specific to the error encountered. In the case of a compiler error, this is a single argument — the OOP of an array of error identifiers. Each identifier is an Array with three elements: (1) the error number (a SmallInteger); (2) the offset into the source string at which the error occurred (also a SmallInteger); and (3) the text of the error message (a String). See the `gcierr.ht` file for a full list of errors and their arguments.

In the case of a fatal error, fatal is set to nonzero (TRUE). Your connection to GemStone is lost, and the current session ID (from **GciGetSessionId**) is reset to GCI_INVALID_SESSION_ID.

The Object Information Structure

Object information is placed in a C structured type named **GciObjInfoSType**. Object information access functions provide information about objects in the database. These functions offer C-style access to much information that would otherwise be available only through calls to GemStone. For more information about the **GciObjInfoSType** structured type, refer to the **GciFetchObjInfo** function on page 5-97.

long *namedSize*
Number of named instance variables in the object.

OopType *objId*
OOP of the object.

OopType *objClass*
Class of object (see the **GciFetchClass** function on page 5-85).

OopType *segment*
The object's segment.

long *objSize*
Object's total size in bytes or OOPs (see the **GciFetchSize** function on page 5-108).

GciObjHdrBFType *objImpl*
(3 bits): Implementation format.

GciObjHdrBFType *isInvariant*
(1 bit): Boolean to show if object is invariant at the object level.

GciObjHdrBFType *isIndexable*
(1 bit): Boolean to show if object is indexable.

GciObjHdrBFType *unnamed*
(11 bits): Reserved for future use.

The `gcioc.ht` include file defines four mnemonics that can be of assistance when you are handling the object implementation field: GC_FORMAT_OOP, GC_FORMAT_BYTE, GC_FORMAT_NSC, and GC_FORMAT_SPECIAL. These mnemonics, and no other values, should be used to supply values for the *objImpl* field, or to test its contents.

The Object Report Structure

Each object report has two parts: a fixed-size header (as defined in the **GciObjRepHdrSType** structure) and a variable-size value buffer (an array of the values of the object's instance variables):

```
typedef struct {
GciObjRepHdrSType hdr; /* object report header */
union {        /* object's value buffer */
   ByteType bytes[1];
   OopType oops[1];
} u;
} GciObjRepSType;
```

The Object Report Header Structure

An object report header is a C structured type named **GciObjRepHdrSType**. This structure holds a general description of an object, and contains the following fields:

long *valueBuffSize*
Size (in bytes) of the object's value buffer.

long *namedSize*
Number of named instance variables in the object.

long *idxSize*
Number of indexed instance variables.

long *firstOffset*
Offset of first value to fetch or store.

OopType *objId*
OOP of the object.

OopType *oclass*
Class of object (see the **GciFetchClass** function on page 5-85).

OopType *segment*
The object's segment.

long *objSize*
Object's total size in bytes or OOPs (see the **GciFetchSize** function on page 5-108).

GciObjHdrBFType *objImpl*
(3 bits): Implementation format.

GciObjHdrBFType *isInvariant*
(1 bit): Boolean to show if object is invariant at the object level.

GciObjHdrBFType *isIndexable*
(1 bit): Boolean to show if object is indexable.

GciObjHdrBFType *unnamed*
(11 bits): Reserved for future use.

short *unused*
Reserved for future use.

If the specified *idxSize* is inadequate to accommodate the contents of the value buffer (the values in *u.bytes* or *u.oops*), this function will automatically increase *idxSize* (the number of the object's indexed variables) as needed. Of course, if the specified *objClass* is not indexable, then the *idxSize* is ignored.

The *firstOffset* indicates where to begin storing values into the object's array of instance variables. In that array, the object's named instance variables are followed by its unnamed variables. If *firstOffset* is not 1, all instance variables (named or indexed) up to the *firstOffset* will be initialized to nil or 0. The *firstOffset* must be in the range (`1, objSize+1`).

On input, if the specified segment is OOP_NIL, the object is created in the session's current segment.

The `gcioc.ht` include file defines four mnemonics that can be of assistance when you are handling the object implementation field (*objImpl*): GC_FORMAT_OOP, GC_FORMAT_BYTE, GC_FORMAT_NSC, and GC_FORMAT_SPECIAL. These mnemonics, and no other values, should be used to supply values for *objImpl*, or to test its contents. However, the `gcioc.ht` file also defines other mnemonics that can be used in other contexts related to object implementations, indexability, and invariance.

An object's implementation may restrict the number of its named instance variables (*namedSize*) and its indexed instance variables (*idxSize*), as contained in the object report header.

- If the object implementation is GC_FORMAT_OOP, the object can have both named and unnamed instance variables.
- If the object implementation is GC_FORMAT_BYTE, the object can only have indexed instance variables, and its *namedSize* is always zero.
- If the object implementation is GC_FORMAT_NSC, the object can have both named and unnamed instance variables. (The NSC's *idxSize* reports the number of unnamed instance variables, even though they are unordered, not indexed.)
- If the object implementation is GC_FORMAT_SPECIAL, the object cannot have any instance variables, and the number of both its named and unnamed variables is always zero.

An object is invariant "at the class level" if its class was created with the argument `instancesInvariant: true` , and then only after the object has been committed to the database. The *isInvariant* field of the object report header does not reflect class invariance.

The *isInvariant* field is set (to 1 or true) only when the object itself is invariant ("at the object level"). This can happen in one of two ways:

- The application program sends the message `immediateInvariant` to the object.
- The application program explicitly sets the *isInvariant* field true in the report header and then uses that report header in a call to **GciStoreTrav**.

Thus, testing the *isInvariant* field of the report header reveals only object-level invariance. If you wish to test an object for both object-level and class-level invariance, you must send the message `Object | isInvariant` to the object and check the result.

Table 5.9 Object Implementation Restrictions on Instance Variables

Object Implementation	Named Instance Variables OK?	Unnamed Instance Variables OK?
0=Pointer	YES	YES (always indexed)
1=Byte	NO	YES (always indexed)
2=NSC	YES	YES (always unordered)
3=Special	NO	NO

For more information about object implementation types, see "Manipulating Objects Through Structural Access" on page 1-16.

The User Action Information Structure

The structured type **GciUserActionSType** describes a user action function. It defines the following fields:

char *userActionName*[GCI_MAX_ACTION_NAME+1]
The user action name (a case-insensitive, null-terminated string). In this release, GCI_MAX_ACTION_NAME is defined to be 31.

long *userActionNumArgs*
The number of arguments in the C function.

GciUserActionFType
userAction
A pointer to the C user action function.

unsigned long *userActionFlags*
Mainly for internal use. If you use it, set it to 0 before passing a pointer to it.

5.4 Structural Access Functions

The following caution applies to GemBuilder's structural access functions listed in Table 5.7 on page 5-7:

WARNING!
Exercise caution when using structural access functions. Although they can improve the speed of GemStone database operations, these functions bypass GemStone's message-sending metaphor. That is, structural access functions may bypass any checking that might be coded into your application's methods. In using structural access functions, you implicitly assume full responsibility for safeguarding the integrity of your system.

Note, however, that structural access functions do not bypass checks on constraint violations, authorization violations, or concurrency conflicts.

5.5 Unix Interrupt Handling

Both versions of GemBuilder (GciLnk and GciRpc) use the SIGIO interrupt handler. If you must install your own interrupt handler (using signal or sigvec), be sure that your application calls the previous interrupt handling routine when done.

WARNING!
Do not, under any circumstances, turn off SIGIO.

5.6 Reserved Prefixes

To avoid identifier name conflicts when linking your application, use naming prefixes that distinguish your own identifiers. You should avoid prefixes and identifiers that clash with those that are used in GemStone. GemStone identifier names are subject to change without notice in future releases of GemStone. Your names should differ from them in more than just alphabetic case. For a complete

list of identifier names to avoid, use the Unix **nm** command on the GemStone object files you use:

```
cd $GEMSTONE/lib
nm gcilnkobj.o gcirpcobj.o gciuser.o gemrpcobj.o
```

In any case, avoid using the following reserved identifier prefixes:

```
Add   Ass     Auth    BagPrim
Bin   Bm      Cfg     Class
Com   DataPg  DatePrim  Dbf
Dnet  DoPrim  Dtime   EUC
Egc   Egs     Err     FindEmpty
Flt   Gci     Gdbg    Gem
Hash  Host    Idx     Int
JIS   Lgc     Lgs     Lom
Lrg   MethPrim  Net   Nsc
Obj   Page    Pom     RDbf
Rep   Root    Scan    Scavenge
Snet  Stn     StrPrim  SysPrim
Tnet  TpSup   Unix    Utl
Ver   Work
```

5.7 GemBuilder Function and Macro Reference

This section provides a complete description of each GemBuilder C function that your application can call.

GciAbort

Abort the current transaction.

Syntax

void GciAbort( )

Description

This function causes the GemStone system to abort the current transaction. All changes to persistent objects that were made since the last committed transaction are lost, and the application is connected to the most recent version of the database. Your application must fetch again from GemStone any changed persistent objects, to refresh the copies of these objects in your C program. Use the **GciDirtySaveObjs** function to determine which of the fetched objects were also changed.

This function has the same effect as issuing a hard break, or the function call `GciExecuteStr("System abortTransaction", OOP_NIL)` . For more information, see “Interrupting GemStone Execution” on page 1-14.

Example

```
GciErrSType myError;

/* Call GciAbort and let the user know we aborted */
GciAbort();
if (GciErr(&myError) &&
   ((myError.category != OOP_ABORTING_ERROR_CAT) ||
    (myError.number  != RT_ERR_ABORT_TRANS)) )
printf("GemStone returned error #%d in category %d\n",
       myError.number, myError.category);
else
printf("Transaction aborted successfully.\n");
```

See Also

GciCheckAuth, page 5-34
GciCommit, page 5-48

GciNbAbort, page 5-155
GciNbCommit, page 5-160

GciAddOopToNsc

Add an OOP to the unordered variables of a non-sequenceable collection.

Syntax

void GciAddOopToNsc(*theNsc*, *theOop*)
    OopType *theNsc*;
    OopType *theOop*;

Input Arguments

theNsc	The OOP of the NSC.
theOop	The OOP to be added.

Description

This function adds an OOP to the unordered variables of an NSC, using structural access.

Example

```
int   i;
OopType oNscObject;
OopType oNum;

oNscObject = GciNewOop(OOP_CLASS_IDENTITY_BAG);
for (i = 0; i < tsize; i ++) {
oNum = GciLongToOop((long)i);
GciAddOopToNsc(oNscObject, oNum);
}
```

See Also

GciAddOopsToNsc, page 5-23
GciRemoveOopFromNsc, page 5-241
GciRemoveOopsFromNsc, page 5-243

GciAddOopsToNsc

Add multiple OOPs to the unordered variables of a non-sequenceable collection.

Syntax

void GciAddOopsToNsc(*theNsc*, *theOops*, *numOops*)
OopType *theNsc*;
const OopType *theOops*[ ];
ArraySizeType *numOops*;

Input Arguments

theNsc	The OOP of the NSC.
theOops	An array of OOPs to be added.
numOops	The number of OOPs to add.

Description

This function adds multiple OOPs to the unordered variables of an NSC, using structural access.

Example

```
int   i;
OopType anObject;
OopType bigptrs[L_SUB_SIZE];

for (i = 0; i < L_SUB_SIZE; i ++)
bigptrs[i] = GciLongToOop((long)i);
GciAddOopsToNsc(anObject, bigptrs, L_SUB_SIZE);
```

See Also

GciAddOopToNsc, page 5-22
GciRemoveOopFromNsc, page 5-241
GciRemoveOopsFromNsc, page 5-243

GciAddSaveObjsToReadSet

Add all objects in the export set to the read set of the current transaction.

Syntax

void GciAddSaveObjsToReadSet( )

Description

The **GciAddSaveObjsToReadSet** function adds to the read set of the current transaction any objects from the export set that are not already in the read set.

See Also

GciSaveObjs, page 5-254

GciAlteredObjs

Within a given array of cached objects, find those that have changed in the database.

Syntax

void GciAlteredObjs(*theOops*, *numOops, canonicalSymbolBuf, numPairs*)
  OopType *theOops[]*;
  ArraySizeType * *numOops*;
  OopType *canonicalSymbolBuf[]*;
  ArraySizeType * *numPairs;*

Input Arguments

theOops	An array of OOPs of objects that may have changed. The caller must provide these values.
numOops	Pointer to the number of OOPs in the input array.
canonicalSymbolBuf	An array of oldSymbolOop/canonicalSymbolOop pairs identifying canonical symbol objects that may have changed. The caller must provide these values.
numPairs	Pointer to the number of pairs in the input array.

Result Arguments

theOops	The resulting array of OOPs of objects. The objects have been modified as a consequence of other transactions since this database session's most recent abort or successful commit. This array is a subset of the input array.
* *numOops*	Modified to be the number of OOPs in the resulting array.
canonicalSymbolBuf	The resulting array of oldSymbolOop/ canonicalSymbolOop pairs. The canonical symbol objects have been modified as a consequence of other transactions since this database session's most recent abort or successful commit. This array is a subset of the input array.
* *numPairs*	Modified to be the number of pairs in the resulting array.

Return Value

GciAlteredObjs returns TRUE if all the modified objects have been returned, and FALSE otherwise. As long as the function returns FALSE, the call should be repeated. If it is not, the unreturned objects persist in the list until the next time **GciAlteredObjs** is called.

Description

Typically, a GemStone C application program caches some database objects in its local object space. After an abort or a successful commit, the user's session is resynchronized with the most recent version of the database. The OOP values previously imported by your C program may no longer accurately represent the corresponding GemStone objects. In such cases, your C program must update its representation of those objects. The function **GciAlteredObjs** permits you to determine which objects your application needs to reread from the database.

Results of the **GciAppend...**, **GciReplace...**, and **GciCreate...** functions are not automatically added to your session's set of modified objects. Results of **GciStore...** are only added to the set of modified objects if the function was called from within a user action.

The application designer must ensure that all the cached objects are placed in the user session's export set. **GciAlteredObjs** only returns changed objects that are in the export set (see **GciSaveObjs**).

NOTE:
GciAlteredObjs *removes all OOPs from the user session's set of modified objects, including the objects that were not returned because they are not in the export set.*

See Also

GciAbort, page 5-20
GciCommit, page 5-48
GciReleaseAllOops, page 5-238
GciReleaseOops, page 5-239
GciSaveObjs, page 5-254

GCI_ALIGN

(MACRO) Align an address to a word boundary.

Syntax

char * GCI_ALIGN(*theAddress*)

Input Arguments

theAddress	The address to be aligned.

Result Value

A word-aligned address that is greater than or equal to the input value of *theAddress*.

Description

This macro is used during object traversals to ensure that the value buffer portion of each object report begins at a word boundary, and that the beginning of each object report in a traversal buffer is properly aligned.

Example

```
unsigned char  travBuf[1000];   /* assumed to be aligned */
unsigned char * ptr = &travBuf[0];

do {
fillReport(ptr);
ptr += (long)GCI_ALIGN(sizeof(GciObjRepHdrSType) +
       ptr->valueBuffSize);
} while (!done);
```

See Also

GciMoreTraversal, page 5-152
GciNewOopUsingObjRep, page 5-191
GciTraverseObjs, page 5-307

GciAppendBytes

Append bytes to a byte object.

Syntax

void GciAppendBytes(*theObject*, *numBytes*, *theBytes*)

OopType	*theObject*;
ArraySizeType	*numBytes*;
const ByteType *	*theBytes*;

Input Arguments

theObject	A byte object to which bytes are to be appended.
numBytes	The number of bytes to be appended.
theBytes	A pointer to the bytes to be appended.

Result Arguments

theObject	The resulting byte object, with the appended bytes.

Description

The **GciAppendBytes** function appends *numBytes* bytes to byte object *theObject*. Its effect is equivalent to `GciStoreBytes(x, GciFetchSize(x)+1,` *theBytes*`,` *numBytes*`)`.

GciAppendBytes raises an error if *theObject* is a binary float. Binary floats are of a fixed and unchangeable size.

See Also

GciAppendChars, page 5-29

GciAppendChars

Append a C string to a byte object.

Syntax

void GciAppendChars(*theObject*, *aString*)
OopType *theObject*;
const char * *aString*;

Input Arguments

theObject	A byte object to which the string is to be appended.
aString	A pointer to the string to be appended.

Result Arguments

theObject	The resulting byte object, with the appended string.

Description

This function appends the characters of *aString* to byte object *theObject*.

See Also

GciAppendBytes, page 5-28

GciAppendOops

Append OOPs to the unnamed variables of a collection.

Syntax

void GciAppendOops(*theObject*, *numOops*, *theOops*)
  OopType *theObject*;
  ArraySizeType *numOops*;
  const OopType* *theOops*;

Input Arguments

theObject	A collection to which additional OOPs are to be added.
numOops	The number of OOPs to be added.
theOops	A pointer to the OOPs to be added.

Result Arguments

theObject	The resulting collection, with the added OOPs.

Description

Appends *numOops* OOPs to the unnamed variables of the collection *theObject*. If the collection is indexable, this is equivalent to:

```
GciStoreOops( theObject, GciFetchSize( theObject)+1, theOops, numOops);
```

If the collection is an NSC, this is equivalent to:

```
GciAddOopsToNsc( theObject, theOops, numOops);
```

If the object is neither indexable nor an NSC, an error is generated.

GciBegin

Begin a new transaction.

Syntax

void GciBegin( )

Description

This function begins a new transaction. If there is a transaction currently in progress, it aborts that transaction. Calling **GciBegin** is equivalent to the function call `GciExecuteStr("System beginTransaction", OOP_NIL)`.

See Also

GciAbort, page 5-20
GciExecuteStr, page 5-72
GciNbAbort, page 5-155
GciNbBegin, page 5-156
GciNbExecuteStr, page 5-167

GCI_BOOL_TO_OOP

(MACRO) Convert a C Boolean value to a GemStone Boolean object.

Syntax

OopType GCI_BOOL_TO_OOP(*aBoolean*)

Input Arguments

aBoolean	The C Boolean value to be translated into a GemStone object.

Result Value

The OOP of the GemStone Boolean object that is equivalent to *aBoolean*.

Description

This macro translates a C Boolean value into the equivalent GemStone Boolean object. A C value of 0 translates to the GemStone Boolean object *false* (represented in your C program as OOP_FALSE). Any other C value translates to the GemStone Boolean object *true* (represented as OOP_TRUE). For more information, see Appendix A, “Reserved OOPs.”

Example

```
#define FALSE 0
#define TRUE 1

OopType theOop;

theOop = GCI_BOOL_TO_OOP(TRUE);
```

See Also

GciOopToBool, page 5-203

GciCallInProgress

Determine if a GemBuilder call is currently in progress.

Syntax

BoolType GciCallInProgress( )

Return Value

This function returns TRUE if a GemBuilder call is in progress, and FALSE otherwise.

Description

This function is intended for use within interrupt handlers. It can be called any time after **GciInit**.

GciCallInProgress returns FALSE if the process is currently executing within a user action and the user action's code is not within a GemBuilder call. It considers the highest (most recent) call context only.

See Also

GciInUserAction, page 5-133

GciCheckAuth

Gather the current authorizations for an array of database objects.

Syntax

void GciCheckAuth(*oopArray*, *arraySize*, *authCodeArray*)
const OopType *oopArray*[ ];
ArraySizeType *arraySize*;
unsigned char *authCodeArray*[ ];

Input Arguments

oopArray An array of OOPs of objects for which the user's authorization level. is to be ascertained. The caller must provide these values.

arraySize The number of OOPs in *oopArray*.

Result Arguments

authCodeArray The resulting array, having at least *arraySize* elements, in which the authorization values of the objects in *oopArray* are returned as 1-byte integer values.

Description

GciCheckAuth checks the current user's authorization for each object in *oopArray* up to *arraySize*, returning each authorization code in the corresponding element of *authCodeArray*. The calling context is responsible for allocating enough space to hold the results.

Authorization levels are:

1. No authorization
2. Read authorization
3. Write authorization

Special objects, such as instances of SmallInteger, are reported as having read authorization.

Authorization values returned are those that have been committed to the database; they do not reflect changes you might have made in your local workspace. To query the local workspace, send an authorization query message to a particular segment using the **GciSendMsg** function.

If any member of *oopArray* is not a legal OOP, **GciCheckAuth** generates the error OBJ_ERR_DOES_NOT_EXIST. In that case, the contents of *authCodeArray* are undefined.

GCI_CHR_TO_OOP

(MACRO) Convert a C character value to a GemStone Character object.

Syntax

OopType GCI_CHR_TO_OOP(*aChar*)

Input Arguments

aChar	The C character value to be translated into a GemStone object.

Result Value

The OOP of the GemStone Character object that is equivalent to *aChar*.

Description

This macro translates a C character value into the equivalent GemStone Character object. For more information, see Appendix A, "Reserved OOPs."

Example

```
OopType theOop;

theOop = GCI_CHR_TO_OOP('a');
```

See Also

GciOopToChr, page 5-205

GciClampedTrav

Traverse an array of objects, subject to clamps.

Syntax

BoolType GciClampedTrav(*theOops*, *numOops*, *travBuff*, *level*)
const OopType * *theOops*;
ArraySizeType *numOops*;
struct * *travArgs*;

Input Arguments

theOops An array of OOPs representing the objects to traverse.

numOops The number of elements in *theOops.*

travArgs Pointer to a **GciClampedTravArgsSType** structure containing the following input argument fields:

OopType *clampSpec*
The OOP of the Smalltalk ClampSpecification to be used, or OOP_NIL, if the traversal is to operate without clamping. Refer to the *GemStone Kernel Reference* for a description of ClampSpecification.

long *level*
Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in *theOops.* When the level is 2, an object report is also obtained for the instance variables of each level-1 object. When the level is 0, the number of levels in the traversal is not restricted.

long *retrievalFlags*
If (*retrievalFlags* & GCI_RETRIEVE_EXPORT != 0) then OOPs of non-special objects for which an object report header is returned in the traversal buffer are automatically added to the SaveObjectsSet (see **GciSaveObjs**). The value of *retrievalFlags* should be given by using the following GemBuilder mnemonics:

GCI_RETRIEVE_DEFAULT
GCI_RETRIEVE_EXPORT
GCI_CLEAR_EXPORT causes the traversal to clear the export set before it adds any OOPs to the traverse buffer.

Result Arguments

travArgs — Pointer to a **GciClampedTravArgsSType** structure containing the following result argument field:

ByteType *	*travBuff* The buffer for the results of the traversal. The first element placed in the buffer is the *actualBufferSize*, a long integer that indicates how many bytes were actually stored in the buffer by this function. The remainder of the traversal buffer consists of a series of object reports, each of which is of type **GciObjRepSType**. You can use the macro GCI_IS_REPORT_CLAMPED to find out if a given object report represents a clamped object. If the report array would be empty, a single object report is created for the object nil.

Return Value

Returns FALSE if the traversal is not yet completed. Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal** (that is, an object report was constructed for each object, minus the special objects).

Description

The **GciClampedTrav** function initiates a traversal of the specified objects, subject to the clamps in the specified ClampSpecification. In order to guarantee that the root object of the traversal will always have an entry in the traversal buffer, the root object is not subject to the specified clamps. Refer to “GciTraverseObjs” on page 5-307 for a detailed discussion of object traversal.

See Also

GCI_IS_REPORT_CLAMPED, page 5-137
GciMoreTraversal, page 5-152
GciSaveObjs, page 5-254

GciClampedTraverseObjs

Traverse an array of objects, subject to clamps.

Syntax

```
BoolType GciClampedTraverseObjs(clampSpec, theOops, numOops, travBuff, level)
    OopType            clampSpec;
    const OopType      theOops[ ];
    ArraySizeType      numOops;
    ByteType           travBuff[ ];
    long               level;
```

Input Arguments

clampSpec	The OOP of the Smalltalk ClampSpecification to be used. Refer to the *GemStone Kernel Reference* for a description of ClampSpecification.
theOops	An array of OOPs representing the objects to traverse.
numOops	The number of elements in *theOops.*
level	Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in *theOops*. When level is 2, an object report is also obtained for the instance variables of each level-1 object. When level is 0, the number of levels in the traversal is not restricted.
travBuff	The buffer for the results of the traversal. The first element placed in the buffer is the *actualBufferSize*, a long integer that indicates how many bytes were actually stored in the buffer by this function. The remainder of the traversal buffer consists of a series of object reports, each of which is of type **GciObjRepSType**. You can use the macro GCI_IS_REPORT_CLAMPED to find out if a given object report represents a clamped object. If the report array would be empty, a single object report is created for the object *nil*.

Return Value

Returns FALSE if the traversal is not yet completed. Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal** (that is, an object report was constructed for each object, minus the special objects).

Description

The **GciClampedTraverseObjs** function initiates a traversal of the specified objects, subject to the clamps in the specified ClampSpecification. If you specify OOP_NIL as the *clampSpec* parameter, the function behaves identically to **GciTraverseObjs**. In order to guarantee that the root object of the traversal will always have an entry in the traversal buffer, the root object is not subject to the specified clamps. Refer to the **GciTraverseObjs** function for a detailed discussion of object traversal.

GciClampedTraverseObjs provides automatic byte swizzling for binary floats.

See Also

GCI_IS_REPORT_CLAMPED, page 5-137
GciTraverseObjs, page 5-307
GciNbClampedTraverseObjs, page 5-158
GciNbTraverseObjs, page 5-183

GciClassMethodForClass

Compile a class method for a class.

Syntax

void GciClassMethodForClass(*source*, *oclass*, *category*, *symbolList*)
OopType *source*;
OopType *oclass*;
OopType *category*;
OopType *symbolList*;

Input Arguments

source The OOP of a Smalltalk string to be compiled as a class method.

oclass The OOP of the class with which the method is to be associated.

category The OOP of a Smalltalk string, which contains the name of the category to which the method is added. If the category is nil (OOP_NIL), the compiler will add this method to the category "`(as yet unclassified)`".

symbolList The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). Smalltalk resolves symbolic references in source code by using symbols that are available from *symbolList*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Description

This function compiles a class method for the given class. You may not compile any method whose selector begins with an underscore (_) character. Such selectors are reserved for use by the GemStone development team as private methods.

In addition, the Smalltalk virtual machine optimizes a small number of selectors. You may not compile any methods with any of those selectors. See the *GemStone Programming Guide* for a list of the optimized selectors.

To *remove* a class method, use **GciExecuteStr** instead.

Example

```
OopType oClass;
OopType oCateg;
OopType oClassMeth;

/* Intervening code goes here, in place of this comment */

oCateg = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(oCateg, 1L, category, strlen(category));
oClassMeth = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(oClassMeth, 1L, methodText, strlen(methodText));

GciClassMethodForClass(oClassMeth, oClass, oCateg, OOP_NIL);
```

See Also

GciInstMethodForClass, page 5-130

GciClassNamedSize

Find the number of named instance variables in a class.

Syntax

int GciClassNamedSize(*oclass*)
    OopType *oclass*;

Input Arguments

oclass	The OOP of the class from which to obtain information about instance variables. Appendix A, “Reserved OOPs,” lists the OOP of each Smalltalk kernel class.

Return Value

Returns the number of named instance variables in the class. In case of error, this function returns zero.

Description

This function returns the number of named instance variables for the specified class, including those inherited from superclasses.

Example

```
int   cNumIVars;
OopType oClass;

/* Intervening code goes here, in place of this comment */

cNumIVars = GciClassNamedSize(oClass);
```

See Also

GciIvNameToIdx, page 5-140

GciClearStack

Clear the Smalltalk call stack.

Syntax

```
void GciClearStack(process)
    OopType                process;
```

Input Arguments

process	The OOP of a Process object (obtained as the value of the *context* field of an error report returned by **GciErr**).

Description

Whenever a session executes a Smalltalk expression or sequence of expressions, the virtual machine creates and maintains a call stack that provides information about its state of execution. The call stack includes an ordered list of activation records related to the methods and blocks that are currently being executed.

If a soft break or an unexpected error occurs, the virtual machine suspends execution, creates a Process object, and raises an error. The Process object represents both the call stack when execution was suspended and any information that the virtual machine needs to resume execution. If there was no fatal error, your program can call **GciContinue** to resume execution. Call **GciClearStack** instead if there was a fatal error, or if you do not want your program to resume the suspended execution.

Example

The following example presents an application whose control loop determines whether to call **GciExecute** (to process a new user query), **GciContinue** (to resume an interrupted query), or **GciClearStack** (to discard an interrupted query).

```
/* interrupt handler used to interrupt a query */
static BoolType executingCode = FALSE;
static void sigIntHandler {
	if (executingCode) {
		GciSoftBreak();   /* suspends Smalltalk execution */
		/* now do whatever application cleanup is needed */
	}
}
main () {
	/* outer control loop; uses no special error handler */
	static struct sigvec oldSig,newSig;
	GciErrSType  errReport;
	struct {
		cmdType   id; /* an enumerated type */
		cmdArgsType args; /* whatever the args are */
		} applCmd;
	char     cmdCode[300];
	errReport.context = OOP_NIL;
	/* install the interrupt handler */
	newSig.sv_mask = -1;
	newSig.sv_onstack = 0;
	newSig.sv_handler = sigIntHandler;
	sigvec(SIGINT, &newSig, &oldSig);
	while (TRUE) {
		getCommand(&applCmd);
		switch (applCmd.id) {
			case USER_LOGOUT:
				GciLogout();
				return;     /* exit application */
			case USER_QUERY:  /* initiate query from user input */
				parseUserQuery(&applCmd.args,cmdCode);
				executingCode = TRUE;
				GciExecute(cmdCode, OOP_NIL);
				executingCode = FALSE;
				break;
```

```
                case USER_RESUME:
                    /* resume most recent interrupted query */
                    if (errReport.context != OOP_NIL) {
                        executingCode = TRUE;
                        GciContinue(errReport.context);
                        executingCode = FALSE;
                    }
                    break;
                case USER_DISCARD:
                    /* discard most recent interrupted query */
                    if (errReport.context != OOP_NIL) {
                        GciClearStack(errReport.context);
                    }
                    break;
                /* other cases */
            } /* end switch */
            if ( GciErr(&errReport) &&
                 (errReport.number != RT_ERR_SOFT_BREAK) ) {
                printErrorMessage(&errReport);
            }
        } /* end while */
} /* end main */
```

See Also

GciContinue, page 5-49
GciSoftBreak, page 5-265

GciCommit

Write the current transaction to the database.

Syntax

BoolType GciCommit( )

Return Value

Returns TRUE if the transaction committed successfully. Returns FALSE if the transaction fails to commit due to a concurrency conflict or in case of error.

Description

The **GciCommit** function attempts to commit the current transaction to the GemStone database.

GciCommit ignores any commit pending action that may be defined in the current GemStone session state.

Example

```
GciErrSType myError;

/* Call GciCommit and see if there was an error */
if ( !GciCommit() || GciErr(&myError))
printf(
    "GemStone returned error %d when attempting to
      commit.\n",       myError.number);
```

See Also

GciAbort, page 5-20
GciCheckAuth, page 5-34
GciNbAbort, page 5-155
GciNbCommit, page 5-160

GciContinue

Continue code execution in GemStone after an error.

Syntax

```
OopType GciContinue(process)
    OopType             process;
```

Input Arguments

process	The OOP of a Process object (obtained as the value of the *context* field of an error report returned by **GciErr**).

Return Value

Returns the OOP of the result of the Smalltalk code that was executed. Returns OOP_NIL in case of error.

Description

The **GciContinue** function attempts to continue Smalltalk execution sometime after it was suspended. It is most useful for proceeding after GemStone encounters a pause message, a soft break (**GciSoftBreak**), or an application-defined error, since continuation is always possible after these events. Because **GciContinue** calls the virtual machine, the application user can also issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 1-14.

It may also be possible to continue Smalltalk execution if the virtual machine detects a nonfatal error during a **GciExecute**..., **GciSendMsg**, or **GciPerform** call. You may then want to use structural access functions to investigate (or modify) the state of the database before you call **GciContinue**.

Example

See the example for the **GciClearStack** function on page 5-45.

See Also

GciClearStack, page 5-45
GciErr, page 5-66
GciExecute, page 5-68
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciSendMsg, page 5-255

GciContinueWith

Continue code execution in GemStone after an error.

Syntax

OopType GciContinueWith (*process, replaceTopOfStack, args, error*)
    OopType *process*;
    OopType *replaceTopOfStack*;
    long *flags*;
    GciErrSType * *error*;

Input Arguments

process	The OOP of a Process object (obtained as the value of the *context* field of an error report returned by **GciErr**).
replaceTopOfStack	If not OOP_ILLEGAL, replace the top of the Smalltalk evaluation stack with this value before continuing. If OOP_ILLEGAL, the evaluation stack is not changed.
flags	Flags to disable or permit asynchronous events and debugging in Smalltalk, as defined for **GciPerformNoDebug**.
error	If not NULL, continue with an error. This argument takes precedence over *replaceTopOfStack*.

Return Value

Returns the OOP of the result of the Smalltalk code that was executed. In case of error, this function returns OOP_NIL.

Description

This function is a variant of the **GciContinue** function, except that it allows you to modify the call stack and the state of the database before attempting to continue the suspended Smalltalk execution. This feature is typically used while implementing a Smalltalk debugger.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciNbContinueWith, page 5-162
GciNbExecute, page 5-165
GciPerformNoDebug, page 5-218

GciCreateByteObj

Create a new byte-format object.

Syntax

```
OopType GciCreateByteObj(oclass, objId, values, numValues, clusterId, makePermanent)
    OopType            oclass;
    OopType            objId;
    const ByteType *   values;
    ArraySizeType      numValues;
    long               clusterId;
    BoolType           makePermanent;
```

Input Arguments

oclass	The OOP of the class of the new object.
objId	The new object's OOP (obtained from **GciGetFreeOop**), or OOP_ILLEGAL.
values	Array of instance variable values.
numValues	Number of elements in values.
clusterId	ID of the cluster bucket in which to place the object.
makePermanent	Flag indicating whether the object is to be permanent or temporary. If *makePermanent* is FALSE, the object is created in temporary object space and the garbage collector will make the object permanent only if the object is or becomes referenced by another permanent object. If *makePermanent* is TRUE, the object is immediately created as a permanent object, thus providing a performance gain by bypassing the garbage collector.

Return Value

GciCreateByteObj returns the OOP of the object it creates. The return value is the same as *objId* unless that value is OOP_ILLEGAL, in which case **GciCreateByteObj** assigns and returns a new OOP itself.

Description

Creates a new object using an object identifier (*objId*) previously obtained from **GciGetFreeOop** or **GciGetFreeOops**. This function provides one means of resolving unresolved forward references. See the **GciGetFreeOop** function on page 5-120 for more information on forward references.

Values are stored into the object starting at the first named instance variable (if any) and continuing to the unnamed (indexed or unordered) variables if *oclass* is indexable or NSC.

If *oclass* is an indexable or NSC class, then *numValues* may be as large or as small as desired. If *oclass* is neither indexable nor NSC, then *numValues* must not exceed the number of named instance variables in the class. If *numValues* is less than number of named instance variables, the size of the object is set to the number of named instance variables and instance variables beyond *numValues* are initialized to zero.

For an indexable object, if *numValues* is greater than zero and *values* is NULL, then the object will be created of size *numValues*, and will be initialized to logical size *numValues*. This is equivalent to `new: aSize` for classes Array or String. Using this approach, you can avoid allocating a buffer of size *numValues* and simply allow **GciCreateByteObj** to initialize all indexed instance variables to the default value of zero.

GciCreateByteObj provides automatic byte swizzling for binary floats. If *objId* is a binary float, then *numValues* must be the actual size for *oclass*. If it is not, then **GciCreateByteObj** raises an error as a safety check.

See Also

GciCreateOopObj, page 5-55
GciGetFreeOop, page 5-120
GciGetFreeOops, page 5-122

GciCreateOopObj

Create a new pointer-format object.

Syntax

OopType GciCreateOopObj(*oclass*, *objId*, *values*, *numValues*, *clusterId*, *makePermanent*)

OopType	*oclass*;
OopType	*objId*;
const OopType *	*values*;
ArraySizeType	*numValues*;
long	*clusterId*;
BoolType	*makePermanent*;

Input Arguments

oclass	The OOP of the class of the new object.
objId	The new object's OOP (obtained from **GciGetFreeOop**), or OOP_ILLEGAL.
values	Array of instance variable values.
numValues	Number of elements in values.
clusterId	ID of the cluster bucket in which to place the object.
makePermanent	Flag indicating whether the object is to be permanent or temporary. If *makePermanent* is FALSE, the object is created in temporary object space and the garbage collector will make the object permanent only if the object is or becomes referenced by another permanent object. If *makePermanent* is TRUE, the object is immediately created as a permanent object, thus providing a performance gain by bypassing the garbage collector.

Return Value

GciCreateOopObj returns the OOP of the object it creates. The return value is the same as *objId* unless that value is OOP_ILLEGAL, in which case **GciCreateOopObj** assigns and returns a new OOP itself.

Description

Creates a new object using an object identifier (*objId*) previously obtained from **GciGetFreeOop** or **GciGetFreeOops**. This function provides one means of resolving unresolved forward references. See “GciGetFreeOop” on page 5-120 for more information on forward references.

Values are stored into the object starting at the first named instance variable (if any) and continuing to the unnamed (indexed or unordered) instance variables if *oclass* is indexable or NSC. Values may be forward references to objects whose identifiers have been allocated with **GciGetFreeOop**, but for which the objects have not yet been created with **GciCreate**. The caller must initialize any unused elements of *values* to OOP_NIL.

If *oclass* is an indexable or NSC class, then *numValues* may be as large or as small as desired. If *oclass* is neither indexable nor NSC, then *numValues* must not exceed the number of named instance variables in the class. If *numValues* is less than number of named instance variables, the size of the object is set to the number of named instance variables and instance variables beyond *numValues* are initialized to OOP_NIL.

For an indexable object, if *numValues* is greater than zero and *values* is NULL, then the object will be created of size *numValues*, and will be initialized to logical size *numValues*. This is equivalent to `new: aSize` for classes Array or String. Using this approach, you can avoid allocating a buffer of size *numValues* and simply allow **GciCreateOopObj** to initialize all indexed instance variables to the default value of OOP_NIL.

See Also

GciCreateByteObj, page 5-53
GciGetFreeOop, page 5-120
GciGetFreeOops, page 5-122

GciCTimeToDateTime

Convert a C date-time representation to GemStone's.

Syntax

```
BoolType GciCTimeToDateTime(arg, result)
    time_t                  arg;
    GciDateTimeSType *      result;
```

Input Arguments

arg The C time value to be converted.

Result Arguments

result A pointer to the C struct in which to place the converted value.

Return Value

Returns TRUE if the conversion succeeds; otherwise returns FALSE.

Description

Converts a **time_t** value to **GciDateTimeSType**. On systems where **time_t** is a signed value, **GciCTimeToDateTime** generates an error if *arg* is negative.

GciDateTimeToCTime

Convert a GemStone date-time representation to C's.

Syntax

time_t GciDateTimeToCTime(*arg*)
const GciDateTimeSType **arg*;

Input Arguments

arg	An instance of **GciDateTimeSType** to be converted.

Return Value

A C time value of type **time_t**.

Description

Converts an instance of **GciDateTimeSType** to the equivalent **time_t** value.

GciDbgEstablish

Specify the debugging function for GemBuilder to execute before most calls to GemBuilder functions.

Syntax

GciDbgFuncType * GciDbgEstablish(*newDebugFunc*)
    GciDbgFuncType *   *newDebugFunc*;

Input Arguments

newDebugFunc	A pointer to a C function that will be called before each subsequent GemBuilder call. Note that this function will not be called before any of the following GemBuilder functions or macros: GCI_ALIGN, GCI_BOOL_TO_OOP, GCI_CHR_TO_OOP, GCI_IS_REPORT_CLAMPED, GCI_VALUE_BUFF, **GciErr**, or **GciDbgEstablish** itself.

The **newDebugFunc** function is passed a single null-terminated string argument, (of type `const char []`), the name of the GemBuilder function about to be called.

Return Value

Returns a pointer to the *newDebugFunc* specified in the previous **GciDbgEstablish** call (if any).

Description

This function establishes the name of a C function (most likely a debugging routine) to be called before your program calls any GemBuilder function or macro (except those named above). Before each GemBuilder call, a single argument, a null-terminated string that names the GemBuilder function about to be executed, is passed to the specified *newDebugFunc*.

To disable previous debugging routines, your program can use the following statement:

```
GciDbgEstablish(NULL);
```

Example

```
void traceGciFunct(aMsg)
char aMsg[];  /* the name of the GemBuilder function about to be
called */

{
printf("%s - Call traced using GciDbgEstablish", aMsg);
}

GciDbgEstablish(traceGciFunct);
```

See Also

GciErr, page 5-66

GciDirtyObjsInit

Begin tracking which objects in the session workspace change.

Syntax

void GciDirtyObjsInit( )

Description

GemStone can track which objects in a session change, but doing so has a measurable cost. By default, GemStone does not do it. The **GciDirtyObjsInit** function permits an application to request GemStone to maintain that set of dirty objects when it is needed. Once initialized, GemStone tracks dirty objects until **GciLogout** is executed.

GciDirtyObjsInit must be called before **GciDirtySaveObjs** in order for those functions to operate properly, because they depend upon GemStone's set of dirty objects.

An object is considered dirty (changed) under one or more of the following conditions:

- A Smalltalk message was sent to the object
- The object was newly created by calling one of the **GciCreate**... or **GciNew**... functions
- The object in memory was modified by structural access in a call to one of the **GciStore**... functions
- A change to the object was committed by another transaction since it was read into this one
- The object is persistent, but was modified in a transaction of this session that was aborted (which implies that the modifications were destroyed, thus changing the state of the object in memory)

See Also

GciDirtySaveObjs, page 5-62

GciDirtySaveObjs

Find all objects in the export set that have changed since the last changes were found.

Syntax

BoolType GciDirtySaveObjs(*theOops*, *numOops*)
    OopType   *theOops*[ ];
    ArraySizeType *   *numOops*;

Input Arguments

numOops	The number of objects that can be put into *theOops* buffer.

Result Arguments

theOops	An array of the dirty cached objects found.
numOops	The number of dirty cached objects found.

Return Value

This function returns a C Boolean value indicating whether or not the complete set of dirty objects has been returned in *theOops* in one or more calls. TRUE indicates that the complete set has been returned, and FALSE indicates that it has not.

Description

GciDirtySaveObjs finds all objects in the session workspace that are in the export set and have also changed since either **GciDirtySaveObjs** or **GciDirtyObjsInit** was last called. An object is considered dirty (changed) under one or more of the following conditions:

- A Smalltalk message was sent to the object.
- The object was changed by a call to any GemBuilder function from within a user action.
- The object was changed by a call to one or more of the following functions: **GciStorePaths**, **GciSymDictAtObjPut**, **GciSymDictAtPut**, **GciStrKeyValueDictAtObjPut**, or **GciStrKeyValueDictAtPut**.
- A change to the object was committed by another transaction since it was read by this one.
- The object is persistent, but was modified in the current session before the session aborted the transaction. (When the transaction is aborted, the modifications are destroyed, thus changing the state of the object in memory).

GciDirtySaveObjs must be called only sometime after **GciDirtyObjsInit** has been executed, because it depends upon GemStone's set of dirty objects. The user is expected to call **GciDirtySaveObjs** repeatedly while it returns FALSE, until it finally returns TRUE. When **GciDirtySaveObjs** returns TRUE, it first clears the set of dirty objects.

See Also

"Garbage Collection" on page 1-31
GciDirtyObjsInit, page 5-61
GciReleaseAllOops, page 5-238
GciSaveObjs, page 5-254

GciEnableSignaledErrors

Establish or remove GemBuilder visibility to signaled errors from GemStone.

Syntax

BoolType GciEnableSignaledErrors(*newState*)
    BoolType *newState*;

Input Arguments

newState The new state of signaled error visibility: TRUE for visible.

Return Value

This function returns TRUE if signaled errors are already visible when it is called.

Description

GemStone permits selective response to signal errors: RT_ERR_SIGNAL_ABORT, RT_ERR_SIGNAL_COMMIT, and RT_ERR_SIGNAL_GEMSTONE_SESSION. The default condition is to leave them all invisible. GemStone responds to each single kind of signal error only after an associated method of class System has been executed: `enableSignaledAbortError` , `enableSignaledObjectsError` , and `enableSignaledGemStoneSessionError` respectively.

After **GciInit** executes successfully, the GemBuilder default condition also leaves all signal errors invisible. The **GciEnableSignaledErrors** function permits GemBuilder to respond automatically to signal errors. However, GemStone must respond to each kind of error in order for GemBuilder to respond to it. Thus, if an application calls **GciEnableSignaledErrors** with *newState* equal to TRUE, then GemBuilder responds automatically to exactly the same kinds of signal errors as GemStone. If GemStone has not executed any of the appropriate System methods, then this call has no effect until it does.

When enabled, GemBuilder checks for signal errors at the start of each function that accesses the database. It treats any that it finds just like any other errors, through **GciErr** or the **longjmp** mechanism, as appropriate.

Automatic checking for signalled errors incurs no extra runtime cost. The check is optimized into the check for a valid session. However, instead of checking automatically, these errors can be polled by calling the **GciPollForSignal** function.

GciEnableSignaledErrors may be called before calling **GciLogin**.

See Also

GciErr, page 5-66
GciPollForSignal, page 5-227

GciErr

Prepare a report describing the most recent GemBuilder error.

Syntax

```
BoolType GciErr(errorReport)
    GciErrSType *            errorReport;
```

Result Arguments

errorReport	Address of a GemBuilder error report structure.

Return Value

TRUE indicates that an error has occurred. The *errorReport* parameter has been modified to contain the latest error information, and the internal error buffer in GemBuilder has been cleared. You can only call **GciErr** once for a given error. If **GciErr** is called a second time, the function returns FALSE.

FALSE indicates no error occurred, and the contents of *errorReport* are unchanged.

Description

Your application program can call **GciErr** to determine whether or not the previous GemBuilder function call resulted in an error. If an error has occurred, this function provides information about the error and about the state of the GemStone system. In the case of a fatal error, your connection to GemStone is lost, and the current session ID (from **GciGetSessionId**) is reset to GCI_INVALID_SESSION_ID.

The **GciErr** function is especially useful when error traps are disabled or are not present. See "GciPopErrJump" on page 5-229 for information about using general-purpose error traps in GemBuilder. The section "The Error Report Structure" on page 5-12 describes the C structure for error reports.

Example

```
BoolType  previous;
GciErrSType err;

previous = GciSetErrJump(FALSE); /* disable error jumps */

/* Intervening code goes here, in place of this comment.
  That code makes a GemBuilder function call at its end. */

if (GciErr(&err)) {
if ((err.category == OOP_GEMSTONE_ERROR_CAT) &&
    (err.number == ErrMnem1))
   { /* do something */ }
else if ((err.category == OOP_GEMSTONE_ERROR_CAT) &&
          (err.number == ErrMnem2))
   { /* do something */ }
else
   { /* do something */ }
}
else
{ /* do something */ }

GciSetErrJump(previous);
/* reset error jumps to previous condition */
```

See Also

GciClearStack, page 5-45
GciContinue, page 5-49
GciExecute, page 5-68
GciPopErrJump, page 5-229
GciSendMsg, page 5-255

GciExecute

Execute a Smalltalk expression contained in a String object.

Syntax

OopType GciExecute(*source*, *symbolList*)
    OopType *source*;
    OopType *symbolList*;

Input Arguments

source	The OOP of a String containing a sequence of one or more statements to be executed.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. This is roughly equivalent to executing the body of a nameless procedure (method).

In most cases, you may find it more efficient to use **GciExecuteStr**. That function takes a C string as its argument, thus reducing the number of network round-trips required to execute the code. With **GciExecute**, you must first convert the source to a String object (see the following example.) If the source is already a String object, however, **GciExecute** will be more efficient.

Because **GciExecute** calls the virtual machine, the user can issue a soft break while this function is executing. For more information, see "Interrupting GemStone Execution" on page 1-14.

Example

```
strcpy (getSize, " ^ myObject size ");
oString = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(oString, 1L, getSize, (long)strlen(getSize));
oResponse = GciExecute(oString,OOP_NIL);
```

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecuteFromContext, page 5-70
GciExecuteStr, page 5-72
GciExecuteStrFromContext, page 5-74
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbExecuteStr, page 5-167
GciNbExecuteStrFromContext, page 5-169
GciSendMsg, page 5-255

GciExecuteFromContext

Execute a Smalltalk expression contained in a String object as if it were a message sent to another object.

Syntax

```
OopType GciExecuteFromContext(source, contextObject, symbolList)
    OopType          source;
    OopType          contextObject;
    OopType          symbolList;
```

Input Arguments

source	The OOP of a String containing a sequence of one or more statements to be executed.
contextObject	The OOP of any GemStone object.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. The source is executed as though *contextObject* were the receiver. That is, the pseudo-variable *self* will have the value *contextObject* during the execution. Messages in the source are executed as defined for *contextObject*.

For example, if *contextObject* is an instance of Association, the *source* can reference the pseudo-variables *key* and *value* (referring to the instance variables of the Association *contextObject*). If any pool dictionaries were available to Association, the *source* could reference them too.

In most cases, you may find it more efficient to use **GciExecuteStrFromContext**. That function takes a C string as its argument, thus reducing the number of network round-trips required to execute the code. With **GciExecuteFromContext**, you must first convert the source to a String object (see the following example.) If the source is already a String object, however, **GciExecuteFromContext** will be more efficient.

Because **GciExecuteFromContext** calls the virtual machine, the user can issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 1-14.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciExecuteStr, page 5-72
GciExecuteStrFromContext, page 5-74
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbExecuteStr, page 5-167
GciNbExecuteStrFromContext, page 5-169
GciSendMsg, page 5-255

GciExecuteStr

Execute a Smalltalk expression contained in a C string.

Syntax

OopType GciExecuteStr(*source*, *symbolList*)
const char *source*[ ];
OopType *symbolList*;

Input Arguments

source	A null-terminated string containing a sequence of one or more statements to be executed.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution.

If the source is already a String object, you may find it more efficient to use **GciExecute**. That function takes the OOP of a String as its argument.

Because **GciExecuteStr** calls the virtual machine, the user can issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 1-14.

Example

```
x = GciExecuteStr("(AllUsers userWithId: 'romeo') symbolList",
                OOP_NIL);
romeotalk = GciExecuteStr("nativeLanguage", x);
```

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciExecuteFromContext, page 5-70
GciExecuteStrFromContext, page 5-74
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbExecuteStr, page 5-167
GciNbExecuteStrFromContext, page 5-169
GciSendMsg, page 5-255

GciExecuteStrFromContext

Execute a Smalltalk expression contained in a C string as if it were a message sent to an object.

Syntax

OopType GciExecuteStrFromContext(*source*, *contextObject*, *symbolList*)
    const char *source*[ ];
    OopType *contextObject*;
    OopType *symbolList*;

Input Arguments

source	A null-terminated string containing a sequence of one or more statements to be executed.
contextObject	The OOP of any GemStone object.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. The source is executed as though *contextObject* were the receiver. That is, the pseudo-variable self will have the value *contextObject* during the execution. Messages in the source are executed as defined for *contextObject*.

For example, if *contextObject* is an instance of Association, the source can reference the pseudo-variables *key* and *value* (referring to the instance variables of the Association *contextObject*). If any pool dictionaries were available to Association, the source could reference them too.

Because **GciExecuteStrFromContext** calls the virtual machine, the user can issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 1-14.

Example

```
OopType oAssoc, oResult ;
/* return the value  instance variable of this Association    */
oAssoc = GciExecuteStr(
"Globals associationAt: #UserProfileSet",OOP_NIL);
oResult = GciExecuteStrFromContext(" ^ value ", oAssoc,
OOP_NIL);
```

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciExecuteFromContext, page 5-70
GciExecuteStr, page 5-72
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbExecuteStr, page 5-167
GciNbExecuteStrFromContext, page 5-169
GciSendMsg, page 5-255

GciExecuteStrTrav

First execute a Smalltalk expression contained in a C string as if it were a message sent to an object, then traverse the result of the execution.

Syntax

BoolType GciExecuteStrTrav(*source*, *contextObject*, *symbolList*, *travArgs*)

const char	*source*[ ];
OopType	*contextObject*;
OopType	*symbolList*;
struct *	*travArgs*;

Input Arguments

source — A null-terminated string containing a sequence of one or more statements to be executed.

contextObject — The OOP of any GemStone object. A value of OOP_ILLEGAL means no context.

symbolList — The OOP of a GemStone symbol list (that is, an Array of instances of SymbolDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

travArgs — Pointer to a **GciClampedTravArgsSType** structure containing the following input argument fields:

OopType	*clampSpec* The OOP of the Smalltalk ClampSpecification to be used, or OOP_NIL, if the traversal is to operate without clamping. Refer to the *GemStone Kernel Reference* for a description of ClampSpecification.
long	*level* Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in the array of OOPs representing the objects to traverse. When level is 2, an object report is also obtained for the instance variables of each level-1 object. When level is 0,

the number of levels in the traversal is not restricted.

long *retrievalFlags*
Flags to control object retrieval. The value of *retrievalFlags* should be given by using the following GemBuilder mnemonics:
GCI_RETRIEVE_DEFAULT
GCI_RETRIEVE_EXPORT
GCI_CLEAR_EXPORT causes the traversal to clear the export set before it adds any OOPs to the traverse buffer.

Result Arguments

travArgs Pointer to a **GciClampedTravArgsSType** structure containing the following result argument field:

ByteType * *travBuff*
The buffer for the results of the traversal. The first element placed in the buffer is the *actualBufferSize*, a long integer that indicates how many bytes were actually stored in the buffer by this function. The remainder of the traversal buffer consists of a series of object reports, each of which is of type **GciObjRepSType**.

Return Value

Returns FALSE if the traversal is not yet completed. You can then call **GciMoreTraversal** to proceed. Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal**.

Description

This function is like **GciPerformTrav**, except that it first does a **GciExecuteStr** instead of a **GciPerform**.

See Also

GciExecuteStr, page 5-72
GciMoreTraversal, page 5-152
GciPerformTrav, page 5-222

GciFetchByte

Fetch one byte from an indexed byte object.

Syntax

ByteType GciFetchByte(*theObject*, *atIndex*)
    OopType *theObject*;
    long *atIndex*;

Input Arguments

theObject	The OOP of the GemStone byte object.
atIndex	The index into *theObject* of the element to be fetched. The index of the first element is 1.

Return Value

Returns the byte value at the specified index. In case of error, this function returns zero.

Description

This function fetches a single element from a byte object at the specified index, using structural access.

Example

```
OopType oString;
int   aNum;
ByteType theChar;

/* Intervening code goes here, in place of this comment */

theChar = GciFetchByte(oString, aNum);
```

See Also

GciFetchBytes, page 5-81
GciStoreByte, page 5-266
GciStoreBytes, page 5-268

GciFetchBytes

Fetch multiple bytes from an indexed byte object.

Syntax

long GciFetchBytes(*theObject*, *startIndex*, *theBytes*, *numBytes*)

OopType	*theObject*;
long	*startIndex*;
ByteType	*theBytes*[ ];
ArraySizeType	*numBytes*;

Input Arguments

theObject	The OOP of the GemStone byte object.
startIndex	The index into *theObject* at which to begin fetching bytes. (The index of the first element is 1.) Note that if *startIndex* is 1 greater than the size of the object, this function returns a byte array of size 0, but no error is generated.
numBytes	The maximum number of bytes to return.

Result Arguments

theBytes	The array of fetched bytes

Return Value

Returns the number of bytes fetched. (This may be less than *numBytes*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function fetches multiple elements from a byte object starting at the specified index, using structural access. A common application of **GciFetchBytes** would be to fetch a text string.

GciFetchBytes permits *theObject* to be a binary float, but it does not provide automatic byte swizzling. In that case, you must provide your own byte swizzling as needed.

Alternatively, you can call **GciFetchObjInfo** instead, and that function will provide any necessary byte swizzling.

Example

This example illustrates a C function that incrementally processes a GemStone String of arbitrary size, while using a limited amount of C memory space.

```
#define BUF_SIZE 5000
static char displayBuff[BUF_SIZE];

void displayByteObject(oObject)
OopType oObject;
{
int   fetchAmount;
long  longIdx;
ArraySizeType fetchIncrement;
BoolType done = FALSE;

fetchIncrement = BUF_SIZE - 1;

longIdx = 1L;
fetchAmount =
   GciFetchBytes(oObject,longIdx,displayBuff,fetchIncrement);
done = (fetchAmount == 0); /* done if object is of size zero */
while (!done) {
   displayBuff[fetchAmount] = '\0';
   printf("%s",displayBuff);
   if (fetchAmount < fetchIncrement)
      break; /* because object was shorter than previous limit */
   /* advance fetch position and perform next fetch */
   longIdx += fetchAmount;
   fetchAmount = GciFetchBytes(oObject, longIdx, displayBuff,
                               fetchIncrement);
   done = (fetchAmount == 0);  /* done if longIdx was past end */
} /* end while */
}
```

See Also

GciFetchByte, page 5-79
GciFetchObjInfo, page 5-97
GciStoreByte, page 5-266
GciStoreBytes, page 5-268

GciFetchChars

Fetch multiple ASCII characters from an indexed byte object.

Syntax

long GciFetchChars(*theObject*, *startIndex*, *cString*, *maxSize*)

OopType	*theObject*;
long	*startIndex*;
char *	*cString*;
ArraySizeType	*maxSize*;

Input Arguments

theObject	The OOP of a text object.
startIndex	The index of the first character to retrieve.
maxSize	Maximum number of characters to fetch.

Result Arguments

cString	Pointer to the location in which to store the returned string.

Return Value

Returns the number of characters fetched.

Description

Equivalent to **GciFetchBytes**, except that it is assumed that *theObject* contains ASCII text. The bytes fetched are stored in memory starting at *cString*. At most *maxSize* - 1 bytes will be fetched from the object, and a `\0` character will be stored in memory following the bytes fetched. The function returns the number of characters fetched, excluding the null terminator character, which is equivalent to `strlen(cString)` if the object does not contain any null characters. If an error occurs, the function result is 0, and the contents of *cString* are undefined.

See Also

GciFetchBytes, page 5-81

GciFetchClass

Fetch the class of an object.

Syntax

OopType GciFetchClass(*theObject*)
OopType *theObject*;

Input Arguments

theObject The OOP of the specified object.

Return Value

Returns the OOP of the object's class. In case of error, this function returns OOP_NIL.

The GemBuilder include file `gcioop.ht` defines a C constant for each of the Smalltalk kernel classes. Those C constants are listed in Appendix A, "Reserved OOPs."

Description

The **GciFetchClass** function obtains the class of an object from GemStone. The GemBuilder session must be valid when **GciFetchClass** is called, unless theObject is an instance of one of the following classes: Boolean, Character, JisCharacter, SmallInteger, or UndefinedObject.

Example

```
OopType oString;
OopType oResponse;
OopType oClass;

/* Intervening code goes here, in place of this comment */

oResponse = GciExecute(oString, OOP_NIL);
oClass = GciFetchClass(oResponse);
```

See Also

GciFetchNamedSize, page 5-92
GciFetchObjImpl, page 5-94
GciFetchSize, page 5-108
GciFetchVaryingSize, page 5-115

GciFetchDateTime

Convert the contents of a DateTime object and place the results in a C structure.

Syntax

void GciFetchDateTime(*datetimeObj*, *result*)
    OopType *datetimeObj*;
    GciDateTimeSType * *result*;

Input Arguments

datetimeObj	OOP of the object to fetch.

Result Arguments

result	C pointer to the structure for the returned object.

Description

Fetches the contents of a DateTime object into the specified C result. Generates an error if *datetimeObj* is not an instance of DateTime. The value that *result* points to is undefined if an error occurs.

GciFetchNamedOop

Fetch the OOP of one of an object's named instance variables.

Syntax

```
OopType GciFetchNamedOop(theObject, atIndex)
    OopType          theObject;
    long             atIndex;
```

Input Arguments

theObject	The OOP of the GemStone object.
atIndex	The index into *theObject*'s named instance variables of the element to be fetched. The index of the first named instance variable is 1.

Return Value

Returns the OOP of the specified named instance variable. In case of error, this function returns OOP_NIL.

Description

This function fetches the contents of an object's named instance variable at the specified index, using structural access.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType theName;

/* C constants to match Smalltalk class definition */
#define COMPONENT_OFF_PARTNUMBER 1L
#define COMPONENT_OFF_NAME    2L
#define COMPONENT_OFF_COST    3L

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

/* fetch the name instance variable of aComponent */
the Name = GciFetchNamedOop(aComponent, COMPONENT_OFF_NAME);

/* fetch named instance variable without knowing its offset at
compile time */
theName = GciFtechNamedOop(aComponent,
            GciIvNameToIdx(GciFetchClass(aComponent), "name"));
```

See Also

GciFetchNamedOops, page 5-90
GciFetchVaryingOop, page 5-110
GciFetchVaryingOops, page 5-113
GciIvNameToIdx, page 5-140
GciStoreIdxOop, page 5-274
GciStoreIdxOops, page 5-276
GciStoreNamedOop, page 5-278
GciStoreNamedOops, page 5-280

GciFetchNamedOops

Fetch the OOPs of one or more of an object's named instance variables.

Syntax

long GciFetchNamedOops(*theObject*, *startIndex*, *theOops*, *numOops*)
  OopType *theObject*;
  long *startIndex*;
  OopType *theOops*[ ];
  ArraySizeType *numOops*;

Input Arguments

theObject	The OOP of the source GemStone object.
startIndex	The index into *theObject*'s named instance variables at which to begin fetching. (The index of the first named instance variable is 1.) Note that if *startIndex* is 1 greater than the number of the object's named instance variables, this function returns an array of size 0, but no error is generated.
numOops	The maximum number of elements to return.

Result Arguments

theOops	The array of fetched OOPs.

Return Value

Returns the number of OOPs fetched. (This may be less than *numOops*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function uses structural access to fetch multiple values from an object's named instance variables, starting at the specified index.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType    aComponent;
ArraySizeType namedSize;
OopType *   oBuffer;

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
              "AllComponents select:[i|i.partnumber = 1234]");

/* fetch named instance variables without knowing how many at
compile time */
namedSize = GciFetchNamedSize(aComponent);
oBuffer = (OopType*) malloc( namedSize * sizeof(OopType));
GciFetchNamedOops(aComponent, 1L, oBuffer, namedSize);
```

See Also

GciFetchNamedOop, page 5-88
GciFetchVaryingOop, page 5-110
GciIvNameToIdx, page 5-140
GciStoreIdxOop, page 5-274
GciStoreNamedOop, page 5-278

GciFetchNamedSize

Fetch the number of named instance variables in an object.

Syntax

```
long GciFetchNamedSize(theObject)
    OopType                 theObject;
```

Input Arguments

theObject The OOP of the specified object.

Return Value

Returns the number of named instance variables in *theObject*. In case of error, this function returns zero.

Description

This function returns the number of named instance variables in a GemStone object. See the example for the **GciFetchNamedOops** function on page 5-90.

GciFetchNameOfClass

Fetch the class name object for a given class.

Syntax

OopType GciFetchNameOfClass(*aClass*)
    OopType *aClass*

Input Arguments

aClass The OOP of a class.

Return Value

The OOP of the class's name, or OOP_NIL if an error occurred.

Description

Given the OOP of a class, this function returns the object identifier of the String object that is the name of the class.

GciFetchObjImpl

Fetch the implementation of an object.

Syntax

long GciFetchObjImpl(*theObject*)
OopType *theObject*;

Input Arguments

theObject	The OOP of the specified object.

Return Value

Returns an integer representing the implementation type of *theObject* (0=pointer, 1=byte, 2=NSC, or 3=special). In case of error, the return value is undefined.

Description

This function obtains the implementation of an object (pointer, byte, NSC, special) from GemStone. For more information about implementation types, see "Direct Access to Metadata" on page 1-17.

Example

```
long  imp;
OopType theObj;

imp = GciFetchObjImpl(theObj);
```

See Also

GciFetchClass, page 5-85
GciFetchNamedSize, page 5-92
GciFetchSize, page 5-108
GciFetchVaryingSize, page 5-115

GciFetchObjectInfo

Fetch information and values from an object.

Syntax

```
BoolType GciFetchObjInfo(theObject, args)
    OopType           theObject;
    struct *          args;
```

Input Arguments

theObject	OOP of any object with byte, pointer, or NSC format.
args	Pointer to an instance of **GciFetchObjInfoArgsSType** with the following input argument fields:

long	*startIndex* The offset in the object at which to start fetching, using **GciFetchOops** or **GciFetchBytes** semantics. *startIndex* is ignored if *bufSize* == 0 or buffer == NULL.
ArraySizeType	*bufSize* The size in bytes of the buffer, maximum number of elements fetched for a byte object. For an OOP object, the maximum number of elements fetched for an OOP object will be *bufSize*/4. If greater than zero, and if a Float or BinaryFloat is being fetched, it must be large enough to fetch the complete object.
long	*retrievalFlags* If (*retrievalFlags* & GCI_RETRIEVE_EXPORT) != 0 then if *theObject* is non-special, *theObject* is automatically added to the SaveObjectsSet (see the **GciSaveObjs** function).

Result Arguments

args — Pointer to an instance of **GciFetchObjInfoArgsSType** with the following result argument fields:

Type	Field
struct *	*info* Pointer to an instance of **GciObjInfoSType**; may be NULL.
ByteType *	*buffer* Pointer to an area where byte or OOP values will be returned; may be NULL.
ArraySizeType	*numReturned* Number of logical elements (bytes or OOPs) returned in buffer. Remember that the size of (**OopType**) is 4 bytes.

If either *info* or *buffer* is NULL, that portion of the result is not filled in.

Return Value

TRUE if successful, FALSE if an error occurs.

Description

This function fetches information and values from an object starting at the specified index using structural access. If either *info* or *buffer* is NULL, then that part of the result is not filled in. If *numReturned* is NULL, then *buffer* will not be filled in.

See Also

GciFetchOops, page 5-101
GciFetchBytes, page 5-81
GciSaveObjs, page 5-254

GciFetchObjInfo

Fetch information and values from an object.

Syntax

BoolType GciFetchObjInfo(*theObject*, *startIndex*, *bufSize*, *info*, *buffer*, *numReturned*)

OopType	*theObject*;
long	*startIndex*;
ArraySizeType	*bufSize*;
GciObjInfoSType *	*info*;
ByteType *	*buffer*;
ArraySizeType *	*numReturned*;

Input Arguments

theObject	OOP of any object with byte, pointer, or NSC format.
startIndex	The index into *theObject* at which to begin fetching elements. (The index of the first element is 1.) If the start index is 1 greater than the size of the object, this function returns an array of size 0, but no error is generated.
bufSize	The size in bytes of the buffer, maximum number of elements fetched for a byte object. For an OOP object, the maximum number of elements fetched for an OOP object will be *bufSize*/4.

Result Arguments

info	Pointer to an instance of **GciObjInfoSType**; may be NULL.
buffer	Pointer to an area where byte or OOP values will be returned; may be NULL.
numReturned	Number of logical elements (bytes or OOPs) returned in buffer. Remember that the sizeof(**OopType**) is 4 bytes.

Return Value

TRUE if successful, FALSE if an error occurs. If an error occurs, *info*, *buffer*, and *numReturned* are undefined.

Description

This function fetches information and values from an object starting at the specified index using structural access. If either *info* or *buffer* is NULL, then that part of the result is not filled in. If *numReturned* is NULL, then *buffer* will not be filled in.

GciFetchObjInfo provides automatic byte swizzling for binary floats. If *theObject* is a binary float, then *startIndex* must be one and *bufSize* must be the actual size for the class of *theObject*. If either of these conditions are not met, then **GciFetchObjInfo** raises an error as a safety check.

GciFetchOop

Fetch the OOP of one instance variable of an object.

Syntax

```
OopType GciFetchOop(theObject, atIndex)
    OopType         theObject;
    long            atIndex;
```

Input Arguments

theObject	The OOP of the source object.
atIndex	The index into *theObject* of the OOP to be fetched. The index of the first OOP is 1.

Return Value

Returns the OOP at the specified index of the source object. In case of error, this function returns OOP_NIL.

Description

This function fetches the OOP of a single instance variable from any object at the specified index, using structural access. It does not distinguish between named and unnamed instance variables. Indices are based at the beginning of the object's array of instance variables. In that array, any existing named instance variables are followed by any existing unnamed instance variables.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType theName;
OopType aSubComponent;
/* C constant to match Smalltalk class definition */
#define COMPONENT_OFF_NAME 2

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

/* Two ways to fetch the name instance variable of aComponent */
theName = GciFetchOop(aComponent, COMPONENT_OFF_NAME);
theName = GciFetchNamedOop(aComponent, COMPONENT_OFF_NAME);

/* Two ways to fetch the 3rd element of aComponent's partsList,
without knowing exactly how many named instance variables exist
*/
aSubComponent =
    GciFetchOop(aComponent,GciFetchNamedSize(aComponent) + 3);
aSubComponent = GciFetchVaryingOop(aComponent, 3);
```

See Also

GciFetchOops, page 5-101
GciStoreOop, page 5-282
GciStoreOops, page 5-284

GciFetchOops

Fetch the OOPs of one or more instance variables of an object.

Syntax

long GciFetchOops(*theObject*, *startIndex*, *theOops*, *numOops*)
    OopType   *theObject*;
    long   *startIndex*;
    OopType   *theOops*[ ];
    ArraySizeType   *numOops*;

Input Arguments

theObject	The OOP of the source object.
startIndex	The index into *theObject* at which to begin fetching OOPs. The index of the first OOP is 1. If *startIndex* is 1 greater than the size of the object, this function returns an array of size 0, but no error is generated.
numOops	The maximum number of OOPs to return.

Result Arguments

theOops	The array of fetched OOPs.

Return Value

Returns the number of OOPs fetched. (This may be less than *numOops*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function fetches the OOPs of multiple instance variables from any object starting at the specified index, using structural access. It does not distinguish between named and unnamed instance variables. Indices are based at the beginning of the object's array of instance variables. In that array, any existing named instance variables are followed by any existing unnamed instance variables.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType    aComponent;
OopType    oArray[10];
ArraySizeType namedSize;

/* Retrieve a random instance of class Component */
aComponent = GciExecuteStr(
            "AllComponents select:[i|i.partnumber = 1234]");

namedSize = GciFetchNamedSize(aComponent);
/* Two ways to fetch first 5 elements of aComponent's partsList */
GciFetchOops(aComponent, namedSize + 1L, oArray, 5);
GciFetchVaryingOops(aComponent, 1L, oArray, 5);

/* Fetch the named instance variables PLUS
the first 5 elements of partsList */
GciFetchOops(aComponent, 1L, oArray, namedSize + 5);

/* oArray[0..namedSize-1] are named instVar values,
oArray[namedSize] is first indexed instVar value */
```

See Also

GciFetchOop, page 5-99
GciFetchVaryingOop, page 5-110
GciStoreOop, page 5-282
GciStoreOops, page 5-284

GciFetchPaths

Fetch selected multiple OOPs from an object tree.

Syntax

BoolType GciFetchPaths(*theOops*, *numOops*, *paths*, *pathSizes*, *numPaths*, *results*)

const OopType	*theOops*[ ];
ArraySizeType	*numOops*;
const long	*paths*[ ];
const long	*pathSizes*[ ];
ArraySizeType	*numPaths*;
OopType	*results*[ ];

Input Arguments

theOops	A collection of OOPs from which you want to fetch.
numOops	The size of *theOops.*
paths	An array of integers. This one-dimensional array contains the elements of all constituent paths, laid end to end.
pathSizes	An array of integers. Each element of this array is the length of the corresponding path in the *paths* array (that is, the number of elements in each constituent path).
numPaths	The number of paths in the *paths* array. This should be the same as the number of integers in the *pathSizes* array.

Result Arguments

results	An array containing the OOPs that were fetched.

Return Value

Returns TRUE if all desired objects were successfully fetched. Returns FALSE if the fetch on any path fails for any reason.

Description

This function allows you to fetch multiple OOPs from selected positions in an object tree with a single GemBuilder call, importing only the desired information from the database.

> *This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of* GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions instead. For more information, see "GciRpc and GciLnk" on page 2-1.*

Each path in the *paths* array is itself an array of integers. Those integers are offsets that specify a path from which to fetch objects. In each path, a positive integer *x* refers to an offset within an object's named instance variables (see **GciFetchNamedOop**), while a negative integer *-x* refers to an offset within an object's indexed instance variables (see **GciFetchVaryingOop**). The **GciStrToPath** function allows you to convert path information from its string representation, in which each element is the name of an instance variable, to the equivalent element of this paths array.

From each object in *theOops*, this function fetches the object pointed to by each element of the paths array, and stores the fetched object into the results array. The *results* array contains `(numOops * numPaths)` elements, stored in the following order:

```
[0,0]..[0,numPaths-1]..
[1,0]..[1,numPaths-1]..
[numOops-1,0]..[numOops-1,numPaths-1]
```

That is, all paths are first applied in order to the first element of *theOops*. This step is repeated for each subsequent object, until all paths have been applied to all elements of *theOops*. The result for object i and path j is represented as:

```
results[ ((i-1) * numPaths) + (j-1) ]
```

If the fetch on any path fails for any reason, the result of that fetch is reported in the results array as OOP_ILLEGAL. Because some path-fetching errors do not necessarily invalidate the remainder of the information fetched, the system will then attempt to continue its fetching with the remaining paths and objects.

This ability to complete a fetching sequence despite errors means that your application won't be slowed by a round-trip to GemStone on each fetch to check for errors. Instead, after a fetch is complete, you can cycle through the result and deal selectively at that time with any errors you find.

The appropriate response to an error in path fetching depends both upon the error itself and on your application. Here are some of the reasons why a fetch might not succeed:

- The user had no read authorization for some object in the path. The seriousness of this depends on your application. In some applications, you may simply wish to ignore the inaccessible data.
- The path was invalid for the object to which it was applied. This can happen if the object from which you're fetching is not of the correct class, or if the path itself is faulty for the class of the object.
- The path was valid but simply not filled out for the object being processed. This would be the case, for example, if you attempted to access *address.zip* when an Employee's Address instance variable contained only *nil*. This is probably the most common path fetching error, and may require only that the application program detect the condition and display some suitable indication to the user that a field is not yet filled in with meaningful data.

Examples

Example 1: Calling sequence for a single object and a single path

```
OopType anOop;   /* the OOP to use as the root of the path */
long  aPath[5]; /* the path itself */
long  aSize;   /* the size of the path */
OopType result;

GciFetchPaths (&anOop, 1, aPath, &aSize, 1, &result);
```

Example 2: Calling sequence for multiple objects with a single path

```
OopType oops[3]; /* the OOPs to use as roots of the path */
ArraySizeType numOops; /* the number of objects */
long  aPath[5]; /* the path itself */
long  aSize;   /* the size of the path */
OopType results[5];

GciFetchPaths (oops, numOops, aPath, &aSize, 1, results);
```

Example 3: Calling sequence for a single object with multiple paths

```
OopType anOop;   /* the OOP to use as the root of the path */
long  paths[50]; /* the paths, stored end-to-end in the array */
long  sizes[5]; /* the sizes of the paths */
ArraySizeType numPaths;  /* the number of paths */
OopType results[5];

GciFetchPaths (&anOop, 1, paths, sizes, numPaths, results);
```

Example 4: Calling sequence for multiple objects with multiple paths

```
OopType oops[3];  /* the OOPs to use as roots of the path */
ArraySizeType numOops;  /* the number of objects */
long  paths[50]; /* the paths, stored end-to-end in the array */
long  sizes[5]; /* the sizes of the paths */
ArraySizeType numPaths;  /* the number of paths */
OopType results[3*5];
/* results for each path for oop1, then for oop2,etc. */

GciFetchPaths (oops, numOops, paths, sizes, numPaths, results);
```

Example 5: Integrated Code

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType oSourceObjs[10];
OopType oResults[10];
OopType theName;
long  pathSizes[10];
long  paths[10];

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
             "AllComponents select:[i|i.partnumber = 1234]");

/* fetch name instVar of 5th element of aComponent's partsList */
oSourceObjs[0] = aComponent;
paths[0] = -5;
paths[1] = GciIvNameToIdx(GciFetchClass(aComponent), "name");
pathSizes[0] = 2;
GciFetchPaths(oSourceObjs, 1, paths, pathSizes, 1, oResults);
theName = oResults[0];
```

See Also

GciPathToStr, page 5-213
GciStorePaths, page 5-287
GciStrToPath, page 5-300

GciFetchSize

Fetch the size of an object.

Syntax

long GciFetchSize(*theObject*)
  OopType *theObject*;

Input Arguments

theObject The OOP of the specified object.

Return Value

Returns the size of *theObject*. In case of error, this function returns zero.

Description

This function obtains the size of an object from GemStone.

The result of this function depends on the object's implementation (see **GciFetchObjImpl**). For byte objects, this function returns the number of bytes in the object. (For Strings, this is the number of Characters in the String; for Floats, the size is 23.) For pointer objects, this function returns the number of named instance variables (**GciFetchNamedSize**) plus the number of indexed instance variables, if any (**GciFetchVaryingSize**). For NSC objects, this function returns the cardinality of the collection. For special objects, the size is always zero.

This differs somewhat from the result of executing the Smalltalk method `Object | size` , as shown in Table 5.10:

Table 5.10 Differences in Reported Object Size

Implementation	Object | size (Smalltalk)	GciFetchSize
0=Pointer	Number of indexed elements in the object (0 if not indexed)	Number of indexed elements PLUS number of named instance variables
1=Byte	Number of indexed elements in the object	Same as Smalltalk message “size”
2=NSC	Number of elements in the object	Same as Smalltalk message “size”
3=Special	0	0

Example

```
long  itsSize;
OopType oIndexedObject;

oIndexedObject = GciNewOop(OOP_CLASS_STRING);

/* Intervening code goes here, in place of this comment */

itsSize = GciFetchSize(oIndexedObject);
```

See Also

GciFetchClass, page 5-85
GciFetchNamedSize, page 5-92
GciFetchObjImpl, page 5-94
GciFetchVaryingOop, page 5-110

GciFetchVaryingOop

Fetch the OOP of one unnamed instance variable from an indexed pointer object or NSC.

Syntax

```
OopType GciFetchVaryingOop(theObject, atIndex)
    OopType          theObject;
    long             atIndex;
```

Input Arguments

theObject	The OOP of the pointer object or NSC.
atIndex	The index of the OOP to be fetched. The index of the first unnamed instance variable's OOP is 1.

Return Value

Returns the OOP of the unnamed instance variable at index *atIndex*. In case of error, this function returns OOP_NIL.

Description

This function fetches the OOP of a single unnamed instance variable at the specified index, using structural access. The numerical index of any unordered variable of an NSC can change whenever the NSC is modified.

Example

In the following example, assume that you've executed the following Smalltalk code to define the class Component and to populate the set AllComponents:

```
! Topaz command to define the class Component run
Array subclass: #Component
instVarNames: #(#partNumber #name #cost
               "indexed variables form the partsList")
classVars: #()
poolDictionaries: #()
inDictionary: UserGlobals
constraints: #[#[#partNumber, SmallInteger]
               #[#name, String],
               #[#cost, Number]]
isInvariant: false
%
run
UserGlobals at: #AllComponents put: Set new
%
run
!
populate AllComponents with other Smalltalk code
!
%
```

Now execute this C code.

```
OopType aComponent;
OopType aSubComponent;

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
            "AllComponents select:[i|i.partnumber = 1234]");
/* fetch 3rd element of aComponent's parts list */
aSubComponent = GciFetchVaryingOop(aComponent, 3);
```

See Also

GciFetchNamedOop, page 5-88
GciFetchNamedOops, page 5-90
GciFetchVaryingOops, page 5-113

GciStoreIdxOop, page 5-274
GciStoreIdxOops, page 5-276
GciStoreNamedOop, page 5-278
GciStoreNamedOops, page 5-280

GciFetchVaryingOops

Fetch the OOPs of one or more unnamed instance variables from an indexed pointer object or NSC.

Syntax

```
long GciFetchVaryingOops(theObject, startIndex, theOops, numOops)
    OopType            theObject;
     long              startIndex;
    OopType            theOops[ ];
    ArraySizeType      numOops;
```

Input Arguments

theObject	The OOP of the pointer object or NSC.
startIndex	The index of the first OOP to be fetched. The index of the first unnamed instance variable's OOP is 1. Note that if *startIndex* is 1 greater than the number of *theObject*'s unnamed instance variables, this function returns an array of size 0, but no error is generated.
numOops	Maximum number of elements to return.

Result Arguments

theOops	The array of fetched OOPs.

Return Value

Returns the number of OOPs fetched. (This may be less than *numOops*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function fetches the OOPs of multiple unnamed instance variables beginning at the specified index, using structural access. The numerical index of any unordered variable of an NSC can change whenever the NSC is modified.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType oArray[10];

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
              "AllComponents select:[i|i.partnumber = 1234]");
/* fetch the first 5 elements of aComponent's parts list */
GciFetchVaryingOops(aComponent, 1L, oArray, 5);
```

See Also

GciFetchNamedOop, page 5-88
GciFetchNamedOops, page 5-90
GciFetchVaryingOop, page 5-110
GciStoreIdxOop, page 5-274
GciStoreIdxOops, page 5-276
GciStoreNamedOop, page 5-278
GciStoreNamedOops, page 5-280

GciFetchVaryingSize

Fetch the number of unnamed instance variables in a pointer object or NSC.

Syntax

long GciFetchVaryingSize(*theObject*)
    OopType *theObject*;

Input Arguments

theObject The OOP of the specified object.

Return Value

Returns the number of unnamed instance variables in *theObject*. In case of error, this function returns zero.

Description

The **GciFetchVaryingSize** function obtains from GemStone the number of indexed variables in an indexable object or the number of unordered variables in an NSC.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
long  partsListSize;

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
                "AllComponents select:{i|i.partnumber = 1234}");

/* fetch the size of aComponent's partsList */
partsListSize = GciFetchVaryingSize(aComponent);
```

See Also

GciFetchClass, page 5-85
GciFetchNamedSize, page 5-92
GciFetchObjImpl, page 5-94
GciFetchSize, page 5-108

GciFindObjRep

Fetch an object report in a traversal buffer.

Syntax

GciObjRepHdrSType * GciFindObjRep(*travBuff*, *theObject*)
  const ByteType   *travBuff*[ ];
  OopType   *theObject*;

Input Arguments

travBuff	A traversal buffer returned by a call to **GciTraverseObjs**.
theObject	The OOP of the object to find.

Return Value

Returns a pointer to an object report within the traversal buffer. In case of error, this function returns NULL.

Description

This function locates an object report within a traversal buffer that was previously returned by **GciTraverseObjs**. If the report is not found within the buffer, this function generates the error GCI_ERR_TRAV_OBJ_NOT_FOUND.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch**... *and* **GciStore**... *functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

Example

```
theReport = GciFindObjRep(myBuff, myObj);
if (GciErr(&myErr)) {
/* Handle the error */
}
else {
/* Go ahead with your work */
}
```

See Also

GciMoreTraversal, page 5-152
GciObjRepSize, page 5-198
GciTraverseObjs, page 5-307

GciFltToOop

Convert a C double value to a Float object.

Syntax

OopType GciFltToOop(*aReal*)
double *aReal*;

Input Arguments

aReal The floating point value to be translated into an object.

Return Value

Returns the OOP of the GemStone Float object that corresponds to the C value. In case of error, this function returns OOP_NIL.

Description

This function translates a C double precision value into the equivalent GemStone Float object.

Example

```
char  unitPrice[MAXLEN + 1]; /* Price of the part */
OopType thePriceOop;      /* The OOP of a product's price */

printf("Unit Price = ");
fflush(stdout);
getString(unitPrice, MAXLEN);
thePriceOop = GciFltToOop(atof(unitPrice));
```

See Also

GciOopToFlt, page 5-207

GciGetFreeOop

Allocate an OOP.

Syntax

OopType GciGetFreeOop( )

Return Value

Returns an unused object identifier (OOP).

Description

Allocates an object identifier without creating an object.

The object identifier returned from this function remains allocated to the Gci session until the session calls **GciLogout** or until the identifier is used as an argument to a function call.

If an object identifier returned from **GciGetFreeOop** is used as a value in a **GciStore**... call before it is used as the *objId* argument of a **GciCreate**... call, then an unresolved forward reference is created in object memory. This is a reference to an object that does not yet exist. This forward reference must be satisfied by using the identifier as the *objId* argument to a **GciCreate**... call before a **GciCommit** can be successfully executed.

If **GciCommit** is attempted prior to satisfying all unresolved forward references, an error is generated and **GciCommit** returns FALSE. In this case, **GciCreate** can be used to satisfy the forward references and **GciCommit** can be attempted again. **GciAbort** removes all unsatisfied forward references from the session's object space, just as it removes any other uncommitted modifications.

As long as it remains an unresolved forward reference, the identifier returned by **GciGetFreeOop** can be used only as a parameter to the following function calls, under the given restrictions:i

- As the *objID* of the object to be created

  `GciCreateByteObj`

- As the *objID* of the object to be created, or as an element of the value buffer

  `GciCreateOopObj`

- As an element of the value buffer only

```
GciStoreOop
GciStoreOops
GciStoreIdxOop
GciStoreIdxOops
GciStoreNamedOop
GciStoreNamedOops
GciStoreTrav
GciAppendOops
GciAddOopToNsc
GciAddOopsToNsc
GciNewOopUsingObjRep
```

- As an element of *newValues* only

  `GciStorePaths`

See Also

GciCreateByteObj, page 5-53
GciCreateOopObj, page 5-55
GciGetFreeOops, page 5-122

GciGetFreeOops

Allocate multiple OOPs.

Syntax

```
void GciGetFreeOops(count, resultOops);
    long                count;
    OopType *           resultOops;
```

Input Arguments

count	The number of OOPS to allocate.

Result Arguments

resultOops	An array to hold the returned OOPs.

Return Value

Returns an unused object identifier (OOP).

Description

Allocates object identifiers without creating objects.

If an object identifier returned from **GciGetFreeOops** is used as a value in a **GciStore**... call before it is used as the *objId* argument of a **GciCreate**... call, then an unresolved forward reference is created in object memory. This is a reference to an object that does not yet exist. This forward reference must be satisfied by using the identifier as the *objId* argument to a **GciCreate**... call before a **GciCommit** can be successfully executed.

If **GciCommit** is attempted prior to satisfying all unresolved forward references, an error is generated and **GciCommit** returns false. In this case, **GciCreate** can be used to satisfy the forward references and **GciCommit** can be attempted again. **GciAbort** removes all unsatisfied forward references from the session's object space, just as it removes any other uncommitted modifications.

As long as it remains an unresolved forward reference, the identifier returned by **GciGetFreeOops** can be used only as a parameter to the following function calls, under the given restrictions:

- As the *objID* of the object to be created

  `GciCreateByteObj`

- As the *objID* of the object to be created, or as an element of the value buffer

  `GciCreateOopObj`

- As an element of the value buffer, only

  ```
  GciStoreOop
  GciStoreOops
  GciStoreIdxOop
  GciStoreIdxOops
  GciStoreNamedOop
  GciStoreNamedOops
  GciStoreTrav
  GciAppendOops
  GciAddOopToNsc
  GciAddOopsToNsc
  GciNewOopUsingObjRep
  ```

- As an element of *newValues*, only

  `GciStorePaths`

See Also

GciCreateByteObj, page 5-53
GciCreateOopObj, page 5-55
GciGetFreeOop, page 5-120

GciGetSessionId

Find the ID number of the current user session.

Syntax

GciSessionIdType GciGetSessionId( )

Return Value

Returns the session ID currently being used for communication with GemStone. Returns GCI_INVALID_SESSION_ID if there is no session ID (that is, if the application is not logged in).

Description

This function obtains the unique session ID number that identifies the current user session to GemStone. An application can have more than one active session, but only one current session.

The ID numbers assigned to your application's sessions are unique within your application, but bear no meaningful relationship to the session IDs assigned to other GemStone applications that may be executing at the same time or accessing the same database.

Example

```
GciSessionIdType SessionID;

GciLogout();

SessionID = GciGetSessionId();
if (SessionID != GCI_INVALID_SESSION_ID){
/* do something */
}
```

See Also

GciLogin, page 5-144
GciSetSessionId, page 5-263

GciHandleError

Check the previous GemBuilder call for an error.

Syntax

BoolType GciHandleError(*errorReport*)
GciErrSType * *errorReport*

Result Arguments

errorReport Pointer to a GemBuilder error report structure.

Return Value

This function returns a boolean. TRUE indicates that an error occurred, in which case *errorReport* contains the latest error information. FALSE indicates that no error occurred, in which case the contents of *errorReport* are undefined.

Description

Your application program can call **GciHandleError** to determine whether the previous GemBuilder function call resulted in an error. If an error has occurred, this function provides information about the error and about the state of the GemStone system.

For linkable GemBuilder applications, the combination of **GciHandleError** and **GciPushErrorHandler** offer performance gains over **GciErr** and **GciPushErrJump**. This function must be called (rather than **GciErr**) if **GciPushErrorHandler** has been used to register a jump buffer. If **GciPushErrorHandler** is used but you do not call **GciHandleError** to service the longjmp, as shown in the example, unpredictable results will occur.

Example

```
void main()
{

      /* Intervening code goes here, in place of this comment */

      GCI_SIG_JMP_BUF_TYPE jumpBuf1;
      GciErrSType       theGciError;

      /* Intervening code goes here, in place of this comment */

      if (!GciInit()) exit; /* add code here to back out gracefully
*/
      if (GCI_SETJMP(jumpBuf1)) {
            if ( GciHandleError(&theGciError) )
                  /* <== do FIRST THING after GCI_LONGJMP */
                  /* code here for user's own analysis of the error
*/
            else
                  printf("No error found by error handler\n");
                  /* or take some other action you prefer */
      }
      GciPushErrHandler(jumpBuf1);

      /* Intervening code goes here, in place of this comment */

      GciSetNet(...);
      GciLogin(...);

      /* Intervening code goes here, in place of this comment */

      GciLogout(...);
}
```

See Also

GciPushErrHandler, page 5-232

GciHardBreak

Interrupt GemStone and abort the current transaction.

Syntax

void GciHardBreak( )

Description

GciHardBreak sends a hard break to the current user session (set by the last **GciLogin** or **GciSetSessionId**), which interrupts Smalltalk execution.

All GemBuilder functions can recognize a hard break, so the users of your application can terminate Smalltalk execution. For example, if your application sends a message to an object (via **GciSendMsg** or **GciPerform**), and for some reason the invoked Smalltalk method enters an infinite loop, the user can interrupt the application.

In order for GemBuilder functions in your program to recognize interrupts, your program will need an interrupt routine that can call the functions **GciSoftBreak** and **GciHardBreak**. Since GemBuilder does not relinquish control to an application until it has finished its processing, soft and hard breaks must be initiated from an interrupt service routine.

If GemStone is executing when it receives the break, it replies with the error message RT_ERR_HARD_BREAK. Otherwise, it ignores the break.

If GemStone is executing any of the following methods of class Repository, then a hard break terminates the entire session, not just Smalltalk execution:

```
fullBackupTo:
restoreFromBackup(s):
markForCollection
objectAudit
auditWithLimit:
repairWithLimit:
pagesWithPercentFree
```

See Also

GciSoftBreak, page 5-265

GciInit

Initialize GemBuilder.

Syntax

BoolType GciInit( )

Return Value

The function **GciInit** returns TRUE or FALSE to indicate successful or unsuccessful initialization of GemBuilder.

Description

The **GciInit** function initializes GemBuilder. Among other things, it establishes the default GemStone login parameters.

If your C application program is linkable, you may wish to call the **GciInitAppName** function, which you must do before you call **GciInit**. After **GciInitAppName**, you *must* call **GciInit** before calling any other GemBuilder functions. Otherwise, GemBuilder behavior will be unpredictable.

When GemBuilder is initialized on Unix platforms, it establishes its own handler for SIGIO interrupts. See “Interrupt Handling in your GemBuilder Application” on page 1-27 for information on **GciInit**’s handling of interrupts and pointers on making GemBuilder, application, and third-party handlers work together,

See Also

GciInitAppName, page 5-129

GciInitAppName

Override the default application configuration file name.

Syntax

void GciInitAppName(*applicationName*, *logWarnings*)
    const char * *applicationName*;
    BoolType *logWarnings*;

Input Arguments

applicationName	The application's name, as a character string.
logWarnings	If TRUE, causes the configuration file parser to print any warnings to standard output at executable startup.

Description

The **GciInitAppName** function affects only linkable applications. It has no effect on RPC applications. If you do not call this function *before* you call **GciInit**, it will have no effect.

A linkable GemBuilder application reads a configuration file called *applicationName*`.conf` when **GciInit** is called. This file can alter the behavior of the underlying GemStone session. For complete information, please see the *GemStone System Administration Guide*.

A linkable GemBuilder application uses defaults until it calls this function (if it does) and reads the configuration file (which it always does). For linkable GemBuilder applications, the default application name is *gci*, so the default executable configuration file is `gci.conf`. The *applicationName* argument overrides the default application name with one of your choice, which causes your linkable GemBuilder application to read its own executable configuration file.

The *logWarnings* argument determines whether or not warnings that are generated while reading the configuration file are written to standard output. If your application does not call **GciInitAppName**, the default log warnings setting is FALSE. The *logWarnings* argument resets the default for your application, which is used in the absence of a LOG_WARNINGS entry in the configuration file, or until that entry is read.

GciInstMethodForClass

Compile an instance method for a class.

Syntax

void GciInstMethodForClass(*source*, *oclass*, *category*, *symbolList*)
    OopType *source*;
    OopType *oclass*;
    OopType *category*;
    OopType *symbolList*;

Input Arguments

source	The OOP of a Smalltalk string to be compiled as an instance method.
oclass	The OOP of the class with which the method is to be associated.
category	The OOP of a Smalltalk string which contains the name of the category to which the method is added. If the category is nil (OOP_NIL), the compiler will add this method to the category “as yet unclassified”.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). Smalltalk resolves symbolic references in source code using symbols that are available from *symbolList*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile symbolList`).

Description

This function compiles an instance method for the given class.

In addition, the Smalltalk virtual machine optimizes a small number of selectors. You may not compile any methods with any of those selectors. See the *GemStone Programming Guide* for a list of the optimized selectors.

To *remove* a class method, use **GciExecuteStr** instead.

Example

```
OopType oClass;
OopType oCateg;
OopType oClassMeth;

/* Intervening code goes here, in place of this comment */

oCateg = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(oCateg, 1L, category, strlen(category));
oInstMeth = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(oInstMeth, 1L, methodText, strlen(methodText));

GciInstMethodForClass(oInstMeth, oClass, oCateg, OOP_NIL);
```

See Also

GciClassMethodForClass, page 5-42

GciInstallUserAction

Associate a C function with a Smalltalk user action.

Syntax

void GciInstallUserAction(*userAction*)
    GciUserActionSType *   *userAction*;

Input Arguments

userAction   A pointer to a C structure that describes the user-written C function.

Description

This function associates a user action name (declared in Smalltalk) with a user-written C function. Your application must call **GciInstallUserAction** before beginning any GemStone sessions with **GciLogin**. This function is typically called from **GciUserActionInit**. For more information, see Chapter 3, “Writing C Functions to be Called from GemStone.”

See Also

“The User Action Information Structure” on page 5-17
GciUserActionInit, page 5-314

GciInUserAction

Determine whether or not the current process is executing a user action.

Syntax

BoolType GciInUserAction( )

Return Value

This function returns TRUE if it is called from within a user action, and FALSE otherwise.

Description

This function is intended for use within interrupt handlers. It can be called any time after **GciInit**.

GciInUserAction returns FALSE if the process is currently executing within a GemBuilder call that was made from a user action. It considers the highest (most recent) call context only.

See Also

GciCallInProgress, page 5-33

GciIsKindOf

Determine whether or not an object is some kind of a given class or class history.

Syntax

BoolType GciIsKindOf(*anObj*, *givenClass*)
    OopType   *anObj*;
    OopType   *givenClass*;

Input Arguments

anObj	The object whose kind is to be checked.
givenClass	A class or class history to compare with the object's kind.

Return Value

GciIsKindOf returns TRUE when the class of *anObj* or any of its superclasses is in the class history of *givenClass*. It returns FALSE otherwise.

Description

GciIsKindOf performs structural access that is equivalent to the `isKindOf:` method of the Smalltalk class Object. It compares *anObj*'s class and superclasses to see if any of them are in a given class history. When *givenClass* is simply a class (which is typical), **GciIsKindOf** uses *givenClass*'s class history. When *givenClass* is itself a class history, **GciIsKindOf** uses *givenClass* directly.

Since **GciIsKindOf** does consider class histories, it *can* help to support schema modification by simplifying checks on the relationship of types when they can change over time. To accomplish a similar operation without seeing the effects of class histories, use the **GciIsKindOfClass** function.

See Also

GciIsKindOfClass, page 5-135
GciIsSubclassOf, page 5-138
GciIsSubclassOfClass, page 5-139

GciIsKindOfClass

Determine whether or not an object is some kind of a given class.

Syntax

BoolType GciIsKindOfClass(*anObj*, *givenClass*)
OopType *anObj*;
OopType *givenClass*;

Input Arguments

anObj	The object whose kind is to be checked.
givenClass	A class to compare with the object's kind.

Return Value

GciIsKindOfClass returns TRUE when the class of *anObj* or any of its superclasses is *givenClass*. It returns FALSE otherwise.

Description

GciIsKindOfClass performs structural access that is equivalent to the `isKindOf:` method of the Smalltalk class Object. It compares *anObj*'s class and superclasses to see if any of them are the *givenClass*.

Since **GciIsKindOfClass** does *not* consider class histories, it *cannot* help to support schema modification. To accomplish a similar operation when the relationship of types can change over time, use the **GciIsKindOf** function.

See Also

GciIsKindOf, page 5-134
GciIsSubclassOf, page 5-138
GciIsSubclassOfClass, page 5-139

GciIsRemote

Determine whether the application is running linked or remotely.

Syntax

BoolType GciIsRemote( )

Return Value

Returns TRUE if this application is running with GciRpc (the remote procedure call version of GemBuilder). Returns FALSE if this application is running with GciLnk (that is, if GemBuilder is linked with your GemStone session).

Description

This function reports whether the current application is using the GciRpc (remote procedure call) or GciLnk (linkable) version of GemBuilder.

GCI_IS_REPORT_CLAMPED

(MACRO) Determine whether or not an object was clamped during traversal.

Syntax

GCI_IS_REPORT_CLAMPED(*theObjectReport*)

Input Arguments

theObjectReport A pointer to an object report (assumed to be type GciObjRepSType*).

Result Value

A C Boolean value. The return value is TRUE if *theObjectReport* represents an object that was clamped during object traversal, and FALSE otherwise.

Description

This macro checks *theObjectReport* to see if it represents an object that was clamped during object traversal.

See Also

GciClampedTraverseObjs, page 5-40

GciIsSubclassOf

Determine whether or not a class is a subclass of a given class or class history.

Syntax

BoolType GciIsSubclassOf(*aClass*, *givenClass*)
    OopType *aClass*;
    OopType *givenClass*;

Input Arguments

aClass	The class that is to be checked.
givenClass	A class or class history to compare with the first class.

Return Value

GciIsSubclassOf returns TRUE when *aClass* or any of its superclasses is in the class history of *givenClass*. It returns FALSE otherwise.

Description

GciIsSubclassOf performs structural access that is equivalent to the `isSubclassOf:` method of the Smalltalk class Behavior. It compares *aClass* and *aClass*'s superclasses to see if any of them are in a given class history. When *givenClass* is simply a class (which is typical), **GciIsSubclassOf** uses *givenClass*'s class history. When *givenClass* is itself a class history, **GciIsSubclassOf** uses *givenClass* directly.

Since **GciIsSubclassOf** does consider class histories, it *can* help to support schema modification by simplifying checks on the relationship of types when they can change over time. To accomplish a similar operation without seeing the effects of class histories, use the **GciIsSubclassOfClass** function.

See Also

GciIsKindOf, page 5-134
GciIsKindOfClass, page 5-135
GciIsSubclassOfClass, page 5-139

GciIsSubclassOfClass

Determine whether or not a class is a subclass of a given class.

Syntax

BoolType GciIsSubclassOf(*aClass*, *givenClass*)
    OopType *aClass*;
    OopType *givenClass*;

Input Arguments

aClass	The class that is to be checked.
givenClass	A class to compare with the first class.

Return Value

GciIsSubclassOf returns TRUE when *aClass* or any of its superclasses is *givenClass*. It returns FALSE otherwise.

Description

GciIsSubclassOfClass performs structural access that is equivalent to the `isSubclassOf:` method of the Smalltalk class Behavior. It compares *aClass* and *aClass*'s superclasses to see if any of them are the *givenClass*.

Since **GciIsSubclassOfClass** does *not* consider class histories, it *cannot* help to support schema modification. To accomplish a similar operation when the relationship of types can change over time, use the **GciIsSubclassOf** function.

See Also

GciIsKindOf, page 5-134
GciIsKindOfClass, page 5-135
GciIsSubclassOf, page 5-138

GciIvNameToIdx

Fetch the index of an instance variable name.

Syntax

int GciIvNameToIdx(*oclass*, *instVarName*)
OopType *oclass*;
const char *instVarName*[ ];

Input Arguments

oclass	The OOP of the class from which to obtain information about instance variables.
instVarName	The instance variable name to search for.

Return Value

Returns the index of *instVarName* into the array of named instance variables for the specified class. Returns 0 if the name is not found or if an error is encountered.

Description

This function searches the array of instance variable names for the specified class (including those inherited from superclasses), and returns the index of the specified instance variable name. This index could then be used as the *atIndex* parameter in the **GciFetchNamedOop** or **GciStoreNamedOop** function call.

Example

```
int vInd;

vInd = GciIvNameToIdx(OOP_CLASS_USER_PROFILE, "password");
```

See Also

GciClassNamedSize, page 5-44
GciFetchNamedOop, page 5-88
GciFetchNamedOops, page 5-90
GciStoreNamedOop, page 5-278
GciStoreNamedOops, page 5-280

GciLoadUserActionLibrary

Load an application user action library.

Syntax

```
BoolType GciLoadUserActionLibrary(uaLibraryName, mustExist, libHandlePtr, infoBuf[],
                                  infoBufSize)
    const char *        uaLibraryName[ ];
    BoolType            mustExist;
    void **             libHandlePtr;
    char                infoBuf[];
    ArraySizeType       infoBufSize;
```

Input Arguments

uaLibraryName	The name and location of the user action library file (a null-terminated string).
mustExist	A flag to make the library required or optional.
libHandlePtr	A variable to store the status of the loading operation.
infoBuf	A buffer to store the name of the user action library or an error message.
infoBufSize	The size of *infoBuf*.

Return Value

A C Boolean value. If an error occurs, the return value is FALSE , and the error message is stored in *infoBuf*, unless *infoBuf* is NULL. Otherwise, the return value is TRUE, and the name of the user action library is stored in *infoBuf*.

Description

This function loads a user action shared library at run time. If *uaLibraryName* does not contain a path, then a standard user action library search is done. The proper prefix and suffix for the current platform are added to the basename if necessary. For more information, see Chapter 3, “Writing C Functions to be Called from GemStone.”

If a library is loaded, *libHandlePtr* is set to a value that represents the loaded library, if *libHandlePtr* is not NULL. If *mustExist* is TRUE, then an error is generated if the library can

not be found. If *mustExist* is FALSE, then the library does not need to exist. In this case, TRUE is returned and *libHandlePtr* is NULL if the library does not exist and non-NULL if it exists.

See Also

GciInstallUserAction, page 5-132
GciInUserAction, page 5-133
GciUserActionInit, page 5-314

GciLogin

Start a user session.

Syntax

void GciLogin(*gemstoneUsername*, *gemstonePassword*)
    const char *gemstoneUsername*[ ];
    const char *gemstonePassword*[ ];

Input Arguments

gemstoneUsername	The user’s GemStone user name (a null-terminated string).
gemstonePassword	The user’s GemStone password (a null-terminated string).

Description

The GemStone system is much like a time-shared computer system in that the user must log in before any work may be performed. This function creates a user session and its corresponding transaction workspace.

This function uses the current network parameters (as specified by **GciSetNet**) to establish the user’s GemStone session.

Example

```
char * StoneName;
char * HostUserId;
char * HostPassword;
char * GemService;
char * gsUserName;
char * gsPassword;

StoneName  = "!tcp@alf!gemserver41";
HostUserId  = "newtoni";
HostPassword = "gravity";
GemService  = "!tcp@lichen!gemnetobject";
gsUserName  = "isaac newton";
gsPassword  = "pomme";

if (!GciInit()) exit; /* required before first GemBuilder login */

GciSetNet(StoneName, HostUserId, HostPassword, GemService);
GciLogin(gsUserName, gsPassword);
```

See Also

GciGetSessionId, page 5-124
GciLogout, page 5-146
GciSetNet, page 5-260
GciSetSessionId, page 5-263

GciLogout

End the current user session.

Syntax

void GciLogout( )

Description

This function terminates the current user session (set by the last **GciLogin** or **GciSetSessionId**), and allows GemStone to release all uncommitted objects created by the application program in the corresponding transaction workspace. The current session ID is reset to GCI_INVALID_SESSION_ID, indicating that the application is no longer logged in. (See "GciGetSessionId" on page 5-124 for more information.)

See Also

GciGetSessionId, page 5-124
GciLoadUserActionLibrary, page 5-142
GciSetSessionId, page 5-263

GCI_LONG_IS_SMALL_INT

(MACRO) Determine whether or not a long can be translated into a SmallInteger.

Syntax

GCI_LONG_IS_SMALL_INT(*aLong*)

Input Arguments

aLong	The C long value to be translated into an object. The C long must be in the range of the GemStone SmallInteger class (-2^{30} to $2^{30} - 1$ inclusive).

Result Value

A C Boolean value. The return value is TRUE if *aLong* can be represented as a small integer, and FALSE otherwise.

Description

This macro tests a long to see if *aLong* is representable as a SmallInteger.

See Also

GCI_OOP_IS_BOOL, page 5-200
GCI_OOP_IS_SMALL_INT, page 5-201
GCI_OOP_IS_SPECIAL, page 5-202

GciLongToOop

Find a GemStone object that corresponds to a C long integer.

Syntax

OopType GciLongToOop(*aLong*)
long *aLong*;

Input Arguments

aLong The C long integer to be translated into an object.

Return Value

The **GciLongToOop** function returns the OOP of a GemStone object whose value is equivalent to the C long integer value of *aLong*.

Description

The **GciLongToOop** function translates the C long integer value *aLong* into a GemStone object that has the same value.

If the value is in the range -1073741824 .. 1073741823, the resulting object is a SmallInteger. If the value is in the range -2147483648 .. -1073741825, the resulting object is a LargeNegativeInteger. If the value is in the range 1073741824 .. 2147483647, the resulting object is a LargePositiveInteger.

Example

```
OopType theZipCodeOop;

printf("Zip Code = ");
fflush(stdout);
getString(zip, MAXLEN);
theZipCodeOop = GciLongToOop(atol(zip));
```

See Also

GCI_LONG_TO_OOP, page 5-150
GciOopToLong, page 5-209
GCI_OOP_TO_LONG, page 5-211
GciUnsignedLongToOop, page 5-313

GCI_LONG_TO_OOP

(MACRO) Find a GemStone object that corresponds to a C long integer.

Syntax

OopType GCI_LONG_TO_OOP(*aLong*)

Input Arguments

aLong	The C long integer to be translated into an object.

Result Value

The **GCI_LONG_TO_OOP** macro returns the OOP of a GemStone object whose value is equivalent to the C long integer value of *aLong*.

Description

The **GCI_LONG_TO_OOP** macro translates the C long integer value *aLong* into a GemStone object that has the same value.

If the value is in the range -1073741824 .. 1073741823, the resulting object is a SmallInteger. If the value is in the range -2147483648 .. -1073741825, the resulting object is a LargeNegativeInteger. If the value is in the range 1073741824 .. 2147483647, the resulting object is a LargePositiveInteger.

Example

```
OopType theZipCodeOop;

printf("Zip Code = ");
fflush(stdout);
getString(zip, MAXLEN);
theZipCodeOop = GCI_LONG_TO_OOP(atol(zip));
```

See Also

GciLongToOop, page 5-148
GciOopToLong, page 5-209
GCI_OOP_TO_LONG, page 5-211
GciUnsignedLongToOop, page 5-313

GciMoreTraversal

Continue object traversal, reusing a given buffer.

Syntax

```
BoolType GciMoreTraversal(travBuff, travBuffSize)
    ByteType            travBuff[ ];
    ArraySizeType       travBuffSize;
```

Input Arguments

travBuffSize The number of bytes allocated to the traversal buffer.

Result Arguments

travBuff A buffer in which the results of the traversal will be placed.

Return Value

Returns FALSE if the traversal is not yet completed, but further traversal would cause the *travBuffSize* to be exceeded. If the *travBuffSize* is reached before the traversal is complete, you can continue to call **GciMoreTraversal** to proceed from the point where *travBuffSize* was exceeded.

Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal**.

Description

When the amount of information obtained in a traversal exceeds the amount of memory available to the buffer (as specified with *travBuffSize*), your application can call **GciMoreTraversal** repeatedly to break the traversal into manageable amounts of information. The information returned by this function begins with the object report

following where the previous unfinished traversal left off. The level value is retained from the initial **GciTraverseObjs** call.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

Generally speaking, an application can continue to call **GciMoreTraversal** until it has obtained all requested information.

Naturally, GemStone will not continue an incomplete traversal if there is any chance that changes to the database in the intervening period might have invalidated the previous report or changed the connectivity of the objects in the path of the traversal. Specifically, GemStone will refuse to continue a traversal if, in the interval before attempting to continue, you:

- Modify the objects in the database directly by calling any of the **GciStore**... or **GciAdd**... functions;
- Call one of the Smalltalk message-sending functions **GciSendMsg**, **GciPerform**, **GciContinue**, or any of the **GciExecute**... functions.
- Abort your transaction, thus invalidating any subsequent information from that traversal.

Any attempt to call **GciMoreTraversal** after one of these events will generate an error.

Note that this holds true across multiple GemBuilder applications sharing the same GemStone session. Suppose, for example, that you were holding on to an incomplete traversal buffer and the user moved from the current application to another, did some work that required executing Smalltalk code, and then returned to the original application. You would be unable to continue the interrupted traversal.

If you attempt to call **GciMoreTraversal** when no traversal is underway, this function will generate the error GCI_ERR_TRAV_COMPLETED.

During the entire sequence of **GciTraverseObjs** and **GciMoreTraversal** calls that constitute a traversal, any single object report will be returned exactly once. Regardless of the connectivity of objects in the GemStone database, only one report will be generated for any non-special object.

The section "Organization of the Traversal Buffer" on page 5-308 describes the organization of traversal buffers in detail.

GciMoreTraversal provides automatic byte swizzling for binary floats.

Example

```
BoolType atEnd;

more = GciTraverseObjs(oopList, numOops, buff1, maxSz, level);
/* insert code here that uses GciFindObjRep to search
the buffer for desired information */
while (!atEnd) { /* and you want more information */
atEnd = GciMoreTraversal(buff2, maxSz);

/* Intervening code goes here, in place of this comment */
}
```

See Also

GCI_ALIGN, page 5-27
GciFindObjRep, page 5-117
GciNbMoreTraversal, page 5-173
GciNbTraverseObjs, page 5-183
GciObjRepSize, page 5-198
GciTraverseObjs, page 5-307

GciNbAbort

Abort the current transaction (nonblocking).

Syntax

void GciNbAbort( )

Description

The **GciNbAbort** function is equivalent in effect to **GciAbort**. However, **GciNbAbort** permits the application to proceed with non-GemStone tasks while the transaction is aborted, and **GciAbort** does not.

See Also

GciAbort, page 5-20
GciCheckAuth, page 5-34
GciCommit, page 5-48
GciNbCommit, page 5-160

GciNbBegin

Begin a new transaction (nonblocking).

Syntax

void GciNbBegin( )

Description

The **GciNbBegin** function is equivalent in effect to **GciBegin**. However, **GciNbBegin** permits the application to proceed with non-GemStone tasks while a new transaction is started, and **GciBegin** does not.

See Also

GciAbort, page 5-20
GciBegin, page 5-31
GciExecuteStr, page 5-72
GciNbAbort, page 5-155
GciNbExecuteStr, page 5-167

GciNbClampedTrav

Traverse an array of objects, subject to clamps (nonblocking).

Syntax

void GciNbClampedTrav(*theOops*, *numOops*, *travBuff*, *level*)

const OopType *	*theOops*;
ArraySizeType	*numOops*;
struct *	*travArgs*;

Input Arguments

theOops	An array of OOPs representing the objects to traverse.
numOops	The number of elements in *theOops.*
travArgs	Pointer to a **GciClampedTravArgsSType** structure. See **GciClampedTrav** for documentation on the fields in *travArgs.*

Result Arguments

travArgs	Pointer to a **GciClampedTravArgsSType** structure containing the result argument field *travBuff.*

Return Value

The **GciNbClampedTrav** function, unlike **GciClampedTrav**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciClampedTrav** by using the argument to **GciNbEnd**.

Description

The **GciNbClampedTrav** function is equivalent in effect to **GciClampedTrav**. However, **GciClampedTrav** permits the application to proceed with non-GemStone tasks while a traversal is carried out, and **GciClampedTrav** does not.

See Also

GciClampedTrav, page 5-37

GciNbClampedTraverseObjs

Traverse an array of objects, subject to clamps (nonblocking).

Syntax

```
void GciNbClampedTraverseObjs(clampSpec, theOops, numOops, travBuff, level)
    OopType            clampSpec;
    const OopType      theOops[ ];
    ArraySizeType      numOops;
    ByteType           travBuff[ ];
    long               level;
```

Input Arguments

clampSpec	The OOP of the Smalltalk ClampSpecification to be used. Refer to the *GemStone Kernel Reference* for a description of ClampSpecification.
theOops	An array of OOPs representing the objects to traverse.
numOops	The number of elements in *theOops.*
level	Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in *theOops*. When level is 2, an object report is also obtained for the instance variables of each level-1 object. When level is 0, the number of levels in the traversal is not restricted.

Result Arguments

travBuff	The buffer for the results of the traversal. The first element placed in the buffer is the *actualBufferSize*, a long integer that indicates how many bytes were actually stored in the buffer by this function. The remainder of the traversal buffer consists of a series of object reports, each of which is of type **GciObjRepSType**. You can use the macro GCI_IS_REPORT_CLAMPED to find out if a given object report represents a clamped object. If the report array would be empty, a single object report is created for the object *nil.*

Return Value

The **GciNbClampedTraverseObjs** function, unlike **GciClampedTraverseObjs**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciClampedTraverseObjs** by using the argument to **GciNbEnd**.

Description

The **GciNbClampedTraverseObjs** function is equivalent in effect to **GciClampedTraverseObjs**. However, **GciNbClampedTraverseObjs** permits the application to proceed with non-GemStone tasks while the traversal is completed, and **GciClampedTraverseObjs** does not.

GciNbClampedTraverseObjs provides automatic byte swizzling for binary floats.

See Also

GciClampedTraverseObjs, page 5-40
GCI_IS_REPORT_CLAMPED, page 5-137
GciTraverseObjs, page 5-307
GciNbTraverseObjs, page 5-183

GciNbCommit

Write the current transaction to the database (nonblocking).

Syntax

void GciNbCommit( )

Return Value

The **GciNbCommit** function, unlike **GciCommit**, does not have a return value. However, when the commit operation is complete, you can access a value identical in meaning to the return value of **GciCommit** by using the argument to **GciNbEnd**.

Description

The **GciNbCommit** function is equivalent in effect to **GciCommit**. However, **GciNbCommit** permits the application to proceed with non-GemStone tasks while the transaction is committed, and **GciCommit** does not.

See Also

GciAbort, page 5-20
GciCheckAuth, page 5-34
GciCommit, page 5-48
GciNbAbort, page 5-155

GciNbContinue

Continue code execution in GemStone after an error (nonblocking).

Syntax

```
void GciNbContinue(process)
    OopType                process;
```

Input Arguments

process	The OOP of a Process object (obtained as the value of the *context* field of an error report returned by **GciErr**).

Return Value

The **GciNbContinue** function, unlike **GciContinue**, does not have a return value. However, when the continued operation is complete, you can access a value identical in meaning to the return value of **GciContinue** by using the argument to **GciNbEnd**.

Description

The **GciNbContinue** function is equivalent in effect to **GciContinue**. However, **GciNbContinue** permits the application to proceed with non-GemStone tasks while the operation continues, and **GciContinue** does not.

See Also

GciClearStack, page 5-45
GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciNbExecute, page 5-165
GciSendMsg, page 5-255

GciNbContinueWith

Continue code execution in GemStone after an error (nonblocking).

Syntax

```
void GciNbContinueWith (process, replaceTopOfStack, flags, error)
    OopType          process;
    OopType          replaceTopOfStack;
    long             flags;
    GciErrSType *    error;
```

Input Arguments

process	The OOP of a Process object (obtained as the value of the *context* field of an error report returned by **GciErr**).
replaceTopOfStack	If not OOP_ILLEGAL, replace the top of the Smalltalk evaluation stack with this value before continuing. If OOP_ILLEGAL, the evaluation stack is not changed.
flags	Flags to disable or permit asynchronous events and debugging in Smalltalk, as defined for **GciPerformNoDebug**.
error	If not NULL, continue with an error. This argument takes precedence over *replaceTopOfStack*.

Description

The **GciNbContinueWith** function is equivalent in effect to **GciContinueWith**. However, **GciNbContinueWith** permits the application to proceed with non-GemStone tasks while the operation continues, and **GciContinueWith** does not.

See Also

GciContinue, page 5-49
GciContinueWith, page 5-51
GciErr, page 5-66
GciExecute, page 5-68
GciNbExecute, page 5-165
GciPerformNoDebug, page 5-218

GciNbEnd

Test the status of nonblocking call in progress for completion.

Syntax

GciNbProgressEType GciNbEnd(*result*)
    void **   *result*;

Input Arguments

result	The address at which **GciNbEnd** should place a pointer to the result of the nonblocking call when it is complete.

Return Value

The **GciNbEnd** function returns an enumerated type. Its value is GCI_RESULT_READY if the outstanding nonblocking call has completed execution and its result is ready, GCI_RESULT_NOT_READY if the call is not complete and there has been no change since the last inquiry, and GCI_RESULT_PROGRESSED if the call is not complete but progress has been made towards its completion.

Description

Once an application calls a nonblocking function, it must call **GciNbEnd** at least once, and must continue to do so until that nonblocking function has completed execution. The intent of the return values is to give the scheduler a hint about whether it is calling **GciNbEnd** too often or not often enough.

Once an operation is complete, you are permitted to call **GciNbEnd** repeatedly. It returns GCI_RESULT_READY and a pointer to the same result each time, until you call a nonblocking function again. It is an error to call **GciNbEnd** before you call any nonblocking functions at all. Use the **GciCallInProgress** function to determine whether or not there is a GemBuilder call currently in progress.

Example

```
void * myResult;

GciNbExecuteStr("aKeyValueDictionary keys sortAscending: ‘’”);
do {
  yield_to_other_tasks();
} while (GciNbEnd(&myResult) != GCI_RESULT_READY);

return *(OopType *)myResult;
```

See Also

GciCallInProgress, page 5-33

GciNbExecute

Execute a Smalltalk expression contained in a String object (nonblocking).

Syntax

void GciNbExecute(*source*, *symbolList*)
OopType *source*;
OopType *symbolList*;

Input Arguments

source	The OOP of a String containing a sequence of one or more statements to be executed.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

The **GciNbExecute** function, unlike **GciExecute**, does not have a return value. However, when the executed operation is complete, you can access a value identical in meaning to the return value of **GciExecute** by using the argument to **GciNbEnd**.

Description

The **GciNbExecute** function is equivalent in effect to **GciExecute**. However, **GciNbExecute** permits the application to proceed with non-GemStone tasks while the Smalltalk expression is executed, and **GciExecute** does not.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68

GciExecuteFromContext, page 5-70
GciExecuteStr, page 5-72
GciExecuteStrFromContext, page 5-74
GciNbContinue, page 5-161
GciNbExecuteStr, page 5-167
GciNbExecuteStrFromContext, page 5-169
GciSendMsg, page 5-255

GciNbExecuteStr

Execute a Smalltalk expression contained in a C string (nonblocking).

Syntax

void GciNbExecuteStr(*source*, *symbolList*)
  const char *source*[ ];
  OopType *symbolList*;

Input Arguments

source	A null-terminated string containing a sequence of one or more statements to be executed.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

The **GciNbExecuteStr** function, unlike **GciExecuteStr**, does not have a return value. However, when the executed operation is complete, you can access a value identical in meaning to the return value of **GciExecuteStr** by using the argument to **GciNbEnd**.

Description

The **GciNbExecuteStr** function is equivalent in effect to **GciExecuteStr**. However, **GciNbExecuteStr** permits the application to proceed with non-GemStone tasks while the Smalltalk expression is executed, and **GciExecuteStr** does not.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68

GciExecuteFromContext, page 5-70
GciExecuteStr, page 5-72
GciExecuteStrFromContext, page 5-74
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbExecuteStrFromContext, page 5-169
GciSendMsg, page 5-255

GciNbExecuteStrFromContext

Execute a Smalltalk expression contained in a C string as if it were a message sent to an object (nonblocking).

Syntax

void GciNbExecuteStrFromContext(*source*, *contextObject*, *symbolList*)
    const char *source*[ ];
    OopType *contextObject*;
    OopType *symbolList*;

Input Arguments

source	A null-terminated string containing a sequence of one or more statements to be executed.
contextObject	The OOP of any GemStone object.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolListDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

The **GciNbExecuteStrFromContext** function, unlike **GciExecuteStrFromContext**, does not have a return value. However, when the executed operation is complete, you can access a value identical in meaning to the return value of **GciExecuteStrFromContext** by using the argument to **GciNbEnd**.

Description

The **GciNbExecuteStrFromContext** function is equivalent in effect to **GciExecuteStrFromContext**. However, **GciNbExecuteStrFromContext** permits the application to proceed with non-GemStone tasks while the Smalltalk expression is executed, and **GciExecuteStrFromContext** does not.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciExecuteFromContext, page 5-70
GciExecuteStr, page 5-72
GciExecuteStrFromContext, page 5-74
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbExecuteStr, page 5-167
GciSendMsg, page 5-255

GciNbExecuteStrTrav

First execute a Smalltalk expression contained in a C string as if it were a message sent to an object, then traverse the result of the execution (nonblocking).

Syntax

BoolType GciExecuteStrTrav(*source*, *contextObject*, *symbolList*, *travArgs*)

const char	*source*[ ];
OopType	*contextObject*;
OopType	*symbolList*;
struct *	*travArgs*;

Input Arguments

source	A null-terminated string containing a sequence of one or more statements to be executed.
contextObject	The OOP of any GemStone object. A value of OOP_ILLEGAL means no context.
symbolList	The OOP of a GemStone symbol list (that is, an Array of instances of SymbolDictionary). The compiler uses the *symbolList* to resolve symbolic references in the code in *source*. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).
travArgs	Pointer to the **GciClampedTravArgsSType** structure. See the **GciExecuteStrTrav** function for field definitions.

Return Value

The **GciNbExecuteStrTrav** function, unlike **GciExecuteStrTrav**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciExecuteStrTrav** by using the argument to **GciNbEnd**.

Description

The **GciNbExecuteStrTrav** function is equivalent in effect to **GciExecuteStrTrav**. However, **GciNbExecuteStrTrav** permits the application to proceed with non-GemStone tasks while the traversal is completed, and **GciExecuteStrTrav** does not.

See Also

GciExecuteStrTrav, page 5-76
GciExecuteStr, page 5-72
GciMoreTraversal, page 5-152
GciPerformTraverse, page 5-224

GciNbMoreTraversal

Continue object traversal, reusing a given buffer (nonblocking).

Syntax

```
void GciNbMoreTraversal(travBuff, travBuffSize)
    ByteType            travBuff[ ];
    ArraySizeType       travBuffSize;
```

Input Arguments

travBuffSize	The number of bytes allocated to the traversal buffer.

Result Arguments

travBuff	A buffer in which the results of the traversal will be placed.

Return Value

The **GciNbMoreTraversal** function, unlike **GciMoreTraversal**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciMoreTraversal** by using the argument to **GciNbEnd**.

Description

The **GciNbMoreTraversal** function is equivalent in effect to **GciMoreTraversal**. However, **GciNbMoreTraversal** permits the application to proceed with non-GemStone tasks while the traversal is completed, and **GciMoreTraversal** does not.

GciNbMoreTraversal provides automatic byte swizzling for binary floats.

See Also

GCI_ALIGN, page 5-27
GciFindObjRep, page 5-117
GciMoreTraversal, page 5-152

GciNbTraverseObjs, page 5-183
GciObjRepSize, page 5-198
GciTraverseObjs, page 5-307

GciNbPerform

Send a message to a GemStone object (nonblocking).

Syntax

```
void GciNbPerform(receiver, selector, args, numArgs)
    OopType             receiver;
    const char          selector[ ];
    const OopType       args[ ];
    ArraySizeType       numArgs;
```

Input Arguments

receiver	The OOP of the receiver of the message.
selector	A string that defines the message selector. For keyword selectors, all keywords are concatenated in the string. (For example, at:put:).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.

Return Value

The **GciNbPerform** function, unlike **GciPerform**, does not have a return value. However, when the performed operation is complete, you can access a value identical in meaning to the return value of **GciPerform** by using the argument to **GciNbEnd**.

Description

The **GciNbPerform** function is equivalent in effect to **GciPerform**. However, **GciNbPerform** permits the application to proceed with non-GemStone tasks while the message is executed, and **GciPerform** does not.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbPerformNoDebug, page 5-177
GciPerform, page 5-216
GciPerformNoDebug, page 5-218
GciPerformSym, page 5-220
GciSendMsg, page 5-255

GciNbPerformNoDebug

Send a message to a GemStone object, and temporarily disable debugging (nonblocking).

Syntax

void GciNbPerformNoDebug(*receiver*, *selector*, *args*, *numArgs*)

OopType	*receiver*;
const char	*selector*[ ];
const OopType	*args*[ ];
ArraySizeType	*numArgs*;

Input Arguments

receiver	The OOP of the receiver of the message.
selector	A string that defines the message selector. For keyword selectors, all keywords are concatenated in the string. (For example, at:put:).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.

Return Value

The **GciNbPerformNoDebug** function, unlike **GciPerformNoDebug**, does not have a return value. However, when the performed operation is complete, you can access a value identical in meaning to the return value of **GciPerformNoDebug** by using the argument to **GciNbEnd**.

Description

The **GciNbPerformNoDebug** function is equivalent in effect to **GciPerformNoDebug**. However, **GciNbPerformNoDebug** permits the application to proceed with non-GemStone tasks while the message is executed, and **GciPerformNoDebug** does not.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbPerform, page 5-175
GciPerform, page 5-216
GciPerformNoDebug, page 5-218
GciPerformSym, page 5-220
GciSendMsg, page 5-255

GciNbPerformTrav

First send a message to a GemStone object, then traverse the result of the message (nonblocking).

Syntax

BoolType GciPerformTrav(*receiver*, *selector*, *args*, *numArgs*, *travBuff*, *level*)

OopType	*receiver*;
const char *	*selector*;
const OopType *	*args*;
unsigned int	*numArgs*;
struct *	*travArgs*;

Input Arguments

receiver	The OOP of the receiver of the message.
selector	The OOP of a String object that defines the message selector. For keyword selectors, all keywords are concatenated in the String. (For example, `at:put:`).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.
travArgs	Pointer to a **GciClampedTravArgsSType** structure. See the **GciClampedTrav** function for documentation of the fields in *travArgs*.

Result Arguments

The result of the **GciPerformTrav** is the first object in the resulting *travBuffs* field in *travArgs*.

Return Value

The **GciNbPerformTrav** function, unlike **GciPerformTrav**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciPerformTrav** by using the argument to **GciNbEnd**.

Description

The **GciNbPerformTrav** function is equivalent in effect to **GciPerformTrav**. However, **GciNbStoreTrav** permits the application to proceed with non-GemStone tasks while the traversal is done, and **GciPerformTrav** does not.

See Also

GciPerformTrav, page 5-222
GciPerform, page 5-216
GciClampedTrav, page 5-37

GciNbStoreTrav

Store multiple traversal buffer values in objects (nonblocking).

Syntax

```
void GciNbStoreTrav(travBuff, behaviorFlag)
    ByteType            travBuff [ ];
    long                behaviorFlag;
```

Input Arguments

travBuff	A buffer that contains the object reports to be stored. The first element in the buffer is a long integer that indicates how many bytes are stored in the buffer. The remainder of the traversal buffer consists of a series of object reports, each of which is of type **GciObjRepSType**.
behaviorFlag	A flag specifying whether the values returned by **GciStoreTrav** should be added to the values in the traversal buffer or should replace the values in the traversal buffer. Flag values, predefined in the `gci.ht` header file, are GCI_STORE_TRAV_NSC_ADD (add to the traversal buffer) and GCI_STORE_TRAV_NSC_REP (replace traversal buffer contents).

Description

The **GciNbStoreTrav** function is equivalent in effect to **GciStoreTrav**. However, **GciNbStoreTrav** permits the application to proceed with non-GemStone tasks while the traversals are stored, and **GciStoreTrav** does not.

GciNbStoreTrav provides automatic byte swizzling for binary floats.

See Also

GciMoreTraversal, page 5-152
GciNbMoreTraversal, page 5-173
GciNbTraverseObjs, page 5-183
GciNewOopUsingObjRep, page 5-191

GciStoreTrav, page 5-292
GciTraverseObjs, page 5-307

GciNbTraverseObjs

Traverse an array of GemStone objects (nonblocking).

Syntax

```
void GciNbTraverseObjs(theOops, numOops, travBuff, level)
    const OopType        theOops[ ];
    ArraySizeType        numOops;
    ByteType             travBuff[ ];
    ArraySizeType        travBuffSize;
    long                 level;
```

Input Arguments

theOops	An array of OOPs representing the objects to traverse.
numOops	The number of elements in theOops.
level	Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in theOops. When level is 2, an object report is also obtained for the instance variables of each level-1 object. When level is 0, the number of levels in the traversal is not restricted.

Result Arguments

travBuff	A buffer in which the results of the traversal will be placed.

Return Value

The **GciNbTraverseObjs** function, unlike **GciTraverseObjs**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciTraverseObjs** by using the argument to **GciNbEnd**.

Description

The **GciNbTraverseObjs** function is equivalent in effect to **GciTraverseObjs**. However, **GciNbTraverseObjs** permits the application to proceed with non-GemStone tasks while the traversal is completed, and **GciTraverseObjs** does not.

GciNbTraverseObjs provides automatic byte swizzling for binary floats.

See Also

GciFindObjRep, page 5-117
GciMoreTraversal, page 5-152
GciNbMoreTraversal, page 5-173
GciNbStoreTrav, page 5-181
GciNewOopUsingObjRep, page 5-191
GciObjRepSize, page 5-198
GciStoreTrav, page 5-292
GciTraverseObjs, page 5-307

GciNewByteObj

Create and initialize a new byte object.

Syntax

```
OopType GciNewByteObj(aClass, value, valueSize)
    OopType              aClass;
    const ByteType *     value;
    ArraySizeType        valueSize;
```

Input Arguments

aClass	The OOP of the class of which an instance is to be created.
value	Pointer to an array of byte values to be stored in the newly-created object.
valueSize	The number of byte values in value.

Return Value

The OOP of the newly created object.

Description

Returns a new instance of *aClass*, of size *valueSize*, and containing a copy of the bytes located at *value*. Equivalent to **GciNewOop** followed by **GciStoreBytes**. *aClass* must be a class whose format is Bytes.

GciNewCharObj

Create and initialize a new character object.

Syntax

OopType GciNewCharObj(*aClass*, *cString*)
  OopType *aClass*;
  const char * *cString*;

Input Arguments

aClass	The OOP of the class of which an instance is to be created. *aClass* must be a class whose format is BYTE.
cString	Pointer to an array of characters to be stored in the newly-created object. The terminating `'\0'` character is not stored.

Return Value

The OOP of the newly-created object.

Description

Returns a new instance of *aClass* which has been initialized to contain the bytes of *cString*, excluding the null terminator.

GciNewDateTime

Create and initialize a new date-time object.

Syntax

```
OopType GciNewDateTime(theClass, timeVal)
    OopType                    theClass;
    const GciDateTimeSType *timeVal;
```

Input Arguments

theClass	The class of the object to be created. *theClass* must be OOP_CLASS_DATE_TIME or a subclass thereof.
timeVal	The time value to be assigned to the newly-created object.

Return Value

Returns the OOP of the newly-created object. If an error occurs, returns OOP_ILLEGAL.

Description

Creates a new instance of *theClass* having the value that *timeVal* points to.

GciNewOop

Create a new GemStone object.

Syntax

OopType GciNewOop(*oclass*)
    OopType *oclass*;

Input Arguments

oclass The OOP of the class of which the new object is an instance. This may be the OOP of a class that you have created, or it may be one of the Smalltalk kernel classes, such as OOP_CLASS_STRING for an object of class String. Appendix A, “Reserved OOPs,” lists the C constants that are defined for each of the Smalltalk kernel classes.

Return Value

Returns the OOP of the new object. In case of error, this function returns OOP_NIL.

Description

This function creates a new object of the specified class and returns the object’s OOP.

Example

```
OopType newSym;

newSym = GciNewOop(OOP_CLASS_SYMBOL);
```

See Also

GciNewOops, page 5-189
GciNewOopUsingObjRep, page 5-191
GciReleaseAllOops, page 5-238
GciReleaseOops, page 5-239

GciNewOops

Create multiple new GemStone objects.

Syntax

void GciNewOops(*numOops*, *oclass*, *idxSize*, *result*)
    ArraySizeType *numOops*;
    const OopType *oclass*[ ];
    const unsigned long *idxSize*[ ];
    OopType *result*[ ];

Input Arguments

numOops	The number of new objects to be created.
oclass	For each new object, the OOP of its class.
idxSize	For each new object, the number of its indexed variables. If the specified oclass of an object is not indexable, its *idxSize* is ignored.

Result Arguments

result	An array of the OOPs of the new objects created with this function.

Return Value

If an error is encountered, this function will stop at the first error and the contents of the *result* array will be undefined.

Description

This function creates multiple objects of the specified classes and sizes, and returns the OOPs of the new objects.

Each OOP in *oclass* may be the OOP of a class that you have created, or it may be one of the Smalltalk kernel classes, such as OOP_CLASS_STRING for an object of class String. If *oclass* contains the OOP of a class with special implementation (such as Boolean), then the corresponding element in *result* is OOP_NIL. Appendix A, “Reserved OOPs,” lists the C constants that are defined for each of the Smalltalk kernel classes.

GciNewOops generates an error when either of the following conditions is TRUE for any object:

- *idxSize* `< 0`
- `(`*idxSize* `+ number_of_named_instance_variables) > maxSmallInt`

Example

```
OopType classes[3];
OopType sizes[3];
OopType newObjs[3];

classes[0] = OOP_CLASS_STRING;
classes[1] = OOP_CLASS_IDENTITY_SET;
classes[2] = OOP_CLASS_ARRAY;
sizes[0] = 50;
sizes[1] = 0;  /* ignored for NSCs anyway */
sizes[2] = 3;

GciNewOops(3, classes, sizes, newObjs);
```

See Also

GciNewOop, page 5-188
GciNewOopUsingObjRep, page 5-191
GciReleaseAllOops, page 5-238
GciReleaseOops, page 5-239
GciStoreTrav, page 5-292

GciNewOopUsingObjRep

Create a new GemStone object from an existing object report.

Syntax

void GciNewOopUsingObjRep(*anObjectReport*)
GciObjRepSType * *anObjectReport*;

Input Arguments

anObjectReport	A pointer to an object report.

Result Arguments

anObjectReport	A modified object report that contains the OOP of the new object (*hdr.objId*), the object's segment (*hdr.segment*), the number of named instance variables in the object (*hdr.namedSize*), the updated number of the object's indexed variables (*hdr.idxSize*), and the object's complete size (the sum of its named and unnamed variables, *hdr.objSize*).

Description

This function allows you to submit an object report that creates a GemStone object and specifies the values of its instance variables. You can use this function to define a String, pointer, or NSC object with known OOPs.

The object report consists of two parts: a header (a **GciObjRepHdrSType** structure) followed by a value buffer (an array of values of the object's instance variables). For more information on object reports, see "The Object Report Structure" on page 5-14.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

GciNewOopUsingObjRep provides automatic byte swizzling for binary floats.

Error Conditions

In addition to general GemBuilder error conditions, this function generates an error if any of the following conditions exist:

- If `(idxSize < 0)`
- If `(idxSize + namedSize) > maxSmallInt`
- If `firstOffset > (objSize + 1)`
- For pointer objects and NSCs, if *valueBuffSize* is not divisible by 4
- If the specified *oclass* is not the OOP of a Smalltalk class object
- If the specified *oclass* and implementation (*objImpl*) do not agree
- If *objId* is a binary float, then *startIndex* must be one and *valueBuffSize* must be the actual size for the class of *objId*.

Note that you cannot use this function to create new special objects (instances of SmallInteger, Character, Boolean, or UndefinedObject).

Word Alignment

You must use the macro GCI_ALIGN to ensure that the object report's value buffer begins at a word boundary. For example:

```
buffsize = (long) GCI_ALIGN(sizeof(GciObjRepSType)) +
           (4 * tsize) + 1000;
```

Example

```
unsigned int buffsize;
int     i;
char *    myBytes;

buffsize = (long) GCI_ALIGN(sizeof(GciObjRepSType)) +
        (4 * tsize) + 1000;
myObject = (GciObjRepSType *) malloc(buffsize);
myObject->firstOffset = 1;
myObject->oclass = OOP_CLASS_STRING;
myObject->idxSize = (long)tsize;
myObject->segment = OOP_NIL;
myObject->objImpl = GC_FORMAT_BYTE;
myObject->valueBuffSize = (long)tsize;
myBytes = (char *)GCI_VALUE_BUFF(myObject);   /* auto-aligns */
for (i = 0; i < tsize; i += 1) {
myBytes[i] = GCI_CHR_TO_OOP(i % 256);
}
GciNewOopUsingObjRep(myObject);
```

See Also

GciNewOop, page 5-188
GciReleaseAllOops, page 5-238
GciReleaseOops, page 5-239
GciTraverseObjs, page 5-307

GciNewString

Create a new String object from a C character string.

Syntax

```
OopType GciNewString(cString)
    const char *                  cString;
```

Input Arguments

cString	Pointer to a character string.

Return Value

The OOP of the newly created object.

Description

Returns a new instance of OOP_CLASS_STRING with the value that *cString* points to.

GciNewSymbol

Create a new Symbol object from a C character string.

Syntax

```
OopType GciNewSymbol(cString)
    const char *            cString;
```

Input Arguments

cString Pointer to a character string.

Return Value

The OOP of the newly-created object.

Description

Returns a new instance of OOP_CLASS_SYMBOL with the value that *cString* points to.

GciObjExists

Determine whether or not a GemStone object exists.

Syntax

BoolType GciObjExists(*theObject*)
OopType *theObject*;

Input Arguments

theObject The OOP of an object.

Return Value

Returns TRUE if *theObject* exists, FALSE otherwise.

Description

This function tests an OOP to see if the object to which it points is a valid object.

GciObjInCollection

Determine whether or not a GemStone object is in a Collection.

Syntax

BoolType GciObjInCollection(*anObj*, *aCollection*)
    OopType *anObj*;
    OopType *aCollection*;

Input Arguments

anObj	The OOP of an object for which to check.
aCollection	The OOP of a collection.

Return Value

Returns TRUE if *anObj* exists in *aCollection*, FALSE otherwise.

Description

Searches the specified collection for the specified object. If *aCollection* is an NSC (a kind of Bag or Set), this is a tree lookup. If *aCollection* is a kind of Array or String, this is a sequential scan. This function is equivalent to the Smalltalk method `Object|in:`.

GciObjRepSize

Find the number of bytes in an object report.

Syntax

long GciObjRepSize(*anObjectReport*)
    const GciObjRepHdrSType **anObjectReport*;

Input Arguments

anObjectReport	A pointer to an object report returned by **GciFindObjRep**.

Return Value

Returns the size of the specified object report.

Description

This function calculates the number of bytes in an object report. Before your application allocates memory for a copy of the object report, it can call this function to obtain the size of the report.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

Example

```
OopType theOop;
long  reportSize;
GciObjRepHdrSType * where;

where = GciFindObjRep(myBuff, theOop);
reportSize = GciObjRepSize(where);
```

See Also

GciFindObjRep, page 5-117
GciMoreTraversal, page 5-152
GciTraverseObjs, page 5-307

GCI_OOP_IS_BOOL

(MACRO) Determine whether or not a GemStone object represents a Boolean value.

Syntax

GCI_OOP_IS_BOOL(*theOop*)

Input Arguments

theOop The OOP of the object to test.

Result Value

A C Boolean value. Returns TRUE if the object represents a Boolean, FALSE otherwise.

Description

This macro tests to see if *theOop* represents a Boolean value.

See Also

GCI_LONG_IS_SMALL_INT, page 5-147
GCI_OOP_IS_SMALL_INT, page 5-201
GCI_OOP_IS_SPECIAL, page 5-202

GCI_OOP_IS_SMALL_INT

(MACRO) Determine whether or not a GemStone object represents a SmallInteger.

Syntax

GCI_OOP_IS_SMALL_INT(*theOop*)

Input Arguments

theOop	The OOP of the object to test.

Result Value

A C Boolean value. Returns TRUE if the object represents a SmallInteger, FALSE otherwise.

Description

This macro tests to see if *theOop* represents a SmallInteger.

See Also

GCI_LONG_IS_SMALL_INT, page 5-147
GCI_OOP_IS_BOOL, page 5-200
GCI_OOP_IS_SPECIAL, page 5-202

GCI_OOP_IS_SPECIAL

(MACRO) Determine whether or not a GemStone object has a special representation.

Syntax

GCI_OOP_IS_SPECIAL(*theOop*)

Input Arguments

theOop	The OOP of the object to test.

Result Value

A C Boolean value. Returns TRUE if the object has a special representation, FALSE otherwise.

Description

This macro tests to see if *theOop* has a special representation.

See Also

GCI_LONG_IS_SMALL_INT, page 5-147
GCI_OOP_IS_BOOL, page 5-200
GCI_OOP_IS_SMALL_INT, page 5-201

GciOopToBool

Convert a Boolean object to a C Boolean value.

Syntax

BoolType GciOopToBool(*theObject*)
    OopType *theObject*;

Input Arguments

theObject	The OOP of the Boolean object to be translated into a C Boolean value.

Return Value

Returns the C Boolean value that corresponds to the GemStone object. In case of error, this function returns FALSE.

Description

This function translates a GemStone Boolean object into the equivalent C Boolean value.

Example

```
BoolType aBool;
OopType theObj;

aBool = GciOopToBool(theObj);
```

See Also

GCI_BOOL_TO_OOP, page 5-32

GCI_OOP_TO_BOOL

(MACRO) Convert a Boolean object to a C Boolean value.

Syntax

GciOopToBool(*theObject*)

Input Arguments

theObject	The OOP of the Boolean object to be translated into a C Boolean value.

Result Value

A C Boolean value. Returns the C Boolean value that corresponds to the GemStone object. In case of error, this macro returns FALSE.

Description

This macro translates a GemStone Boolean object into the equivalent C Boolean value. GCI_OOP_TO_BOOL runs faster than the equivalent function, GciOopToBool. If the argument is out of range for the result type, GciOopToBool is called to generate an error.

Example

```
BoolType aBool;
OopType theObj;

aBool = GCI_OOP_TO_BOOL(theObj);
```

See Also

GCI_BOOL_TO_OOP, page 5-32

GciOopToChr

Convert a Character object to a C character value.

Syntax

char GciOopToChr(*theObject*)
    OopType *theObject*;

Input Arguments

theObject	The OOP of the Character object to be translated into a C character value.

Return Value

Returns the C character value that corresponds to the GemStone object. In case of error, this function returns zero.

Description

This function translates a GemStone Character object into the equivalent C character value.

Example

```
char  aChar;
OopType theObj;

aChar = GciOopToChr(theObj);
```

See Also

GCI_CHR_TO_OOP, page 5-36

GCI_OOP_TO_CHR

(MACRO) Convert a Character object to a C character value.

Syntax

GCI_OOP_TO_CHR(*theObject*)

Input Arguments

theObject	The OOP of the Character object to be translated into a C character value.

Result Value

The **GCI_OOP_TO_CHR** macro returns the C character value that corresponds to the GemStone object. In case of error, it returns zero.

Description

The **GCI_OOP_TO_CHR** macro translates a GemStone Character object into the equivalent C character value.

Example

```
char  aChar;
OopType theObj;

aChar = GCI_OOP_TO_CHR(theObj);
```

See Also

GCI_CHR_TO_OOP, page 5-36
GciOopToChr, page 5-205

GciOopToFlt

Convert a Float object to a C double value.

Syntax

double GciOopToFlt(*theObject*)
OopType *theObject*;

Input Arguments

theObject	The OOP of the Float object to be translated into a C floating point value.

Return Value

Returns the C double precision value that corresponds to the GemStone object. In case of any error other than HOST_ERR_INEXACT_PRECISION, this function returns a PlusQuietNaN.

Description

This function translates a GemStone Float object into the equivalent C double precision value.

If your C compiler's floating point package doesn't have a representation that corresponds to one of the values listed below, **GciOopToFlt** may generate the following errors when converting GemStone Float objects into C values:

HOST_ERR_INEXACT_PRECISION
when called to convert a number whose precision exceeds that of the C double type. This error is not fielded by GemBuilder's setjmp/longjmp mechanism. If you want to check for this error, you must explicitly call **GciErr** after **GciOopToFlt**.

HOST_ERR_MAGNITUDE_OUT_OF_RANGE
when called to convert a number whose exponent is too large (or small) to be held in a C double precision value

HOST_ERR_NO_PLUS_INFINITY
when called to convert a value of positive infinity

HOST_ERR_NO_MINUS_INFINITY
when called to convert a value of negative infinity

HOST_ERR_NO_PLUS_QUIET_NAN
when called to convert a positive quiet NaN

HOST_ERR_NO_MINUS_QUIET_NAN
when called to convert a negative quiet NaN

HOST_ERR_NO_PLUS_SIGNALING_NAN
when called to convert a positive signaling NaN

HOST_ERR_NO_MINUS_SIGNALING_NAN
when called to convert a negative signaling NaN

Example

```
OopType thePriceOop; /* The OOP of a product's price */
double unitPrice;  /* The price of the part */

unitPrice = GciOopToFlt(thePriceOop);
```

See Also

GciFltToOop, page 5-119

GciOopToLong

Convert a Gemstone object to a C long integer value.

Syntax

long GciOopToLong(*theObject*)
    OopType *theObject*;

Input Arguments

theObject The OOP of the object to be translated into a C integer value.

Return Value

The **GciOopToLong** function returns the C long integer value that is equivalent to the value of *theObject*.

Description

The **GciOopToLong** function translates a GemStone object into a C long integer value that has the same value.

The object identified by *theObject* must be a SmallInteger, a LargePositiveInteger with a value less than 2147483648, or a LargeNegativeInteger with a value greater than -2147483649. If the object is not one of these kinds or it does not meet the value restrictions, **GciOopToLong** generates the error OBJ_ERR_NOT_LONG.

Example

```
OopType anIntObject;
long  val;

/* Intervening code goes here, in place of this comment */

val = GciOopToLong (anIntObject);
/* val now contains a long integer which has the same value
as the GemStone object indicated by anIntObject */
```

See Also

GciLongToOop, page 5-148
GCI_LONG_TO_OOP, page 5-150
GCI_OOP_TO_LONG, page 5-211
GciUnsignedLongToOop, page 5-313

GCI_OOP_TO_LONG

(MACRO) Convert a GemStone object to a C long integer value.

Syntax

GCI_OOP_TO_LONG(*theObject*)

Input Arguments

theObject	The OOP of the object to be translated into a C integer value.

Result Value

The **GCI_OOP_TO_LONG** macro returns the C long integer value that is equivalent to the value of *theObject*.

Description

The **GCI_OOP_TO_LONG** macro translates a GemStone object into a C long integer value that has the same value.

The object identified by *theObject* must be a SmallInteger, a LargePositiveInteger with a value less than 2147483648, or a LargeNegativeInteger with a value greater than -2147483649. If the object is not one of these kinds or it does not meet the value restrictions, **GCI_OOP_TO_LONG** generates the error OBJ_ERR_NOT_LONG.

GCI_OOP_TO_LONG runs faster than the equivalent function, **GciOopToLong**, for SmallIntegers.

Example

```
OopType anIntObject;
long  val;

/* Intervening code goes here, in place of this comment */

val = GCI_OOP_TO_LONG(anIntObject);
/* val now contains a long integer which has the same value
as the GemStone object indicated by anIntObject */
```

See Also

GciLongToOop, page 5-148
GCI_LONG_TO_OOP, page 5-150
GciOopToLong, page 5-209
GciUnsignedLongToOop, page 5-313

GciPathToStr

Convert a path representation from numeric to string.

Syntax

```
BoolType GciPathToStr(aClass, path, pathSize, maxResultSize, result)
    OopType          aClass;
    const long       path[ ];
    ArraySizeType    pathSize;
    ArraySizeType    maxResultSize;
    char             result[ ];
```

Input Arguments

aClass	The class of the object for which this path will apply. That is, for each instance of this class, store or fetch objects along the designated path.
path	The path array to be converted to string format.
pathSize	The number of integers in the path array.
maxResultSize	The maximum allowable length of the resulting path string, excluding the null terminator.

Result Arguments

result	The resulting path string, terminated with a null character. The resulting string is of the form `foo.bar.name`. Each element of the path string is the name of an instance variable (that is, bar is an instance variable of foo, and name is an instance variable of bar).

Return Value

Returns TRUE if the path array was successfully converted to a string. Returns FALSE otherwise.

Description

The **GciPathToStr** function converts the numeric representation of a path to its equivalent string representation.

The functions **GciFetchPaths** and **GciStorePaths** allow you to specify paths along which to fetch from, or store into, objects within an object tree.

A path may be represented as an array of integers, in which each step along the path is represented by an integral offset from the beginning of an object. For example, an array containing the integers 5 and 2 would represent the offsets of the fifth and second instance variables, respectively. Alternatively, a path may be represented as a string in which each element is the name of the corresponding instance variable. For example, *address.zip*, in which *zip* is an instance variable of *address*.

For more information about paths, see the discussion of the **GciFetchPaths** function on page 5-103.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

Restrictions

Note that **GciPathToStr** can convert a numeric path only if:

- The instance variables of the specified Smalltalk class (*aClass*) are constrained in such a way that the path is guaranteed to be valid for all instances.
- The path touches only named instance variables. If a path leads through the indexed variables of some object, then no symbolic representation can be used.

If your application does not impose GemStone constraints on classes of all objects from which you to fetch, or if you want to fetch from indexable objects, then you need to maintain your paths as arrays of integers.

Error Conditions

The following errors may be generated by this function:

GCI_ERR_RESULT_PATH_TOO_LARGE
The *result* was larger than the specified *maxResultSize*.

RT_ERR_PATH_TO_STR_IVNAME
One of the instance variable offsets in the path array was invalid.

RT_ERR_STR_TO_PATH_CONSTRAINT
One of the instance variables in the path string was not sufficiently constrained.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
long  path[10];
ArraySizeType pathSize;
char  result[100];

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

path[0] = 3; /* 3 happens to be the offset of the cost instVar */
pathSize = 1;
GciPathToStr(GciFetchClass(aComponent), path, pathSize, 100,
          result);
```

See Also

GciFetchPaths, page 5-103
GciStorePaths, page 5-287
GciStrToPath, page 5-300

GciPerform

Send a message to a GemStone object.

Syntax

```
OopType GciPerform(receiver, selector, args, numArgs)
    OopType          receiver;
    const char       selector[ ];
    const OopType    args[ ];
    ArraySizeType    numArgs;
```

Input Arguments

receiver	The OOP of the receiver of the message.
selector	A string that defines the message selector. For keyword selectors, all keywords are concatenated in the string. (For example, at:put:).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.

Return Value

Returns the OOP of the result of Smalltalk execution. In case of error, this function returns OOP_NIL.

Description

This function sends a message (that is, the selector along with any keyword arguments and their corresponding values) to the specified receiver (an object in the GemStone database). Because **GciPerform** calls the virtual machine, you can issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 1-14.

The **GciSendMsg** function provides an alternate method of sending messages to GemStone objects. For a comparison of those functions, see “Sending Messages to GemStone Objects” on page 1-12.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbPerform, page 5-175
GciNbPerformNoDebug, page 5-177
GciPerformNoDebug, page 5-218
GciPerformSym, page 5-220
GciSendMsg, page 5-255

GciPerformNoDebug

Send a message to a GemStone object, and temporarily disable debugging.

Syntax

```
OopType GciPerformNoDebug(receiver, selector, args, numArgs, flags)
    OopType              receiver;
    const char           selector[ ];
    const OopType        args[ ];
    ArraySizeType        numArgs;
    long                 flags;
```

Input Arguments

receiver	The OOP of the receiver of the message.
selector	A string that defines the message selector. For keyword selectors, all keywords are concatenated in the string. (For example, at:put:).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.
flags	Flags to disable or permit asynchronous events and debugging in Smalltalk.

Return Value

Returns the OOP of the result of Smalltalk execution. In case of error, this function returns OOP_NIL.

Description

This function is a variant of **GciPerform** that is identical to it except for just one difference. **GciPerformNoDebug** disables any breakpoints and single step points that currently exist in GemStone while the message is executing. This feature is typically used while implementing a Smalltalk debugger.

The value of *flags* should be given by using one or both of the following GemBuilder mnemonics:

- GCI_PERFORM_FLAG_ENABLE_DEBUG makes **GciPerformNoDebug** behave like **GciPerform** with respect to debugging.
- GCI_PERFORM_FLAG_DISABLE_ASYNC_EVENTS disables asynchronous events.

These two can either be used alone or logically "or"ed together.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciNbContinue, page 5-161
GciNbExecute, page 5-165
GciNbPerform, page 5-175
GciNbPerformNoDebug, page 5-177
GciPerform, page 5-216
GciPerformSym, page 5-220
GciSendMsg, page 5-255

GciPerformSym

Send a message to a GemStone object, using a String object as a selector.

Syntax

OopType GciPerform(*receiver*, *selector*, *args*, *numArgs*, *isNoDebug*)

OopType	*receiver*;
OopType	*selector*;
const OopType	*args*[ ];
ArraySizeType	*numArgs*;
BoolType	*isNoDebug*;

Input Arguments

receiver	The OOP of the receiver of the message.
selector	The OOP of a String object that defines the message selector. For keyword selectors, all keywords are concatenated in the String. (For example, `at:put:`).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.
isNoDebug	A flag to disable or permit debugging in Smalltalk.

Return Value

Returns the OOP of the result of Smalltalk execution. In case of error, this function returns OOP_NIL.

Description

If the isNoDebug flag is FALSE, this function is a variant of **GciPerform**; if the flag is TRUE, this function is a variant of **GciPerformNoDebug**. In either case, its operation is identical to the other function. The difference is that **GciPerformSym** takes an OOP as its selector instead of a C string.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciPerform, page 5-216
GciPerformNoDebug, page 5-218
GciSendMsg, page 5-255

GciPerformTrav

First send a message to a GemStone object, then traverse the result of the message.

Syntax

BoolType GciPerformTrav(*receiver*, *selector*, *args*, *numArgs*, *travBuff*, *level*)

OopType	*receiver*;
const char *	*selector*;
const OopType *	*args*;
unsigned int	*numArgs*;
struct *	*travArgs*;

Input Arguments

receiver	The OOP of the receiver of the message.
selector	The OOP of a String object that defines the message selector. For keyword selectors, all keywords are concatenated in the String. (For example, `at:put:`).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.
travArgs	Pointer to a **GciClampedTravArgsSType** structure. See the **GciClampedTrav** function for documentation of the fields in *travArgs*.

Result Arguments

The result of the **GciPerform** is the first object in the resulting *travBuffs* field in *travArgs*.

Return Value

Returns TRUE if the result is complete and no errors occurred. Returns FALSE if the traversal is not yet completed. You can then call **GciMoreTraversal** to proceed, if there is no GciError.

Description

This function is does the equivalent of a **GciPerform** using the first four arguments, and then performs a **GciClampedTrav**, starting from the result of the perform, and doing a traversal as specified by *travArgs*. In all GemBuilder traversals, objects are traversed post depth first.

See Also

GciPerform, page 5-216
GciClampedTrav, page 5-37

GciPerformTraverse

First send a message to a GemStone object, then traverse the result of the message.

Syntax

BoolType GciPerformTraverse(*receiver*, *selector*, *args*, *numArgs*, *travBuff*, *level*)

OopType	*receiver*;
const char	*selector*[ ];
const OopType	*args*[ ];
unsigned int	*numArgs*;
ByteType	*travBuff*[ ];
long	*level*;

Input Arguments

receiver	The OOP of the receiver of the message.
selector	The OOP of a String object that defines the message selector. For keyword selectors, all keywords are concatenated in the String. (For example, `at:put:`).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.
level	Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in theOops. When level is 2, an object report is also obtained for the instance variables of each level-1 object. When level is 0, the number of levels in the traversal is not restricted.

Result Arguments

travBuff	A buffer in which the results of the traversal are placed

Return Value

Returns FALSE if the traversal is not yet completed, but further traversal would cause the *travBuffSize* to be exceeded. If the *travBuffSize* is reached before the traversal is complete, you can then call **GciMoreTraversal** to proceed from the point where *travBuffSize* was exceeded.

Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal**.

Description

Consider the following function call:

```
BoolType atEnd;

atEnd = GciPerformTraverse(receiver, selector, args, numArgs,
          travBuff, level);
```

It is equivalent to the following code:

```
BoolType result;
OopType tmp[1];

*tmp = GciPerform(receiver, selector, args, numArgs);
atEnd = GciTraverseObjs(tmp, 1, travBuff, level);
```

GciPerformTraverse provides automatic byte swizzling for binary floats.

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciFindObjRep, page 5-117
GciMoreTraversal, page 5-152
GciNewOopUsingObjRep, page 5-191
GciObjRepSize, page 5-198
GciPerform, page 5-216
GciPerformNoDebug, page 5-218

GciPerformSym, page 5-220
GciSendMsg, page 5-255
GciStoreTrav, page 5-292
GciTraverseObjs, page 5-307

GciPollForSignal

Poll GemStone for signal errors without executing any Smalltalk methods.

Syntax

BoolType GciPollForSignal( )

Return Value

This function returns TRUE if a signal error or an asynchronous error exists, and FALSE otherwise.

Description

GemStone permits selective response to signal errors: RT_ERR_SIGNAL_ABORT, RT_ERR_SIGNAL_COMMIT, and RT_ERR_SIGNAL_GEMSTONE_SESSION. The default condition is to leave them all invisible. GemStone responds to each single kind of signal error only after an associated method of class System has been executed: `enableSignaledAbortError`, `enableSignaledObjectsError`, and `enableSignaledGemStoneSessionError` respectively.

After **GciInit** executes successfully, the GemBuilder default condition also leaves all signal errors invisible. The **GciPollForSignal** function permits GemBuilder to check signal errors manually. However, GemStone must respond to each kind of error in order for GemBuilder to respond to it. Thus, if an application calls **GciPollForSignal**, then GemBuilder can check exactly the same kinds of signal errors as GemStone responds to. If GemStone has not executed any of the appropriate System methods, then this call has no effect until it does.

GemBuilder treats any signal errors that it finds just like any other errors, through **GciErr** or the **longjmp** mechanism, as appropriate. Instead of checking manually, these errors can be checked automatically by calling the **GciEnableSignaledErrors** function.

GciPollForSignal also detects any asynchronous errors whenever they occur, including but not limited to the following errors: ABORT_ERR_LOST_OT_ROOT, GS_ERR_SHRPC_CONNECTION_FAILURE, GS_ERR_STN_NET_LOST, GS_ERR_STN_SHUTDOWN, and GS_ERR_SESSION_SHUTDOWN.

See Also

GciEnableSignaledErrors, page 5-64
GciErr, page 5-66

GciPopErrJump

Discard a previously saved error jump buffer.

Syntax

```
void GciPopErrJump(jumpBuffer)
    jmp_buf                 jumpBuffer;
```

Input Arguments

jumpBuffer	A pointer to a jump buffer specified in an earlier call to **GciPushErrJump** or **GciPushErrHandler**.

Description

This function discards one or more jump buffers that were saved with earlier calls to **GciPushErrJump** or **GciPushErrHandler**. Your program must call this function when a saved execution environment is no longer useful for error handling.

GemBuilder maintains a stack of error jump buffers. After your program calls **GciPopErrJump**, the jump buffer at the top of the stack will be used for subsequent GemBuilder error handling. If no jump buffers remain, your program will need to call **GciErr** and test for errors locally.

To pop multiple jump buffers in a single call to **GciPopErrJump**, specify the *jumpBuffer* argument from an earlier call to **GciPushErrJump** or **GciPushErrHandler**. See the following example.

Example

```
/* Assume that jump buffers 1-4 are all local to the same
function. */

GciPushErrJump (jumpBuff1);

/* Intervening code goes here, in place of this comment */

GciPushErrJump (jumpBuff2);

/* Intervening code goes here, in place of this comment */

GciPushErrJump (jumpBuff3);

/* Intervening code goes here, in place of this comment */

GciPushErrJump (jumpBuff4);

/* Intervening code goes here, in place of this comment */

GciPopErrJump (jumpBuff1); /* pops buffers 1-4 */
```

See Also

GciErr, page 5-66
GciPushErrJump, page 5-233
GciPushErrHandler, page 5-232
GciSetErrJump, page 5-258

GciProcessDeferredUpdates

Process deferred updates to objects that do not allow direct structural update.

Syntax

long GciProcessDeferredUpdates(*void*)

Return Value

Returns the number of objects that had deferred updates.

Description

This function processes updates to instances of classes that have the noStructuralUpdate bit set, including AbstractDictionary, Bag, Set, and their subclasses. After operations that modify an instance of once of these classes, either **GciProcessDeferredUpdates** must be called, or the final **GciStoreTrav** must have GCI_STORE_TRAV_FINISH_UPDATES set.

The following GemBuilder calls operate on instances whose classes have noStructuralUpdate set: **GciCreateOopObj**, **GciStoreTrav**, **GciStore...Oops, GciAdd...Oops, GciReplace...Oops**. Behavior of other GemBuilder update calls on such instances is undefined.

An attempt to commit automatically executes a deferred update.

Executing a deferred update before all forward references are resolved can produce errors that require the application to recover by doing a **GciAbort** or **GciLogout**.

An OOP buffer used to update the varying portion of an object with noStructuralUpdate must contain the OOPs to be added to the varying portion of the object, with two exceptions:

- If the object is a kind of KeyValueDictionary that does not store Associations, the buffer must contain (key, value) pairs.
- If the object is a kind of AbstractDictionary that stores Associations or (key, Association) pairs, the value buffer must contain Associations.

See Also

GciStoreTrav, page 5-292

GciPushErrHandler

Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.

Syntax

void GciPushErrHandler(*jumpBuffer*)
GCI_SIG_JMP_BUF_TYPE*jumpBuffer*;

Input Arguments

jumpBuffer	A pointer to a jump buffer.

Description

The **GciPushErrHandler** function pushes *jumpBuffer* onto the stack of GemBuilder jump buffers. You must prepare *jumpBuffer* for pushing by calling `GCI_SETJMP(` *jumpBuffer*`)` before calling **GciPushErrHandler**. You must also declare *jumpBuffer* as type `GCI_SIG_JMP_BUF_TYPE` in your application.

As long as *jumpBuffer* is on the stack, **GciPushErrHandler** offers a performance gain over **GciPushErrJump**, not only in error processing, but in every call to a GemBuilder function.

To use *jumpBuffer* for a long jump in your application (once it is pushed), call the macro `GCI_LONGJMP`. The block of code that fields the long jump must not call `GCI_LONGJMP` itself. If that code also processes an error, then it must call the **GciHandleError** function as its first action. Failure to meet this requirement will result in unpredictable program behavior.

Example

For an illustration of the use of **GciPushErrHandler**, refer to the example for the **GciHandleError** function on page 5-125.

See Also

GciHandleError, page 5-125

GciPushErrJump

Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.

Syntax

void GciPushErrJump(*jumpBuffer*)
jmp_buf *jumpBuffer*;

Result Arguments

jumpBuffer A pointer to a jump buffer, as described below.

Description

This function allows your application program to take advantage of the setjmp/longjmp error-handling mechanism from within any GemBuilder function call. However, you cannot use this mechanism to handle errors within **GciPushErrJump** itself, or within the related functions **GciPopErrJump** and **GciSetErrJump**.

When your program calls **setjmp**, the context of the C environment is saved in a jump buffer that you designate. To associate subsequent GemBuilder error handling with that jump buffer, you would then call **GciPushErrJump**.

GemBuilder maintains a stack of up to 20 error jump buffers. A buffer is pushed onto the stack when **GciPushErrJump** is called, and popped when **GciPopErrJump** is called. When an error occurs during a GemBuilder call, the GemBuilder function causes a **longjmp** to the buffer currently at the top of GemBuilder's error jump stack, and pops that buffer from the stack. At that time, the previous environment is restored.

For functions with local error recovery, your program can call **GciSetErrJump** to temporarily disable the setjmp/longjmp error handling mechanism (and to re-enable error handling afterwards).

Whenever the jump stack is empty, you must use **GciErr** to field any GemBuilder errors.

The **setjmp** and **longjmp** functions are described in your C compiler documentation.

Example

The pseudo-code in this example illustrates two distinct error jumps: one to handle any application-dependent errors, and a second to handle any GemBuilder errors. After setting up the error jumps, the program instructs GemBuilder to use *jumpBuf2* on any error, then enters a while loop getting and doing work.

```
myFunction () {
OopType oString, myErrors;
jmp_buf jumpBuf1, jumpBuf2;
GciErrSType error;

/* Assume that you have created the GemStone Dictionary
   myErrors, which contains your user-defined English-
   language error messages. Obtain the OOP of myErrors
   for subsequent use in error handling. */

myErrors = GciExecuteStr("myErrors", OOP_NIL)

/* Intervening code goes here, in place of this comment */

/* Note that setjmp returns 0 when first called. Thus, the
   following error-handling code is skipped the first time
   through. */

/* trap application-dependent errors here, using longjmp */
if (setjmp (jumpBuf1)) {
   /* do error handling */
}

/* trap  GemBuilder errors here */
if (setjmp (jumpBuf2)) {
   GciErr(&error);
   switch(error.category) {
      case OOP_COMPILER_ERROR_CAT:
         /* do something */
         break;
      case OOP_RUNTIME_ERROR_CAT:
         switch (error.number) {
            case ErrMnemonic1:
               /* do something */
               break;
```

```
                case ErrMnemonic2:
                    /* do something */
                    break;
                case default:
                    /* do something */
                    break;
            }
            break;
        case OOP_ABORTING_ERROR_CAT:
            /* do something */
            break;
        case OOP_FATAL_ERROR_CAT:
            /* do something */
            break;
        case myErrors:
            switch (error.number) {
                case ErrMnemonic3:
                    /* do something */
                    break;
                case default:
                    /* do something */
            }
            break;
        default:
            /* do something */
    }
}
/* arm (or rearm) GemBuilder to use jumpBuf2 for error handling         */
GciPushErrJump (jumpBuf2);

while (true) {
    /* Now do some work.   When an error occurs during a
     GemBuilder call, control is passed to the jumpBuf2
     error handler. */

}
}
```

See Also

GciErr, page 5-66
GciPopErrJump, page 5-229
GciSetErrJump, page 5-258

GciRaiseException

Signal an error, synchronously, within a user action.

Syntax

void GciRaiseException(*err*)
    GciErrSType * *err*;

Input Arguments

err A pointer to the error type to raise.

Description

When executed from within a user action, this function raises an exception and passes the given error to the error signalling mechanism, causing control to return to Smalltalk.

This function has no effect when executed outside of a user action.

GciReleaseAllOops

Mark all imported GemStone OOPs as eligible for garbage collection.

Syntax

void GciReleaseAllOops( )

Description

The **GciReleaseAllOops** function removes all OOPs from the user session's export set, thus permitting GemStone to consider removing them as a result of garbage collection. Objects that are connected to persistent objects are not removed during garbage collection, even if they are not in the export set. It is typical usage to call **GciReleaseAllOops** after successfully committing a transaction.

The **GciSaveObjs** function may be used to make objects ineligible for garbage collection. Note that results of the **GciNew...**, **GciCreate...**, **GciSendMsg**, **GciPerform...**, and **GciExecute...** functions are automatically ineligible. You must release those objects explicitly if they are to be eligible.

WARNING!

Before releasing all objects, be sure that you do not need to retain any of them for any reason.

See Also

"Garbage Collection" on page 1-31
GciReleaseOops, page 5-239
GciSaveObjs, page 5-254

GciReleaseOops

Mark an array of GemStone OOPs as eligible for garbage collection.

Syntax

void GciReleaseOops(*theOops*, *numOops*)
const OopType *theOops*[ ];
ArraySizeType *numOops*;

Input Arguments

theOops	An array of OOPs. Each element of the array corresponds to an object to be released.
numOops	The number of elements in *theOops*.

Description

The **GciReleaseOops** function removes the specified OOPs from the user session's export set, thus permitting GemStone to remove them as a result of garbage collection.

The **GciSaveObjs** function may be used to make objects ineligible for garbage collection. Note that results of the **GciNew...**, **GciCreate...**, **GciSendMsg**, **GciPerform...**, and **GciExecute...** functions are automatically ineligible. You must release those objects explicitly if they are to be eligible.

WARNING!

Before releasing any of your objects, be sure that you do not need to retain them for any reason.

Example

```
OopType obj[2];

obj[0] = GciFetchOop(OOP_CLASS_ARRAY);
printf("%ld\n", obj[0]);
obj[1] = GciFetchOop(obj[0],1);
printf("%ld\n", obj[1]);
GciReleaseOops(obj, 2);
```

See Also

“Garbage Collection” on page 1-31
GciReleaseAllOops, page 5-238
GciSaveObjs, page 5-254

GciRemoveOopFromNsc

Remove an OOP from an NSC.

Syntax

```
BoolType GciRemoveOopFromNsc(theNsc, theOop)
    OopType             theNsc;
    OopType             theOop;
```

Input Arguments

theNsc	The OOP of the NSC from which to remove an OOP.
theOop	The OOP of the object to be removed.

Result Arguments

theNsc	The OOP of the modified NSC.

Return Value

Returns FALSE if *theOop* was not present in the NSC. Returns TRUE if *theOop* was present in the NSC.

Description

This function removes an OOP from the unordered variables of an NSC, using structural access.

Example

```
int   i;
OopType oNscObject;
OopType oNum;
BoolType rmvd;

for (i = 0; i < 6; i++) {
oNum = GciLongToOop((long) i);
rmvd = GciRemoveOopFromNsc(oNscObject, oNum);
}
```

See Also

GciAddOopToNsc, page 5-22
GciAddOopsToNsc, page 5-23
GciRemoveOopsFromNsc, page 5-243

GciRemoveOopsFromNsc

Remove one or more OOPs from an NSC.

Syntax

BoolType GciRemoveOopsFromNsc(*theNsc*, *theOops*, *numOops*)

OopType	*theNsc*;
const OopType	*theOops*[ ];
ArraySizeType	*numOops*;

Input Arguments

theNsc	The OOP of the NSC from which to remove the OOPs.
theOops	The array of OOPs to be removed from the NSC.
numOops	The number of OOPs to remove.

Result Arguments

theNsc	The OOP of the modified NSC.

Return Value

Returns FALSE if any element of *theOops* was not present in the NSC. Returns TRUE if all elements of *theOops* were present in the NSC.

Description

This function removes multiple OOPs from the unordered variables of an NSC, using structural access. If any individual OOP is not present in the NSC, this function returns FALSE, but it still removes all OOPs that it finds in the NSC.

Example

```
ArraySizeType remove_cnt = 6;
OopType oNscObject;
OopType bigptrs[L_SUB_SIZE];
BoolType all_rmvd;

all_rmvd = GciRemoveOopsFromNsc(oNscObject, bigptrs, remove_cnt);
```

See Also

GciAddOopToNsc, page 5-22
GciAddOopsToNsc, page 5-23
GciRemoveOopFromNsc, page 5-241

GciReplaceOops

Replace all instance variables in a GemStone object.

Syntax

void GciReplaceOops(*theObj*, *theOops*, *numOops*)
    OopType *theObj*;
    const OopType *theOops*[ ];
    ArraySizeType *numOops*;

Input Arguments

theOops	The array of OOPs used as the replacements.
numOops	The number of OOPs in *theOops.*

Result Arguments

theObj	The object whose instance variables are replaced.

Description

GciReplaceOops uses structural access to replace *all* the instance variables in the object. However, it does so in a context that is external to the object. Hence, it completely ignores private named instance variables in its operation.

If *theObj* is of fixed size, then it is an error for *numOops* to be of a different size. If *theObj* is of a variable size, then it is an error for *numOops* to be of a size smaller than the number of named instance variables (*namedSize*) of the object. For variable-sized objects, **GciReplaceOops** resets the number of unnamed variables to *numOops* - *namedSize.*

GciReplaceOops is not recommended for use with variable-sized objects unless they are indexable or are NSCs. Other variable-sized objects, such as KeyValue dictionaries, do not store values at fixed offsets.

See Also

GciReplaceVaryingOops, page 5-247
GciStoreIdxOops, page 5-276
GciStoreNamedOops, page 5-280
GciStoreOops, page 5-284

GciReplaceVaryingOops

Replace all unnamed instance variables in an NSC object.

Syntax

```
void GciReplaceVaryingOops(theNsc, theOops, numOops)
    OopType             theNsc;
    const OopType       theOops[ ];
    ArraySizeType       numOops;
```

Input Arguments

theOops	The array of objects used as the replacements.
numOops	The number of objects in *theOops.*

Result Arguments

theNsc	The NSC object whose unnamed instance variables are replaced.

Description

GciReplaceVaryingOops uses structural access to replace all unnamed instance variables in the NSC object.

See Also

GciReplaceOops, page 5-245
GciStoreIdxOops, page 5-276
GciStoreNamedOops, page 5-280
GciStoreOops, page 5-284

GciResolveSymbol

Find the OOP of the object to which a symbol name refers, in the context of the current session's user profile.

Syntax

```
OopType  GciResolveSymbol(cString, symbolList)
     const char *           cString;
     OopType                symbolList;
```

Input Arguments

cString	The name of a symbol as a character string.
symbolList	The OOP of an instance of OOP_CLASS_SYMBOL_LIST or OOP_NIL.

Return Value

The OOP of the object that corresponds to the specified symbol.

Description

Attempts to resolve the symbol name *cString* using symbol list *symbolList*. If *symbolList* is OOP_NIL, this function searches the symbol list in the user's UserProfile. If the symbol is not found or an error is generated, the result is OOP_ILLEGAL. If result is OOP_ILLEGAL and **GciErr** reports no error, then the symbol could not be resolved using the given *symbolList*. If an error such as an authorization error occurs, the result is OOP_ILLEGAL and the error is accessible by **GciErr**.

This function is similar to **GciResolveSymbolObj**, except that the symbol argument is a C string instead of an object identifier.

See Also

GciResolveSymbolObj, page 5-249

GciResolveSymbolObj

Find the OOP of the object to which a symbol object refers, in the context of the current session's user profile.

Syntax

OopType GciResolveSymbolObj(*aSymbolObj*, *symbolList*)
OopType *aSymbolObj*;
OopType *symbolList*;

Input Arguments

aSymbolObj	The OOP of a kind of String. That is, this object's class must be OOP_CLASS_STRING or a subclass thereof.
symbolList	The OOP of an instance of OOP_CLASS_SYMBOL_LIST or OOP_NIL.

Return Value

The OOP of the object that corresponds to the specified symbol.

Description

Attempts to resolve *aSymbolObj* using symbol list *symbolList*. If *symbolList* is OOP_NIL, this function searches the symbol list in the user's UserProfile. If the symbol is not found or an error is generated, the result is OOP_ILLEGAL. If the result is OOP_ILLEGAL and **GciErr** reports no error, then the symbol could not be resolved using the given *symbolList*. If an error such as an authorization error occurs, the result is OOP_ILLEGAL and the error is accessible by **GciErr**.

This function is similar to **GciResolveSymbol**, except that the symbol argument is an object identifier instead of a C string.

See Also

GciResolveSymbol, page 5-248

GciRtlIsLoaded

Report whether a GemBuilder library is loaded.

Syntax

BoolType GciRtlIsLoaded( )

Return Value

Returns TRUE if a GemBuilder library is loaded and FALSE if not.

Description

The **GciRtlIsLoaded** function reports whether an executable has loaded one of the versions of GemBuilder. The GemBuilder library files are dynamically loaded at run time. See “The GemBuilder Shared Libraries” on page 2-2 for more information.

See Also

GciRtlLoad, page 5-251
GciRtlUnload, page 5-253

GciRtlLoad

Load a GemBuilder library.

Syntax

BoolType GciRtlLoad(*useRpc*, *path*, *errBuf[]*, *errBufSize)*

BoolType	*useRpc*
const char *	*path*
char	*errBuf[]*
ArraySizeType	*errBufSize*

Input Arguments

useRpc	A flag to specify the RPC or linked version of GemBuilder.
path	A list of directories (separated by ;) to search for the GemBuilder library.
errBuf	A buffer to store any error message.
errBufSize	The size of *errBuf*.

Return Value

Returns TRUE if a GemBuilder library loads successfully. If the load fails, the "The GemBuilder Shared Libraries" on page 3-2return value is FALSE, and a null-terminated error message is stored in *errBuf*, unless *errBuf* is NULL.

Description

The **GciRtlLoad** function attempts to load one of the GemBuilder libraries. If *useRpc* is TRUE, the RPC version of GemBuilder is loaded. If *useRpc* is FALSE, the linked version of GemBuilder is loaded. See "The GemBuilder Shared Libraries" on page 2-2 for more information.

If *path* is not NULL, it must point to a list of directories to search for the library to load. If *path* is NULL, then a default path is searched.

If a GemBuilder library is already loaded, the call fails.

See Also

GciRtlIsLoaded, page 5-250
GciRtlUnload, page 5-253

GciRtlUnload

Unload a GemBuilder library.

Syntax

void GciRtlUnload( )

Description

The **GciRtlUnload** function causes the library loaded by **GciRtlLoad** to be unloaded. Once the current library is unloaded, **GciRtlLoad** can be called again to load a different GemBuilder library. See "The GemBuilder Shared Libraries" on page 2-2 for more information.

See Also

GciRtlLoad, page 5-251
GciRtlIsLoaded, page 5-250

GciSaveObjs

Mark an array of OOPs as ineligible for garbage collection.

Syntax

void GciSaveObjs(*theOops*, *numOops*)
  const OopType   *theOops*[ ];
  ArraySizeType   *numOops*;

Input Arguments

theOops	An array of OOPs.
numOops	The number of elements in *theOops*.

Description

The **GciSaveObjs** function places the specified OOPs in the user session's export set, thus preventing GemStone from removing them as a result of garbage collection. **GciSaveObjs** can add any OOP to the export set.

The **GciSaveObjs** function does *not* itself make objects persistent, and it does *not* create a reference to them from a persistent object so that the next commit operation will try to do so either. It only protects them from garbage collection.

Note that results of the **GciNew...**, **GciCreate...**, **GciSendMsg**, **GciPerform...**, and **GciExecute...** functions are automatically added to the export set. The **GciRelease...** functions may be used to make objects eligible for garbage collection.

See Also

"Garbage Collection" on page 1-31
GciAddSaveObjsToReadSet, page 5-24
GciReleaseAllOops, page 5-238
GciReleaseOops, page 5-239

GciSendMsg

Send a message to a GemStone object.

Syntax

OopType GciSendMsg(*receiver*, *numMsgParts*, *msgParts*, ...)
OopType *receiver*;
unsigned int *numMsgParts*;
const char *msgParts*[ ], ...;

Input Arguments

receiver	The OOP of the receiver of the message.
numMsgParts	The number of parts to the message (the selector, plus any keywords and their arguments). For unary selectors (messages with no arguments), *numMsgParts* is 1. For keyword selectors, *numMsgParts* is always even (one value corresponding to each keyword).
msgParts	A variable number of arguments. For unary selectors, this is a single string. For keyword selectors, this is a series of strings (the keywords themselves) and OOPs (the values corresponding to each keyword). For example: `"at:"`, `oKey`, `"put:"`, `oValue` .

Return Value

Returns the OOP of the message result. In case of error, this function returns OOP_NIL.

Description

This function sends a message (that is, the selector along with any keyword arguments and their corresponding values) to the specified receiver (an object in the GemStone database). Because **GciSendMsg** calls the virtual machine, you can issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 1-14.

The **GciPerform** function provides an alternate method of sending messages to GemStone objects. For a comparison of those functions, see “Sending Messages to GemStone Objects” on page 1-12.

Example

```
OopType theIdOop; /* The OOP of an identifier string */

theIdOop = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(theIdOop, 1L, id, (long)strlen(id));
ourPart = GciSendMsg(partOOP, 2, "newWithId:", theIdOop);
```

The following example uses **GciSendMsg** to create a new subclass of Object called TestClass:

```
theNewClass = GciNewOop(OOP_CLASS_STRING);
GciStoreBytes(theNewClass, 1L, "TestClass", 9);

theDict = GciExecuteStr(
         "^ (System myUserProfile symbolList at: 1)", OOP_NIL);

oArray = GciNewOop(OOP_CLASS_ARRAY);
oTestClass = GciSendMsg(OOP_CLASS_OBJECT, 14L, "subclass:",
theNewClass, "instVarNames:", oArray, "classVarNames:",
oArray, "poolDictionaries:", oArray, "inDictionary:", theDict,
"constraints:", oArray, "isInvariant:", OOP_FALSE);
```

Here are some other illustrations of **GciSendMsg** calls:

```
oString = GciSendMsg(oValue, 1, "asString");
oValue = GciSendMsg(oDict, 4, "at:", oKey, "ifAbsent:", oBlock);
```

See Also

GciContinue, page 5-49
GciErr, page 5-66
GciExecute, page 5-68
GciPerform, page 5-216

GciSessionIsRemote

Determine whether or not the current session is using a Gem on another machine.

Syntax

BoolType GciSessionIsRemote( )

Return Value

The **GciSessionIsRemote** function returns TRUE if the current GemBuilder session is connected to a remote Gem. It returns FALSE if the current GemBuilder session is connected to a linked Gem.

GciSessionIsRemote raises an error if the current session is invalid.

GciSetErrJump

Enable or disable the current error handler.

Syntax

```
BoolType GciSetErrJump(aBoolean)
    BoolType            aBoolean;
```

Input Arguments

aBoolean	TRUE enables error jumps to the execution environment saved by the most recent **GciPushErrJump** or **GciPushErrHandler**; FALSE disables error jumps.

Return Value

Returns TRUE if error handling was previously enabled for the jump buffer at the top of the error jump stack. Returns FALSE if error handling was previously disabled. If your program has no buffers saved in its error jump stack, this function returns FALSE. (This function cannot generate an error.)

For most GemBuilder functions, calling **GciErr** after a successful function call will return zero (that is, false). In such cases, the **GciErrSType** error report structure will contain some default values. (See the **GciErr** function on page 5-66 for details.) However, a successful call to **GciSetErrJump** does not alter any previously existing error report information. That is, calling **GciErr** after a successful call to **GciSetErrJump** will return the same error information that was present before this function was called.

Description

This function enables or disables the error handler at the top of GemBuilder's error jump stack.

Example

```
myFunction() {
BoolType  prevValue;
GciErrSType error;

prevValue = GciSetErrJump(FALSE);
/* disable error jumps and save the previous setting */

/* Intervening code goes here, in place of this comment */

if (GciErr(&error)) /* handle any errors locally */
   localHandler;

/* Intervening code goes here, in place of this comment */

GciSetErrJump(prevValue); /* reset error jump flag */

/* Intervening code goes here, in place of this comment */

GciErr(&error); /* returns PREVIOUS error */
}
```

See Also

GciErr, page 5-66
GciPopErrJump, page 5-229
GciPushErrJump, page 5-233
GciPushErrHandler, page 5-232

GciSetNet

Set network parameters for connecting the user to the Gem and Stone processes.

Syntax

void GciSetNet(*StoneName*, *HostUserId*, *HostPassword*, *GemService*)
const char *StoneName*[ ];
const char *HostUserId*[ ];
const char *HostPassword*[ ];
const char *GemService*[ ];

Input Arguments

StoneName	Network resource string for the database monitor process.
HostUserId	Login name.
HostPassword	Password of the user.
GemService	Network resource string for the GemStone service.

Description

Your application, your GemStone session (Gem), and the database monitor (Stone) can all run in separate processes, on separate machines in your network. The **GciSetNet** function specifies the network parameters that are used to connect the current user to GemStone on the host, whenever **GciLogin** is called. Network resource strings specify the information needed to establish communications between these processes (see Appendix B, “Network Resource String Syntax”). See the *GemStone System Administration Guide* for complete information on the network environment.

StoneName identifies the name and network location of the database monitor process (Stone), which is the final arbiter of all sessions that access a specific database. Every session must communicate with a Stone, in both linked and remote applications. Hence, *StoneName* is a required argument.

A Stone process called “gemserver41” on node “lichen” could be described in a network resource string as:

```
!tcp@lichen!gemserver41
```

A Stone of the same name that is running on the same machine as the application could be described in shortened form simply as:

```
gemserver41
```

GemService identifies the name and network location of the GemStone service that creates a session process (Gem), which then arbitrates data access between the database and the application. Every GemStone session requires a Gem. In linked applications, one Gem is present within the same process as the application; in remote applications no such Gem is present. Therefore, each time an application user logs in to GemStone (after the first time in linked applications), the GemStone service must create a new Gem. Hence, *GemService* is a required argument, except in the special case of a linked application that limits itself to one GemStone login per application process. In this special case, specify *GemService* as an empty string.

For most installations, the GemStone service name is *gemnetobject*. Specify, for example:

```
!tcp@lichen!gemnetobject
```

If you use the C shell (`/bin/csh`) under Unix, the GemStone service name is *gemnetobjcsh*.

HostUserId and *HostPassword* are your login name and password, respectively, on the machines that host the Gem and Stone processes. Do not confuse these values with your GemStone username and password. These arguments provide authentication for such tasks as creating a Gem and establishing communications with a Stone. When such authentication is required, an application user cannot login to GemStone until the host login is verified for the machine running the Stone or Gem, in addition to the GemStone login itself.

Authentication is always required if the netldi process that is related to the Stone is running in secure mode. In this case, it makes no difference whether the application is linked or remote. Authentication is also required to create a remote Gem, unless the netldi process is running in guest mode. Remote applications must always create a Gem, but linked applications may also do so.

With TCP/IP, GemBuilder can also try to find a username and password for authentication on a host machine in your network initialization file. Because this file contains your password, you should ensure that other users do not have authorization to read it. Under UNIX, the file is named `.netrc` and it should contain lines of the form:

`machine` *machine_name* `login` *user_name* `password` *passwd*

For example:

```
machine alf login joebob password mypassword
```

To prevent GemBuilder from looking for authentication information in the network initialization file, supply a valid non-empty C string for the *HostUserId* argument. Also supply a non-empty string for the *HostPassword* argument to provide a password. An empty string and a NULL pointer both mean that no password will be used for authentication.

Alternatively, to direct GemBuilder to look in the network initialization file at need, supply an empty C string or a NULL pointer for the *HostUserId* argument. In this case, supply a NULL pointer for the *HostPassword* argument as well. Any valid string that you supply for a password is ignored in favor of the information that is present in the network initialization file.

Example

```
char * StoneName;
char * HostUserId;
char * HostPassword;
char * GemService;
char * gsUserName;
char * gsPassword;

StoneName  = "!tcp@alf!gemserver41";
HostUserId  = "newtoni";
HostPassword = "gravity";
GemService  = "!tcp@lichen!gemnetobject";
gsUserName  = "isaac newton";
gsPassword  = "pomme";

if (!GciInit()) exit; /* required before first      GemBuilder login */

GciSetNet(StoneName, HostUserId, HostPassword, GemService);
GciLogin(gsUserName, gsPassword);
```

See Also

GciLoadUserActionLibrary, page 5-142
“Network Resource String Syntax” on page B-1

GciSetSessionId

Set an active session to be the current one.

Syntax

void GciSetSessionId(*sessionId*)
GciSessionIdType *sessionId*;

Input Arguments

sessionId The session ID of an active (logged-in) GemStone session.

Description

This function can be used to switch between multiple GemStone sessions in an application program with multiple logins.

Example

```
GciLogin (USERID1, PASSWORD1);
/* Intervening code goes here, in place of this comment */
theFirstSessionId = GciGetSessionId( );
/* Intervening code goes here, in place of this comment */
GciLogin (USERID2, PASSWORD2); /* use previous network parameters */
/* Intervening code goes here, in place of this comment */
GciSetSessionId (theFirstSessionId);
```

See Also

GciGetSessionId, page 5-124
GciLoadUserActionLibrary, page 5-142

GciShutdown

Logout from all sessions and deactivate GemBuilder.

Syntax

void GciShutdown( )

Description

This function is intended to be called by image exit routines, such as the **on_exit** system call. In the linkable GemBuilder, **GciShutdown** calls **GciLogout**. In the RPC version, it logs out all sessions connected to the Gem process and shuts down the networking layer, thus releasing all memory allocated by GemBuilder.

It is especially important to call this function explicitly on any computer whose operating system does not automatically deallocate resources when a process quits. This effect is found on most small, single-user systems, such as the Macintosh or any machine that uses MS-DOS.

GciSoftBreak

Interrupt the execution of Smalltalk code, but permit it to be restarted.

Syntax

void GciSoftBreak( )

Description

This function sends a soft break to the current user session (set by the last **GciLogin** or **GciSetSessionId**).

GemBuilder allows users of your application to terminate Smalltalk execution. For example, if your application sends a message to an object (via **GciSendMessage** or **GciPerform**), and for some reason the invoked Smalltalk method enters an infinite loop, the user can interrupt the application.

GciSoftBreak interrupts only the Smalltalk virtual machine (if it is running), and does so in such a way that the it can be restarted. The only GemBuilder functions that can recognize a soft break include **GciSendMessage**, **GciPerform**, and **GciContinue**, and the **GciExecute**... functions.

In order for GemBuilder functions in your program to recognize interrupts, your program will need an interrupt routine that can call the functions **GciSoftBreak** and **GciHardBreak**. Since GemBuilder does not relinquish control to an application until it has finished its processing, soft and hard breaks must be initiated from an interrupt service routine.

If GemStone is executing when it receives the break, it replies with the error message RT_ERR_SOFT_BREAK. Otherwise, it ignores the break.

For an example of how **GciSoftBreak** is used, see the **GciClearStack** function on page 5-45.

See Also

GciClearStack, page 5-45
GciContinue, page 5-49
GciExecute, page 5-68
GciHardBreak, page 5-127
GciPerform, page 5-216
GciSendMsg, page 5-255

GciStoreByte

Store one byte in a byte object.

Syntax

```
void GciStoreByte(theObject, atIndex, theByte)
    OopType        theObject;
    long           atIndex;
    ByteType       theByte;
```

Input Arguments

theObject	The OOP of the GemStone byte object.
atIndex	The index into theObject at which to store the byte.
theByte	The 8-bit value to be stored.

Result Arguments

theObject	The resulting GemStone byte object.

Description

The **GciStoreByte** function stores a single element in a byte object at a specified index, using structural access.

GciStoreByte raises an error if *theObject* is a binary float. You must store all the bytes of a binary float if you store any.

Example

```
int   numI;
OopType oIndexedObject;

oIndexedObject = GciNewOop(OOP_CLASS_STRING);

for (numI = 0; numI < tsize; numI += 1) {
GciStoreByte(oIndexedObject, (long)(numI + 1),
             (ByteType)(numI % 256));
}
```

See Also

GciFetchByte, page 5-79
GciFetchBytes, page 5-81
GciStoreBytes, page 5-268

GciStoreBytes

Store multiple bytes in a byte object.

Syntax

void GciStoreBytes(*theObject*, *startIndex*, *theBytes*, *numBytes*)

OopType	*theObject*;
long	*startIndex*;
const ByteType	*theBytes*[ ];
ArraySizeType	*numBytes*;

Input Arguments

theObject	The OOP of the GemStone byte object.
startIndex	The index into theObject at which to begin storing bytes.
theBytes	The array of bytes to be stored.
numBytes	The number of elements to store.

Result Arguments

theObject	The resulting GemStone byte object.

Description

The **GciStoreBytes** function uses structural access to store multiple elements from a C array in a byte object, beginning at a specified index. A common application of **GciStoreBytes** would be to store a text string.

Error Conditions

GciStoreBytes raises an error if *theObject* is a binary float. Use **GciStoreBytesInstanceOf** instead for binary floats.

Example

```
ByteType id[MAXLEN + 1]; /* Holds the ID of the supplier */
OopType theIdOop;    /* Holds the OOP of the ID */

getString(id, MAXLEN);
theIdOop = GciNewOop(OOP_CLASS_SYMBOL);
GciStoreBytes(theIdOop, 1L, id, (ArraySizeType)strlen(id));
```

See Also

GciFetchByte, page 5-79
GciFetchBytes, page 5-81
GciStoreByte, page 5-266
GciStoreBytesInstanceOf, page 5-270
GciStoreChars, page 5-272

GciStoreBytesInstanceOf

Store multiple bytes in a byte object.

Syntax

void GciStoreBytesInstanceOf(*theClass*, *theObject*, *startIndex*, *theBytes*, *numBytes*)

OopType	*theClass*;
OopType	*theObject*;
long	*startIndex*;
const ByteType	*theBytes*[ ];
ArraySizeType	*numBytes*;

Input Arguments

theClass	The OOP of the class of the GemStone byte object.
theObject	The OOP of the GemStone byte object.
startIndex	The index into theObject at which to begin storing bytes.
theBytes	The array of bytes to be stored.
numBytes	The number of elements to store.

Result Arguments

theObject	The resulting GemStone byte object.

Description

The **GciStoreBytesInstanceOf** function uses structural access to store multiple elements from a C array in a byte object, beginning at a specified index. A common application of **GciStoreBytesInstanceOf** would be to store a binary float.

GciStoreBytesInstanceOf provides automatic byte swizzling for binary floats. The presence of the argument *theClass* enables the swizzling to be implemented more efficiently. If *theObject* is a binary float, then *theClass* must match the actual class of *theObject*, *startIndex* must be one, and *numBytes* must be the actual size for *theClass*. If any of these conditions are not met, then **GciStoreBytesInstanceOf** raises an error as a safety check.

If *theObject* is not a binary float, then *theClass* is ignored. Hence, you must supply the correct class for *theClass* if *theObject* is a binary float, but you can use OOP_NIL otherwise.

Example

```
double  pi;    /* the value of the new Float obj */
OopType theFloat; /* the OOP of the new Float obj */

pi    = 3.1415926;
theFloat = GciNewOop(OOP_CLASS_FLOAT);
GciStoreBytesInstanceOf(OOP_CLASS_FLOAT, theFloat, 1L,
     (ByteType *) &pi, (ArraySizeType) sizeof(pi));
```

See Also

GciFetchByte, page 5-79
GciFetchBytes, page 5-81
GciStoreByte, page 5-266
GciStoreBytes, page 5-268
GciStoreChars, page 5-272

GciStoreChars

Store multiple ASCII characters in a byte object.

Syntax

void GciStoreChars(*theObject*, *startIndex*, *aString*)

OopType	*theObject*;
long	*startIndex*;
const char *	*aString*;

Input Arguments

theObject	The OOP of the GemStone byte object.
startIndex	The index into theObject at which to begin storing the string.
aString	The string to be stored.

Result Arguments

theObject	The resulting GemStone byte object.

Description

The **GciStoreChars** function uses structural access to store a C string in a byte object, beginning at a specified index.

GciStoreChars raises an error if *theObject* is a binary float. ASCII characters have no meaning as bytes in a binary float.

Example

```
char * id;      /* String holding the ID of the supplier */
OopType theIdOop; /* Holds the OOP of the ID */

printf("ID    = ");
fflush(stdout);
getString(id, MAXLEN);
theIdOop = GciNewOop(OOP_CLASS_SYMBOL);
GciStoreChars(theIdOop, 1L, id);
```

See Also

GciFetchByte, page 5-79
GciFetchBytes, page 5-81
GciStoreByte, page 5-266
GciStoreBytes, page 5-268

GciStoreIdxOop

Store one OOP in a pointer object's unnamed instance variable.

Syntax

void GciStoreIdxOop(*theObject*, *atIndex*, *theOop*)
OopType *theObject*;
long *atIndex*;
OopType *theOop*;

Input Arguments

theObject	The pointer object.
atIndex	The index into *theObject* at which to store the object.
theOop	The OOP to be stored.

Result Arguments

theObject	The resulting pointer object.

Description

This function stores a single OOP into an indexed variable of a pointer object at the specified index, using structural access. Note that this function cannot be used for NSCs. (To add an OOP to an NSC, use the **GciAddOopToNsc** function on page 5-22.)

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent, newValue;

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr("AllComponents select:
                                [i|i.partnumber = 1234]");

/* store new value into 3rd element of aComponent's parts list */
newValue = the_new_value ;
GciStoreIdxOop(aComponent, 3L, newValue );
```

See Also

GciAddOopToNsc, page 5-22
GciFetchVaryingOop, page 5-110
GciFetchVaryingOops, page 5-113
GciStoreIdxOops, page 5-276

GciStoreIdxOops

Store one or more OOPs in a pointer object's unnamed instance variables.

Syntax

```
void GciStoreIdxOops(theObject, startIndex, theOops, numOops)
    OopType            theObject;
    long               startIndex;
    const OopType      theOops[ ];
    ArraySizeType      numOops;
```

Input Arguments

theObject	The pointer object.
startIndex	The index into *theObject* at which to begin storing OOPs.
theOops	The array of OOPs to be stored.
numOops	The number of OOPs to store.

Result Arguments

theObject	The resulting pointer object.

Description

This function uses structural access to store multiple OOPs from a C array into the indexed variables of a pointer object, beginning at the specified index. Note that this call cannot be used with NSCs. (To add multiple OOPs to an NSC, use the **GciAddOopsToNsc** function on page 5-23.)

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType oArray[10];
/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

/* store new values for first 5 elements of aComponent's parts
  list   */
oArray[0] = aValue;

/* Intervening code goes here, in place of this comment */

oArray[4] = aValue;
GciStoreIdxOops(aComponent, 1L, oArray, 5);
```

See Also

GciAddOopsToNsc, page 5-23
GciFetchVaryingOop, page 5-110
GciFetchVaryingOops, page 5-113
GciReplaceOops, page 5-245
GciReplaceVaryingOops, page 5-247
GciStoreIdxOop, page 5-274
GciStoreIdxOops, page 5-276
GciStoreNamedOops, page 5-280
GciStoreOops, page 5-284

GciStoreNamedOop

Store one OOP into an object's named instance variable.

Syntax

```
void GciStoreNamedOop(theObject, atIndex, theOop)
    OopType         theObject;
    long            atIndex;
    OopType         theOop;
```

Input Arguments

theObject	The object in which to store the OOP.
atIndex	The index into *theObject*'s named instance variables at which to store the OOP.
theOop	The OOP to be stored.

Result Arguments

theObject	The resulting object with the new OOP.

Description

This function stores a single OOP into an object's named instance variable at the specified index, using structural access.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType theName;
OopType newValue;

/* C constants to match Smalltalk class definition */
#define COMPONENT_OFF_PARTNUMBER 1L
#define COMPONENT_OFF_NAME    2L
#define COMPONENT_OFF_COST    3L

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr("AllComponents select:
                                          [i|i.partnumber = 1234]");

/* assign a new value to the name instance variable of aComponent
*/
newValue = a_new_value;
GciStoreNamedOop(aComponent, COMPONENT_OFF_NAME, newValue);

/* alternate approach: assign a new value to a named instance
variable without knowing its offset at compile time */
GciStoreNamedOop(aComponent,
GciIvNameToIdx(GciFetchClass(aComponent), "name"), newValue);
```

See Also

GciFetchNamedOop, page 5-88
GciFetchNamedOops, page 5-90
GciStoreIdxOop, page 5-274
GciStoreNamedOops, page 5-280

GciStoreNamedOops

Store one or more OOPs into an object's named instance variables.

Syntax

void GciStoreNamedOops(*theObject*, *startIndex*, *theOops*, *numOops*)

OopType	*theObject*;
long	*startIndex*;
const OopType	*theOops*[ ];
ArraySizeType	*numOops*;

Input Arguments

theObject	The object in which to store the OOPs.
startIndex	The index into *theObject*'s named instance variables at which to begin storing OOPs.
theOops	The array of OOPs to be stored.
numOops	The number of OOPs to store. If `(numOops+startIndex)` exceeds the number of named instance variables in *theObject*, an error is generated.

Result Arguments

theObject	The resulting object with the new OOPs.

Description

This function uses structural access to store multiple OOPs from a C array into an object's named instance variables, beginning at the specified index.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
ArraySizeType namedSize
long  i;
OopType *oBuffer;

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

/* store named instance variables without knowing how many at
compile time */
namedSize = GciFetchNamedSize(aComponent);
oBuffer = (OopType *) malloc( namedSize * sizeof(OopType));
for (i=0; i < namedSize; i++) {
oBuffer[i] = a_new_value; }  /* assign new values */
GciStoreNamedOops(aComponent, 1L, oBuffer, namedSize);
```

See Also

GciFetchNamedOop, page 5-88
GciFetchNamedOops, page 5-90
GciReplaceOops, page 5-245
GciReplaceVaryingOops, page 5-247
GciStoreIdxOop, page 5-274
GciStoreIdxOops, page 5-276
GciStoreNamedOop, page 5-278
GciStoreNamedOops, page 5-280
GciStoreOops, page 5-284

GciStoreOop

Store one OOP into an object's instance variable.

Syntax

void GciStoreOop(*theObject*, *atIndex*, *theOop*)
    OopType   *theObject*;
    long   *atIndex*;
    OopType   *theOop*;

Input Arguments

theObject	The object in which to store the OOP.
atIndex	The index into *theObject* at which to store the OOP. This function does not distinguish between named and unnamed instance variables. Indices are based at the beginning of an object's array of instance variables. In that array, the object's named instance variables are followed by its unnamed instance variables.
theOop	The OOP to be stored.

Result Arguments

theObject	The resulting object.

Description

This function stores a single OOP into an object at the specified index, using structural access. Note that this function cannot be used for NSCs. To add an object to an NSC, use the **GciAddOopToNsc** function on page 5-22.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType newValue;

/* C constants to match Smalltalk class definition */
#define COMPONENT_OFF_NAME 2L

/* retrieve a random instance of class Component */

aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

/* Two ways to assign new value to the name instance variable of
  aComponent */

newValue = an_oop;
GciStoreOop(aComponent, COMPONENT_OFF_NAME, newValue);
GciStoreNamedOop(aComponent, COMPONENT_OFF_NAME, newValue);

/* Two ways to assign a new value to the 3rd element of
  aComponent's   parts list without knowing exactly how many named
  instance variables exist */

GciStoreOop(aComponent, GciFetchNamedSize(aComponent) + 3L,
              newValue);
GciStoreIdxOop(aComponent, 3L, newValue);
```

See Also

GciAddOopToNsc, page 5-22
GciFetchVaryingOop, page 5-110
GciFetchVaryingOops, page 5-113
GciFetchOops, page 5-101
GciStoreOops, page 5-284

GciStoreOops

Store one or more OOPs into an object's instance variables.

Syntax

void GciStoreOops(*theObject*, *startIndex*, *theOops*, *numOops*)
    OopType *theObject*;
    long *startIndex*;
    const OopType *theOops*[ ];
    ArraySizeType *numOops*;

Input Arguments

theObject	The object in which to store the OOPs.
startIndex	The index into *theObject* at which to begin storing OOPs. This function does not distinguish between named and unnamed instance variables. Indices are based at the beginning of an object's array of instance variables. In that array, the object's named instance variables are followed by its unnamed instance variables.
theOops	The array of OOPs to be stored.
numOops	The number of OOPs to store.

Result Arguments

theObject	The resulting object.

Description

This function uses structural access to store multiple OOPs from a C array into a pointer object, beginning at the specified index. Note that this call cannot be used with NSCs. To add multiple OOPs to an NSC, use the **GciAddOopsToNsc** function on page 5-23.

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType newValues[10];
ArraySizeType namedSize;

/* retrieve a random instance of class Component */

aComponent = GciExecuteStr(
             "AllComponents select:[i|i.partnumber = 1234]");

/* Assign new values to the named instance variables PLUS the
  first 5 elements of aComponent's parts list */

namedSize = GciFetchNamedSize(aComponent);
newValues[0] = a_new_value;
.
.
newValues[namedSize-1 + 5] =    an_oop;
/* newValues[0..namedSize-1] are named instVar values */
/* newValues[namedSize] is first indexed instVar value */

GciStoreOops(aComponent, 1L, newValues, namedSize + 5);

/* Two ways to assign new values to the first 5 elements of
aComponent's parts list */

namedSize = GciFetchNamedSize(aComponent);
for (i=0; i < namedSize; i++) {
newValues[i] = a_value;
}
GciStoreOops(aComponent, namedSize + 1, newValues, 5);
GciStoreIdxOops(aComponent, 1, newValues, 5);
```

See Also

GciAddOopsToNsc, page 5-23
GciFetchNamedOops, page 5-90
GciFetchOop, page 5-99
GciFetchOops, page 5-101
GciFetchVaryingOop, page 5-110
GciReplaceOops, page 5-245
GciReplaceVaryingOops, page 5-247
GciStoreIdxOops, page 5-276
GciStoreNamedOops, page 5-280
GciStoreOop, page 5-282
GciStoreOops, page 5-284

GciStorePaths

Store selected multiple OOPs into an object tree.

Syntax

BoolType GciStorePaths(*theOops*, *numOops*, *paths*, *pathSizes*, *numPaths*, *newValues*, *failCount*)

const OopType	*theOops*[ ];
ArraySizeType	*numOops*;
const long	*paths*[ ];
const long	*pathSizes*[ ];
ArraySizeType	*numPaths*;
const OopType	*newValues*[ ];
long *	*failCount*;

Input Arguments

theOops	A collection of OOPs into which you want to store new values.
numOops	The size of *theOops*.
paths	An array of integers. This one-dimensional array contains the elements of all constituent paths, laid end to end.
pathSizes	An array of integers. Each element of this array is the length of the corresponding path in the *paths* array (that is, the number of elements in each constituent path).
numPaths	The number of paths in the *paths* array. This should be the same as the number of integers in the *pathSizes* array.
newValues	An array containing the new values to be stored into *theOops*.

Result Arguments

failCount	A pointer to a long that indicates which element of the *newValues* array could not be successfully stored. If all values were successfully stored, *failCount* is 0. If the *i*th store failed, *failCount* is *i*.

Return Value

Returns TRUE if all values were successfully stored. Returns FALSE if the store on any path fails for any reason.

Description

This function allows you to store multiple objects at selected positions in an object tree with a single GemBuilder call, exporting only the desired information to the database.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

Each path in the *paths* array is itself an array of longs. Those longs are offsets that specify a path along which to store objects. In each path, a positive integer x refers to an offset within an object's named instance variables, while a negative integer -x refers to an offset within an object's indexed instance variables. (The function **GciStrToPath** allows you to convert path information from its string representation, in which each element is the name of an instance variable, to the equivalent element of this paths array.)

The *newValues* array contains `(`*numOops* `*` *numPaths*`)` elements, stored in the following order:

```
[0,0]..[0,numPaths-1]..[1,0]..[1,numPaths-1]..
[numOops-1,0]..[numOops-1,numPaths-1]
```

The first element of this newValues array is stored along the first path into the first element of *theOops*. New values are then stored into the first element of *theOops* along each remaining element of the paths array. Similarly, new values are stored into each subsequent element of *theOops*, until all paths have been applied to all its elements.

The new value to be stored into object i along path j is thus represented as:

```
newValues[ ((i-1) * numPaths) + (j-1) ]
```

The expressions `i-1` and `j-1` are used because C has zero-based arrays.

If the store on any path fails for any reason, this function stops and generates a GemBuilder error. Any objects that were successfully stored before the error occurred will remain stored.

Examples

Example 1: Calling sequence for a single object and a single path

```
OopType anOop;   /* the OOP to use as the root of the path */
long   aPath[5]; /* the path itself */
long   aSize;   /* the size of the path */
OopType newValue;
long   failCount;

GciStorePaths(&anOop, 1, aPath, &aSize, 1, &newValue, &failCount);
```

Example 2: Calling sequence for multiple objects with a single path

```
OopType oops[3]; /* the OOPs to use as roots of the path */
ArraySizeType numOops; /* the number of objects */
long   aPath[5]; /* the path itself */
long   aSize;   /* the size of the path */
OopType newValues[5];
long   failCount;

GciStorePaths(oops, numOops, aPath, &aSize, 1, newValues,
          &failCount);
```

Example 3: Calling sequence for a single object with multiple paths

```
OopType anOop;   /* the OOP to use as the root of the path */
long   paths[50]; /* the paths, stored end-to-end in the array */
long   sizes[5]; /* the sizes of the paths */
ArraySizeType numPaths; /* the number of paths */
OopType newValues[5];
long   failCount;

GciStorePaths(&anOop, 1, paths, sizes, numPaths, newValues,
          &failCount);
```

Example 4: Calling sequence for multiple objects with multiple paths

```
OopType oops[3];  /* the OOPs to use as roots of the path */
ArraySizeType numOops;  /* the number of objects */
long  paths[50]; /* the paths, stored end-to-end in the array */
long  sizes[5]; /* the sizes of the paths */
ArraySizeType numPaths;  /* the number of paths */
OopType newValues[3*5]; /* new values for each path for oop1,
                           then each path for oop2, etc. */
long  failCount;

GciStorePaths(oops, numOops, paths, sizes, numPaths, newValues,
          &failCount);
```

Example 5: Integrated Code

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType SourceObjs[10];
OopType newValues[10];
long   pathSizes[10];
long   paths[10];
char *  newName;
BoolType success;

/* retrieve a random instance of class Component */
aComponent = GciExecuteStr(
            "AllComponents select:[i|i.partnumber = 1234]");

/* assign a new value to the name instVar of 5th element of
aComponent's parts list */
SourceObjs[0] = aComponent;
paths[0] = -5;
paths[1] = GciIvNameToIdx(GciFetchClass(aComponent), "name");
pathSizes[0] = 2;
newValues[1] = GciNewOop(OOP_CLASS_STRING);
newName = "Wing Assembly";
GciStoreBytes(newValues[1], 1, newName, strlen(newName));
success = GciStorePaths(sourceObjs, 1, paths, pathSizes, 1,
                        newValues);
if (!success)
 { /* error processing */ };
```

See Also

GciFetchPaths, page 5-103
GciPathToStr, page 5-213
GciStrToPath, page 5-300

GciStoreTrav

Store multiple traversal buffer values in objects.

Syntax

```
void GciStoreTrav(travBuff, behaviorFlag)
    ByteType            travBuff [ ];
    long                behaviorFlag;
```

Input Arguments

travBuff	A traversal buffer, which contains object data to be stored.
behaviorFlag	A flag that determines how the objects should be handled.

Description

The **GciStoreTrav** function stores data from the traversal buffer *travBuff* (a C-language structural description) into multiple GemStone objects. The first element in the traversal buffer is a long integer that indicates how many bytes are stored in the buffer. The remainder of the traversal buffer consists of a series of object reports. Each object report is a C structure of type **GciObjRepSType**, which includes a variable-length data area. **GciStoreTrav** stores data object by object, using one object report at a time. **GciStoreTrav** raises an error if the traversal buffer contains a report for any object of special implementation format.

GciStoreTrav allows you to reduce the number of GemBuilder calls that are required for your application program to store complex objects in the database.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

The value of *behaviorFlag* should be given by using one or more of the following GemBuilder mnemonics: GCI_STORE_TRAV_DEFAULT, GCI_STORE_TRAV_NSC_REP, GCI_STORE_TRAV_CREATE, GCI_STORE_TRAV_CREATE_PERMANENT, and GCI_STORE_TRAV_FINISH_UPDATES. The first of these must be used alone. The others

can either be used alone or can be logically "or"ed together. GCI_STORE_TRAV_CREATE_PERMANENT has no effect, however, unless it is used with GCI_STORE_TRAV_CREATE. The effect of the mnemonics depends somewhat upon the implementation format of the objects that are stored.

GciStoreTrav can create new objects and store data into them, or it can modify existing objects with the data in their object reports, or a combination of the two. By default (GCI_STORE_TRAV_DEFAULT), it can only modify existing objects, and it raises an error if an object does not already exist.

When GCI_STORE_TRAV_CREATE is used, it modifies any object that already exists and creates a new object when an object does not exist. Naturally, any new object is initialized with the data in its object report. If GCI_STORE_TRAV_CREATE_PERMANENT is not used, then a new object is created in temporary object space and the garbage collector will make the object permanent only if the object is or becomes referenced by another permanent object. But if GCI_STORE_TRAV_CREATE_PERMANENT is also used, then the object is immediately created as a permanent object, thus providing a performance gain by bypassing the garbage collector.

When GCI_STORE_TRAV_FINISH_UPDATES is used, **GciStoreTrav** automatically executes **GciProcessDeferredUpdates** after processing the last object report in the traversal buffer.

When **GciStoreTrav** modifies an existing object of byte or pointer format, it replaces that object's data with the data in its object report, regardless of *behaviorFlag*. All instance variables, named (if any) or indexed (if any), receive new values. Named instance variables for which values are not given in the object report are initialized to nil or to zero. Indexable objects may change in size; the object report determines the new number of indexed variables.

Contrast byte and pointer object handling with the default when **GciStoreTrav** modifies an existing NSC. It replaces all named instance variables of the NSC (if any), but adds further data in its object report to the unordered variables, increasing its size. If *behaviorFlag* indicates GCI_STORE_TRAV_NSC_REP, then it removes all existing unordered variables and adds new unordered variables with values from the object report.

GciStoreTrav provides automatic byte swizzling for binary floats.

Use of Object Reports

The **GciStoreTrav** function stores values in GemStone objects according to the object reports contained in *travBuff*. Each object report is of type **GciObjRepSType** (described on page 5-14), and has two parts: a header (of type **GciObjRepHdrSType**, described on page 5-15) and a value buffer (an array of values of the object's instance variables). **GciStoreTrav** uses the fields in each object report as follows:

hdr.valueBuffSize
The size (in bytes) of the value buffer, where object data is stored. If *objId* is a binary float and *valueBuffSize* differs from the actual size for objects of *objId*'s class, then **GciStoreTrav** raises an error.

hdr.namedSize
Ignored by this function.

hdr.idxSize
Used only if the object is indexable. The number of indexed variables in the object stored by **GciStoreTrav** is never less than this quantity. It may be more if the value buffer contains enough data. **GciStoreTrav** stores all the indexed variables that it finds in the value buffer. If an existing object has more indexed variables, then it also retains the extras, up to a total of *idxSize*, and removes any beyond *idxSize*. If *idxSize* is larger than the number of indexed variables in both the current object and the value buffer, then **GciStoreTrav** creates slots for elements in the stored object up to index *idxSize* and initializes any added elements to nil.

hdr.firstOffset
Ignored for NSC objects. The absolute offset into the target object at which to begin storing values from the value buffer. The absolute offset of the object's first named instance variable (if any) is one; the offset of its first indexed variable (if any) is one more than the number of its named instance variables. Values are stored into the object in the order that they appear in the value buffer, ignoring the boundary between named and indexed variables. Variables whose offset is less than *firstOffset* (if any) are initialized to nil or zero. For non-indexable objects, **GciStoreTrav** raises an error if *valueBuffSize* and *firstOffset* imply a size that exceeds the actual size of the object. If *objId* is a binary float and *firstOffset* is not one, then **GciStoreTrav** raises an error.

hdr.objId
The OOP of the object to be stored.

hdr.oclass
Used only when creating a new object, to identify its intended class.

hdr.segment
The segment in which to store the object. This value may be different from the segment where an existing object is currently stored. The value OOP_NIL implies that the current segment is used.

hdr.objSize
Ignored by this function. The function sets the object's size.

hdr.objImpl
Must be consistent with the object's implementation.

hdr.isInvariant
Boolean value. If equal to 1 (true), then set the stored object to be invariant. **GciStoreTrav** raises an error if you try to store an existing object that is already invariant.

hdr.isIndexable
Ignored by this function.

u.bytes[ ]
The value buffer of an object of byte format.

u.oops[ ]
The value buffer of an object of pointer or NSC format.

Handling Error Conditions

If you get a runtime error while executing **GciStoreTrav**, the recommended course of action is to abort the current transaction.

See Also

GciMoreTraversal, page 5-152
GciNbMoreTraversal, page 5-173
GciNbStoreTrav, page 5-181
GciNbTraverseObjs, page 5-183
GciNewOopUsingObjRep, page 5-191
GciProcessDeferredUpdates, page 5-231
GciTraverseObjs, page 5-307

GciStrKeyValueDictAt

Find the value in a symbol KeyValue dictionary at the corresponding string key.

Syntax

```
void GciStrKeyValueDictAt(theDict, keyString, value)
    OopType          theDict;
    const char *     keyString;
    OopType *        value;
```

Input Arguments

theDict	The OOP of a SymbolKeyValueDictionary.
keyString	The OOP of a key in the SymbolKeyValueDictionary.

Result Arguments

value	A pointer to the variable that is to receive the OOP of the returned value.

Description

Returns the value in symbol KeyValue dictionary *theDict* that corresponds to key *keyString*. If an error occurs or *keyString* is not found, *value* is OOP_ILLEGAL. KeyValue dictionaries do not have associations, so no association is returned. **GciStrKeyValueDictAt** is equivalent to **GciStrKeyValueDictAtObj** except that the key is a character string, not an object.

GciStrKeyValueDictAtObj

Find the value in a symbol KeyValue dictionary at the corresponding object key.

Syntax

void GciStrKeyValueDictAtObj(*theDict*, *keyObj*, *value*)
    OopType   *theDict*;
    OopType   *keyObj*;
    OopType *   *value*;

Input Arguments

theDict	The OOP of a SymbolKeyValueDictionary.
keyObj	The OOP of a key in the SymbolKeyValueDictionary.

Result Arguments

value	A pointer to the variable that is to receive the OOP of the returned value.

Description

Returns the value in symbol KeyValue dictionary *theDict* that corresponds to key *keyObj*. If an error occurs or *keyObj* is not found, *value* is OOP_ILLEGAL. KeyValue dictionaries do not have associations, so no association is returned. Equivalent to the Smalltalk expression:

```
^ #[ theDict at:keyObj]
```

GciStrKeyValueDictAtObjPut

Store a value into a symbol KeyValue dictionary at the corresponding object key.

Syntax

void GciStrKeyValueDictAtObjPut(*theDict*, *keyObj*, *theValue*)
    OopType *theDict*;
    OopType *keyObj*;
    OopType *theValue*;

Input Arguments

theDict	The OOP of the SymbolKeyValueDictionary into which the object is to be stored.
keyObj	The OOP of the key under which the object is to be stored.
theValue	The OOP of the object to be stored in the SymbolKeyValueDictionary.

Description

Adds object *theValue* to symbol KeyValue dictionary *theDict* with key *keyObj*. Equivalent to the Smalltalk expression:

```
theDict at: keyObj put: theValue
```

GciStrKeyValueDictAtPut

Store a value into a symbol KeyValue dictionary at the corresponding string key.

Syntax

void GciStrKeyValueDictAtPut(*theDict*, *keyString*, *theValue*)
    OopType *theDict*;
    const char * *keyString*;
    OopType *theValue*;

Input Arguments

theDict	The OOP of the SymbolKeyValueDictionary into which the object is to be stored.
keyString	The string key under which the object is to be stored.
theValue	The OOP of the object to be stored in the SymbolKeyValueDictionary.

Description

Adds object *theValue* to symbol KeyValue dictionary *theDict* with key *keyString*. **GciStrKeyValueDictAtPut** is equivalent to **GciStrKeyValueDictAtObjPut**, except the key is a character string, not an object.

GciStrToPath

Convert a path representation from string to numeric.

Syntax

```
BoolType GciStrToPath(aClass, pathString, maxPathSize, resultPathSize, resultPath)
    OopType            aClass;
    const char         pathString[ ];
    ArraySizeType      maxPathSize;
    ArraySizeType *    resultPathSize;
    long               resultPath[ ];
```

Input Arguments

aClass	The class of the object for which this path will apply. That is, for each instance of this class, store or fetch objects along the designated path.
pathString	The (null-terminated) path string to be converted to the equivalent numeric array.
maxPathSize	The maximum allowable size of the resulting path array (the number of elements). This is the size of the buffer that will be allocated for the resulting path array.

Result Arguments

resultPathSize	A pointer to the actual size of *resultPath.*
resultPath	The resulting array of integers. Those integers are offsets that specify a path from which to fetch objects. A positive integer x refers to an object's xth named instance variable. When a path goes through an indexed instance variable (an Array element, for example), the position of that object must be represented by a negative integer. The third element of an Array, for example, would be denoted in a path by -3.

Return Value

Returns TRUE if the path string was successfully translated to an array of integer offsets. Returns FALSE otherwise.

Description

The functions **GciFetchPaths** and **GciStorePaths** allow you to specify paths along which to fetch from, or store into, objects within an object tree.

NOTE:
This function is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

A path may be represented as a string, in which each element is the name of an instance variable (for example, '`address.zip`', in which *zip* is an instance variable of *address*.) Alternatively, a path may be represented as an array of integers, in which each step along the path is represented by the corresponding integral offset from the beginning of an object (for example, an array containing the integers 5 and 2 would represent the offsets of the fifth and second instance variables, respectively).

This function (**GciStrToPath**) converts the string representation of a path to its equivalent numeric representation, for use with **GciFetchPaths** or **GciStorePaths**.

For more information about paths, see the description of the **GciFetchPaths** function on page 5-103.

Restrictions

Note that **GciStrToPath** can convert a numeric path only if the instance variables of the specified Smalltalk class (*aClass*) are constrained in such a way that the path is guaranteed to be valid for all instances.

If your application doesn't impose GemStone constraints on classes of all objects from which you to fetch, then you'll need to maintain your paths as arrays of integers.

Error Conditions

The following errors may be generated by this function:

GCI_ERR_RESULT_PATH_TOO_LARGE
The *resultPath* was larger than the specified *maxPathSize*

RT_ERR_STR_TO_PATH_IVNAME
One of the instance variable names in the path string was invalid

RT_ERR_STR_TO_PATH_CONSTRAINT
One of the instance variables in the path string was not sufficiently constrained

Example

In the following example, assume that you've defined the class Component and populated the set AllComponents, as shown in the example for the **GciFetchVaryingOop** function on page 5-110.

```
OopType aComponent;
OopType oSourceObjs[10];
OopType oResults[10];
ArraySizeType pathSize;
long  paths[10];
ArraySizeType actualSize;

/* retrieve a random instance of class Component */

aComponent = GciExecuteStr(
                "AllComponents select:[i|i.partnumber = 1234]");

/* fetch the name instVar of the first 10 elements of
  aComponent's part list*/

actualSize = GciFetchVaryingOops(aComponent, 1, oSourceObjs, 10);
GciStrToPath(GciFetchClass(aComponent), "name", 10, &pathSize,
             paths);
GciFetchPaths(oSourceObjs, actualSize, paths, &pathSize, 1,
             oResults);p
```

See Also

GciFetchPaths, page 5-103
GciPathToStr, page 5-213
GciStorePaths, page 5-287

GciSymDictAt

Find the value in a symbol dictionary at the corresponding string key.

Syntax

```
void GciSymDictAt(theDict, keyString, value, association)
    OopType            theDict;
    const char *       keyString;
    OopType *          value;
    OopType *          association;
```

Input Arguments

theDict	The OOP of a SymbolDictionary.
keyString	The OOP of a key in the SymbolDictionary.

Result Arguments

value	A pointer to the variable that is to receive the OOP of the returned value.
association	A pointer to the variable that is to receive the OOP of the association.

Description

Returns the value in symbol dictionary *theDict* that corresponds to key *keyString*. If an error occurs or *keyString* is not found, *value* is OOP_ILLEGAL. If *association* is not NULL and an error does not occur, stores the OOP of the association for *keyString* at **association*, or stores OOP_ILLEGAL if *keyString* was not found. Equivalent to **GciSymDictAtObj** except that the key is a character string, not an object.

To operate on kinds of Dictionary other than SymbolDictionary, such as KeyValueDictionary, use **GciPerform** or **GciSendMsg**, since the KeyValueDictionary class is implemented in Smalltalk. If your dictionary will be large (greater than 20 elements) a KeyValueDictionary is more efficient than a SymbolDictionary.

GciSymDictAtObj

Find the value in a symbol dictionary at the corresponding object key.

Syntax

void GciSymDictAtObj(*theDict*, *keyObj*, *value*, *association*)
OopType *theDict*;
OopType *keyObj*;
OopType * *value*;
OopType * *association*;

Input Arguments

theDict	The OOP of a SymbolDictionary.
keyObj	The OOP of a key in the SymbolDictionary.

Result Arguments

value	A pointer to the variable that is to receive the OOP of the returned value.
association	A pointer to the variable that is to receive the OOP of the association.

Description

Returns the value in symbol dictionary *theDict* that corresponds to key *keyObj*. If an error occurs or *keyObj* is not found, *value* is OOP_ILLEGAL. If *association* is not NULL and an error does not occur, stores the OOP of the association for *keyObj* at **association*, or stores OOP_ILLEGAL if *keyObj* was not found. Similar to the Smalltalk expression:

```
^ #[ theDict at:keyObj , theDict: associationAt:keyObj]
```

GciSymDictAtObjPut

Store a value into a symbol dictionary at the corresponding object key.

Syntax

void GciSymDictAtObjPut(*theDict*, *keyObj*, *theValue*)
OopType *theDict*;
OopType *keyObj*;
OopType *theValue*;

Input Arguments

theDict	The OOP of the SymbolDictionary into which the value is to be stored.
keyObj	The OOP of the key under which the value is to be stored.
theValue	The OOP of the object to be stored in the SymbolDictionary.

Description

Adds object *theValue* to symbol dictionary *theDict* with key *keyObj*. Equivalent to the Smalltalk expression:

```
theDict at: keyObj put: theValue
```

GciSymDictAtPut

Store a value into a symbol dictionary at the corresponding string key.

Syntax

```
void GciSymDictAtPut(theDict, keyString, theValue)
    OopType         theDict;
    const char *    keyString;
    OopType         theValue;
```

Input Arguments

theDict	The OOP of the SymbolDictionary into which the object is to be stored.
keyString	The string key under which the object is to be stored.
theValue	The OOP of the object to be stored in the SymbolDictionary.

Description

Adds object *theValue* to symbol dictionary *theDict* with key *keyString*. Equivalent to **GciSymDictAtObjPut**, except the key is a character string, not an object.

GciTraverseObjs

Traverse an array of GemStone objects.

Syntax

BoolType GciTraverseObjs(*theOops*, *numOops*, *travBuff*, *level*)
    const OopType *theOops*[ ];
    ArraySizeType *numOops*;
    ByteType *travBuff*[ ];
    long *level*;

Input Arguments

theOops	An array of OOPs representing the objects to traverse.
numOops	The number of elements in theOops.
level	Maximum traversal depth. When the level is 1, an object report is written to the traversal buffer for each element in theOops. When level is 2, an object report is also obtained for the instance variables of each level-1 object. When level is 0, the number of levels in the traversal is not restricted.

Result Arguments

travBuff	A buffer in which the results of the traversal will be placed.

Return Value

Returns FALSE if the traversal is not yet completed. Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal**.

Description

This function allows you to reduce the number of GemBuilder calls that are required for your application program to obtain information about complex objects in the database.

NOTE:

This function is most useful with applications that are linked with GciRpc (the

"remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch***... and* **GciStore***... functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

There are no built-in limits on how much information can be obtained in the traversal. You can use the *level* argument to restrict the size of the traversal.

GciTraverseObjs provides automatic byte swizzling for binary floats.

Organization of the Traversal Buffer

The first element placed in a traversal buffer is a long integer that indicates how many bytes were actually stored in the buffer by this function. The remainder of the traversal buffer consists of a series of object reports, each of which is of type **GciObjRepSType**, as follows:

`long`
`actualBufferSize;`
How many bytes were actually stored in the traversal buffer by this function (used by **GciFindObjRep**); `travBuff[actualBufferSize - 1]` is the final byte written.

`GciObjRepSType`
`report_i;`
An object report.

`GciObjRepSType`
`report_j;`
Another object report.

In order for the traversal buffer to accommodate *m* objects, each of which is of size *n* bytes, your application should allocate at least enough memory so that the traversal buffer's size can be assigned according to the following formula:

```
travBuffSize = sizeof(long) +
            m * (long)GCI_ALIGN(sizeof(GciObjRepHdrSType) + n);
```

The macro GCI_ALIGN ensures that the value buffer portion of each object report begins at a word boundary.

This function ensures that each object report header and value buffer begins on a word boundary. To provide proper alignment, 0 to 3 bytes may be inserted between each header and value buffer.

The Value Buffer

The object report's *value buffer* begins at the first byte following the object report header. For byte objects, the value buffer (*u.bytes*[ ]) is an array of type **ByteType**; for pointer objects and NSCs, the buffer (*u.oops*[ ]) is an array of type **OopType**. The size of the report's value buffer (*hdr.valueBuffSize*) is the number of bytes of the object's value returned by this traversal. That number is no greater than the size of the object.

To obtain a pointer to the value buffer, use the macro

```
GCI_VALUE_BUFF(theReport)
```

How This Function Works

This section explains how **GciTraverseObjs** stores object reports in the traversal buffer and values in the value buffer.

1. First, **GciTraverseObjs** verifies that the traversal buffer is large enough to accommodate at least one object report header (**GciObjRepHdrSType**). If the buffer is too small, GemBuilder returns an error.

```
/* useful macros to simplify the code */
#define OHDR_SIZE ((long)GCI_ALIGN(sizeof(GciObjRepHdrSType)))
#define MIN_TBUF_SIZE (OHDR_SIZE + sizeof(long))

/* check traversal buffer for minimum size */
if (travBuffSize < (MIN_TBUF_SIZE))
    /* ERROR: not big enough for one object report header */
```

2. For each object in the traversal, **GciTraverseObjs** discovers if there is enough space left in the traversal buffer to store both the object report header and the object's values. If there isn't enough space remaining, the function returns 0, and your program can call **GciMoreTraversal** to continue the traversal. Otherwise (if there is enough space), the object's values are stored in the traversal buffer.
3. When there are no more objects left to traverse, **GciTraverseObjs** returns a nonzero value to indicate that the traversal is complete.

Special Objects

For each occurrence of an object with a special implementation (that is, an instance of SmallInteger, Character, Boolean, or UndefinedObject) contained in *theOops*, this function will return an accurate object report. For any special object encountered at some deeper point in the traversal, no object report will be generated.

Authorization Violations

If the user is not authorized to read some object encountered during the traversal, the traversal will continue. No value will be placed in the object report's value buffer, but the report for the forbidden object will contain the following values:

hdr.valueBuffSize `0`
hdr.namedSize `0`
hdr.idxSize `0`
hdr.firstOffset `1`
hdr.objId `theOop`
hdr.oclass `OOP_NIL`
hdr.segment `OOP_NIL`
hdr.objSize `0`
hdr.objImpl `GC_FORMAT_SPECIAL`
hdr.isInvariant `0`

Incomplete Object Reports

To check the completeness of an object report's value, examine its *hdr.objSize* field, which contains the size of the GemStone object represented by the object report. Then examine the object report's *hdr.valueBuffSize*, which contains the size of the portion of the object that was actually imported into the object report. If the two numbers are not the same, then you have an incomplete object report.

Continuing the Traversal

When the amount of information obtained in a traversal exceeds the amount of available memory (as specified with *travBuffSize*), your application can break the traversal into manageable amounts of information by issuing repeated calls to **GciMoreTraversal**. Generally speaking, an application can continue to call **GciMoreTraversal** until it has obtained all requested information.

During the entire sequence of **GciTraverseObjs** and **GciMoreTraversal** calls that constitute a traversal, any single object report will be returned exactly once. Regardless of the connectivity of objects in the GemStone database, only one report will be generated for any non-special object.

When Traversal Can't Be Continued

Naturally, GemStone will not continue an incomplete traversal if there is any chance that changes to the database in the intervening period might have invalidated the previous report or changed the connectivity of the objects in the path of the traversal. Specifically,

GemStone will refuse to continue a traversal if, in the interval before attempting to continue, you:

- Modify the objects in the database directly, by calling any of the **GciStore**... or **GciAdd**... functions;
- Call one of the Smalltalk message-sending functions **GciSendMsg**, **GciPerform**, **GciContinue**, or any of the **GciExecute**... functions;
- Abort your transaction, thus invalidating any subsequent information from that traversal.

Any attempt to call **GciMoreTraversal** after one of these actions will generate an error.

Note that this holds true across multiple GemBuilder applications sharing the same GemStone session. Suppose, for example, that you were holding on to an incomplete traversal buffer and the user moved from the current application to another, did some work that required executing Smalltalk code, and then returned to the original application. You would be unable to continue the interrupted traversal.

Example

The following example fetches an object report on a Document object stored in the GemStone database, then obtains more detailed information about the title of that document.

1. First, the example calls **GciTraverseObjs** to fetch information about each element of the *docList* (a collection of Document OOPs) from the database into the first traversal buffer (*buff1*).
2. Next, the example calls **GciFindObjRep** to scan the buffer for a report on the desired document (*myDocument*).
3. After locating that object report in the traversal buffer, the program obtains the necessary information from the buffer: in this case, the class and size of *myDocument*, a pointer to its value buffer, and the title and author of that document.
4. The example calls **GciFindObjRep** again to hunt for a report on *myDocument*'s *title* instance variable.

5. Subsequent **GciMoreTraversal** calls may be used to obtain additional object reports pertinent to elements in the specified *docList*.

```
BoolType atEnd;

atEnd = GciTraverseObjs(docList, numOops, buff1, maxSz, level);

/* use GciFindObjRep to scan buff1 for a report on myDocument
(an element in the docList) */
myDocumentReport = GciFindObjRep(buff1, myDocument);

/* get all desired information from buff1 */
myClass = myDocumentReport->class;
mySize = myDocumentReport->size;
value = GCI_VALUE_BUFF(myDocumentReport);
myTitle = value[0];
myAuthor = value[1];

/* Intervening code goes here, in place of this comment */


/* now look for a report on myDocument's title instance var */
titleReport = GciFindObjRep(buff1, myTitle);
theTitle = GCI_VALUE_BUFF(titleReport);

if (!atEnd) /* and you want more information */
  atEnd = GciMoreTraversal(buff2, maxSz);
```

See Also

GciFindObjRep, page 5-117
GciMoreTraversal, page 5-152
GciNbMoreTraversal, page 5-173
GciNbStoreTrav, page 5-181
GciNbTraverseObjs, page 5-183
GciNewOopUsingObjRep, page 5-191
GciObjRepSize, page 5-198
GciStoreTrav, page 5-292

GciUnsignedLongToOop

Find a GemStone object that corresponds to a C unsigned long integer.

Syntax

OopType GciUnsignedLongToOop(*anUnsignedLong*)
unsigned long *anUnsignedLong*;

Input Arguments

anUnsignedLong The C unsigned long integer to be translated into an object.

Return Value

The **GciUnsignedLongToOop** function returns the OOP of a GemStone object whose value is equivalent to the C unsigned long integer value of *anUnsignedLong*.

Description

The **GciUnsignedLongToOop** function translates the C long integer value *anUnsignedLong* into a GemStone object that has the same value.

If the value is in the range 0 .. 1073741823, the resulting object is a SmallInteger. If the value is larger than 1073741823, the resulting object is a LargePositiveInteger.

See Also

GciLongToOop, page 5-148
GCI_LONG_TO_OOP, page 5-150
GciOopToLong, page 5-209
GCI_OOP_TO_LONG, page 5-211

GciUserActionInit

Declare user actions for GemStone.

Syntax

void GciUserActionInit( )

Description

GciUserActionInit is implemented by the application developer, but it is called by **GciIinit**. It enables Smalltalk to find the entry points for the application's user actions, so that they can be executed from the database.

GCI_VALUE_BUFF

(MACRO) Find a pointer to the value buffer of an object report.

Syntax

char * GCI_VALUE_BUFF(*theReport*)

Input Arguments

theReport The object report (assumed to be of type GciObjRepSType *) for whose value buffer the pointer is sought.

Result Value

A C pointer to the value buffer of the input object report.

Description

This macro is used during object traversals to obtain a pointer to the value buffer of an object report.

This macro is most useful with applications that are linked with GciRpc (the "remote procedure call" version of GemBuilder*). If your application will be linked with GciLnk (the "linkable"* GemBuilder*), you'll usually achieve best performance by using the simple* **GciFetch**... *and* **GciStore**... *functions rather than object traversal. For more information, see "GciRpc and GciLnk" on page 2-1.*

Example

```
char * myBytes;
myBytes = (char *)GCI_VALUE_BUFF(&myObjectReport);
```

See Also

GciMoreTraversal, page 5-152
GciNewOopUsingObjRep, page 5-191
GciTraverseObjs, page 5-307

GciVersion

Return a string that describes the GemBuilder version.

Syntax

const char* GciVersion( )

Description

GciVersion returns a string terminated by 0. Version fields in the string are delimited by periods (.). The first field is the major version number, GCI_VER_MAJOR_DIGIT. The second field is the minor version number, GCI_VER_MINOR_DIGIT. Any number of additional fields may exist to describe the exact release of GemBuilder.

For more version information, use the methods in class System in the Version Management category.

Appendix A

Reserved OOPs

The GemBuilder for C include file `gcioop.ht` defines C mnemonics for the OOPs of certain GemStone objects that are already defined in your GemStone software package. Your C application can compare all these mnemonics with any value of type `OopType`. However, the value of any mnemonic is subject to change without notice in future software releases. Your C application should refer to the OOPs of predefined GemStone objects *by mnemonic name only*.

The following mnemonic names for predefined GemStone objects are available to C programs:

- A value that, strictly speaking, is not an object at all, but that represents a value that is never used to represent any object in the database. You can use this mnemonic to test whether or not an OOP is valid, that is, whether or not it actually points to any GemStone object.
  - OOP_ILLEGAL
- Special objects
  - OOP_NIL (*nil*)
  - OOP_FALSE (*FALSE*)
  - OOP_TRUE (*true*)

- Instances of SmallInteger
  - OOP_MinusOne
  - OOP_Zero
  - OOP_One
  - OOP_Two
  - OOP_MAX_SMALL_INT
  - OOP_MIN_SMALL_INT
- Instances of Character
  - OOP_ASCII_NUL represents the first ASCII character OOP
  - 255 other OOPs represent the remaining ASCII characters, but they have no mnemonics
- Instances of JISCharacter
  - OOP_FIRST_JIS_CHAR_OOP
  - OOP_LAST_JIS_CHAR_OOP
- The GemStone Smalltalk kernel classes
  - OOP_CLASS_*className* (in this case, the class name is in capital letters, with words separated by underscore characters)
  - OOP_LAST_KERNEL_OOP (which has the same value as the last class)
  - OOP_CLASS_EXCEPTION
- The GemStone error dictionary
  - OOP_GEMSTONE_ERROR_CAT
- The cluster bucket category
  - OOP_ALL_CLUSTER_BUCKETS

Appendix

B Network Resource String Syntax

This appendix describes the syntax for network resource strings. A network resource string (NRS) provides a means for uniquely identifying a GemStone file or process by specifying its location on the network, its type, and authorization information. GemStone utilities use network resource strings to request services from a NetLDI.

B.1 Overview

One common application of NRS strings is the specification of login parameters for a remote process (RPC) GemStone application. An RPC login typically requires you to specify a GemStone repository monitor and a Gem service on a remote server, using NRS strings that include the remote server's hostname. For example, to log in from Topaz to a Stone process called "gemserver50" running on node "handel", you would specify two NRS strings:

```
topaz> set gemstone !@handel!gemserver50
topaz> set gemnetid !@handel!gemnetobject
```

Many GemStone processes use network resource strings, so the strings show up in places where command arguments are recorded, such as the GemStone log file. Looking at log messages will show you the way an NRS works. For example:

```
Opening transaction log file for read,
filename = !tcp@oboe#dbf!/user1/gemstone/data/tranlog0.dbf
```

An NRS can contain spaces and special characters. On heterogeneous network systems, you need to keep in mind that the various UNIX shells have their own rules for interpreting these characters. If you have a problem getting a command to work with an NRS as part of the command line, check the syntax of the NRS recorded in the log file. It may be that the shell didn't expand the string as you expected.

NOTE
Before you begin using network resource strings, make sure you understand the behavior of the software that will process the command.

See each operating system's documentation for a full discussion of its own rules. For example, under the UNIX C shell, you must escape an exclamation point (!) with a preceding backslash (\) character:

```
% waitstone \!tcp@oboe\!gemserver50 -1
```

If there is a space in the NRS, you can replace the space with a colon (:), or you can enclose the string in quotes (" "). For example, the following network resource strings are equivalent:

```
% waitstone !tcp@oboe#auth:user@password!gemserver50

% waitstone "!tcp@oboe#auth user@password!gemserver50"
```

B.2 Defaults

The following items uniquely identify a network resource:

- communications protocol— such as TCP/IP, DECnet, or SNA
- destination node—the host that has the resource
- authentication of the user—such as a system authorization code
- resource type—such as server, database extent, or task
- environment—such as a NetLDI, a directory, or the name of a log file
- resource name—the name of the specific resource being requested.

A network resource string can include some or all of this information. In most cases, you need not fill in all of the fields in a network resource string. The information required depends upon the nature of the utility being executed and the task to be accomplished. Most GemStone utilities provide some context-sensitive defaults. For example, the Topaz interface prefixes the name of a Stone process with the **#server** resource identifier.

When a utility needs a value for which it does not have a built-in default, it relies on the system-wide defaults described in the syntax productions in “Syntax” on page B-4. You can supply your own default values for NRS modifiers by defining an environment variable named GEMSTONE_NRS_ALL in the form of the *nrs-header* production described in the Syntax section. If GEMSTONE_NRS_ALL defines a value for the desired field, that value is used in place of the system default. (There can be no meaningful default value for “resource name.”)

A GemStone utility picks up the value of GEMSTONE_NRS_ALL as it is defined when the utility is started. Subsequent changes to the environment variable are not reflected in the behavior of an already-running utility.

When a client utility submits a request to a NetLDI, the utility uses its own defaults and those gleaned from its environment to build the NRS. After the NRS is submitted to it, the NetLDI then applies additional defaults if needed. Values submitted by the client utility take precedence over those provided by the NetLDI.

B.3 Notation

Terminal symbols are printed in boldface. They appear in a network resource string as written:

#server

Nonterminal symbols are printed in italics. They are defined in terms of terminal symbols and other nonterminal symbols:

username ::= *nrs-identifier*

Items enclosed in square brackets are optional. When they appear, they can appear only one time:

address-modifier ::= [*protocol*] [**@** *node*]

Items enclosed in curly braces are also optional. When they appear, they can appear more than once:

nrs-header ::= **!** [*address-modifier*] {*keyword-modifier*} **!**

Parentheses and vertical bars denote multiple options. Any single item on the list can be chosen:

protocol ::= (**tcp** | **decnet** | **serial** | **default**)

B.4 Syntax

nrs ::= [*nrs-header*] *nrs-body*

where:

nrs-header ::= **!** [*address-modifier*] {*keyword-modifier*} [*resource-modifier*]**!**
All modifiers are optional, and defaults apply if a modifier is omitted. The value of an environment variable can be placed in an NRS by preceding the name of the variable with "`$`". If the name needs to be followed by alphanumeric text, then it can be bracketed by "{" and "}". If an environment variable named `foo` exists, then either of the following will cause it to be expanded: `$foo` or `${foo}`. Environment variables are only expanded in the *nrs-header*. The *nrs-body* is never parsed.

address-modifier ::= [*protocol*] [@ *node*]
Specifies where the network resource is.

protocol ::= (**tcp** | **decnet** | **serial** | **default**)
Supports heterogeneous connections by predicating address on a network type. If no protocol is specified, GCI_NET_DEFAULT_PROTOCOL is used. On UNIX hosts, this default is **tcp**.

node ::= *nrs-identifier*
If no node is specified, the current machine's network node name is used. The identifier may also be an Internet-style numeric address. For example:

```
!tcp@120.0.0.4#server!cornerstone
```

nrs-identifier ::= *identifier*
Identifiers are runs of characters; the special characters `!`, `#`, `$`, `@`, `^` and white space (blank, tab, newline) must be preceded by a "`^`". Identifiers are words in the UNIX sense.

keyword-modifier ::= (*authorization-modifier* | *environment-modifier*)
Keyword modifiers may be given in any order. If a keyword modifier is specified more than once, the latter replaces the former. If a keyword modifier takes an argument, then the keyword may be separated from the argument by a space or a colon.

authorization-modifier ::= ((#**auth** | #**encrypted**) [**:**] *username* [@ *password*] | #**krb**)

#**auth** specifies a valid user on the target network. A valid password is needed only if the resource type requires authentication. #**encrypted** is used by GemStone utilities. If no authentication information is specified, the system will try to get it from the `.netrc` file. This type of authorization is the default.

#**krb** specifies that kerberos authentication is to be used instead of a user name and password.

username ::= *nrs-identifier*

If no user name is specified, the default is the current user.
(See the earlier discussion of *nrs-identifier*.)

password ::= *nrs-identifier*

If no password is specified, the system will try to obtain it from the user's `.netrc` file. (See the earlier discussion of *nrs-identifier*.)

environment-modifier ::= (#**netldi** | #**dir** | #**log**) [**:**] *nrs-identifier*

#**netldi** causes the named NetLDI to be used to service the request. If no NetLDI is specified, the default is `netldi50`. (See the earlier discussion of *nrs-identifier*.)

#**dir** sets the default directory of the network resource. It has no effect if the resource already exists. If a directory is not set, the pattern "`%H`" (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

#**log** sets the name of the log file of the network resource. It has no effect if the resource already exists. If the log name is a relative path, it is relative to the working directory. If a log name is not set, the pattern "`%N%P%M.log`" (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

The argument to #**dir** or #**log** can contain patterns that are expanded in the context of the created resource. The following patterns are supported:

%H home directory
%M machine's network node name
%N executable's base name
%P process pid
%U user name
%% %

resource-modifier ::= (**#server** | **#spawn** | **#task** | **#dbf** | **#monitor** | **#file**)

Identifies the intended purpose of the string in the *nrs-body*. An NRS can contain only one resource modifier. The default resource modifier is context sensitive. For instance, if the system expects an NRS for a database file, then the default is **#dbf**.

#server directs the NetLDI to search for the network address of a server, such as a Stone or another NetLDI. If successful, it returns the address. The *nrs-body* is a network server name. A successful lookup means only that the service has been defined; it does not indicate whether the service is currently running. A new process will not be started. (Authorization is needed only if the NetLDI is on a remote node and is running in secure mode.)

#task starts a new Gem. The *nrs-body* is a NetLDI service name (such as "gemnetobject"), followed by arguments to the command line. The NetLDI creates the named service by looking first for an entry in `$GEMSTONE/bin/services.dat` , and then in the user's home directory for an executable having that name. The NetLDI returns the network address of the service. (Authorization is needed to create a new process unless the NetLDI is in guest mode.) The **#task** resource modifier is also used internally to create page servers.

#dbf is used to access a database file. The *nrs-body* is the file spec of a GemStone database file. The NetLDI creates a page server on the given node to access the database and returns the network address of the page server. (Authorization is needed unless the NetLDI is in guest mode).

#spawn is used internally to start the garbage-collection Gem process.

#monitor is used internally to start up a shared page cache monitor.

#file means the *nrs-body* is the file spec of a file on the given host (not currently implemented).

nrs-body ::= unformatted text, to end of string

The *nrs-body* is interpreted according to the context established by the *resource-modifier*. No extended identifier expansion is done in the *nrs-body*, and no special escapes are needed.

Appendix C Linking to Static User Action Code

Shared libraries on Unix platforms cannot link to non-shared code, which creates a problem if a user action needs to call code in a static library and no equivalent shared library is available. For this reason, GemBuilder provides a way to link non-shared code into a custom Gem. Static linking requires that all entry points be resolved at link time. You must have a C development environment, and if you create an RPC product, you must ship the libraries with it.

C.1 Creating the Custom Gem

A Gem is customized if you have defined user actions that it can execute in its own process. To produce a custom Gem with static user action functions, complete the following steps:

Step 1. Modify the user action code.

The static user action code must define **StaticUserActionInit**, which will be called by the Gem the static code is linked into, and the code must include the files `staticua.hf` and `gci.hf`. Include `gciuser.hf` also, if you wish to use the GCI_DECLARE_ACTION macro.

Step 2. Test the static user action code.

You can implement your own Topaz to test the code. Use the **GciLoadUserActionLibrary** function to implement your version of the Topaz **loadua** command. To use a custom Topaz, just execute it.

Step 3. Link the static code into a custom Gem.

Once the code is compiled, you can create a custom Gem by linking your user action code with `gemrpcobj.o`, which contains the object code for an RPC Gem.

Each command line illustrates how to link a set of user actions for a Gem. The link step produces a custom Gem executable named *usrgem*.

Solaris (Sun):

```
$ cc -o   usrgem  usract.o
   $GEMSTONE/lib/gemrpcobj.o -lm -lsocket -lnsl
```

HP-UX (HP):

```
$ cc -z -o   usrgem  usract.o
   $GEMSTONE/lib/gemrpcobj.o -lm
```

The `-z` switch is highly recommended.

AIX (IBM):

```
$ cc -o   usrgem  usract.o
   $GEMSTONE/lib/gemrpcobj.o -lm
```

C.2 Deploying Static User Actions for Custom Gems

This section describes how to make static user actions available to custom RPC Gem processes at run time. A linked Gem has the same user actions available as does the application with which it is linked.

How GemStone Starts Gem Processes

When an application logs in to GemStone, an RPC Gem process must be started for the GemStone session, unless it is the first GemStone login from a linked application. (One Gem is already running in the application process in that case.) The application calls the **GciLogin** function. GemBuilder then sends a network resource string (NRS) to the NetLDI process on the machine where the RPC Gem process is to be created. The NRS tells the NetLDI everything it needs to know in order to start the Gem process.

An RPC Gem is a GemStone service. The NRS specifies at least the service name of the RPC Gem. It can also supply more information to govern the operation of the process. Consult Appendix B, "Network Resource String Syntax," for details.

To make the custom Gem available to any application user, it is generally necessary to define a new service name in the GemStone services file in a shared GemStone installation. Then either the application or application user must specify that service name as the default for starting Gems.

On Unix platforms, a custom Gem requires a specially-linked executable file. The new GemStone service generally runs the executable file from a special script that sets the execution environment.

By default, GemBuilder starts RPC Gems by sending an NRS with just the default service name, `gemnetobject`. If the GEMSTONE_NRS_ALL environment variable is set in the application process, its value overrides the GemStone default. The application programmer can override these values by calling **GciSetNet**. Such means can be used to specify different settings for the `gemnetobject` service, or to specify a different service altogether. They can also supply a different (non-GemStone) default for the application. **GciSetNet** can also be used to specify the NRS for RPC Gems directly based upon user input.[1]

When GemBuilder asks the NetLDI to start an RPC Gem process, the NetLDI reads the GemStone services file. This file, named `services.dat`, is found in the GemStone `bin` directory. The file associates a service name with an executable file to be run when the service is requested. Each service name in the file must be unique, but more than one service name can be associated with a single executable file. The NetLDI executes the file associated with the service name to start the new process.

For more details on placing custom Gems into your GemStone installation, please see your *GemStone System Administration Guide*.

Starting a Private Custom Gem Under Unix

Assume that you have built (linked) a custom Gem, *usrgem*, and have placed it in the directory *usrgemdir*. Assume also that the user actions that it contains have already been debugged. You should not run this configuration with user actions that have not been debugged (see "Risk of Database Corruption" on page 4-4 for

1. Topaz sets its RPC Gem NRS from user input with the **set gemnetid** command. GemBuilder for Smalltalk sets the RPC Gem NRS when you add or edit the Gem service of a session entry from the GemStone Session Browser.

details). To use *usrgem* yourself, but without allowing others to use it for now, perform the following steps:

Step 1. Select a name for a script that will execute the custom Gem while it remains private. Be sure that the name does not match any entry in the GemStone services file, `$GEMSTONE/bin/services.dat`. We will call this script *run_usrgem* for the purposes of this procedure.

Step 2. Determine which shell you wish to use for *run_usrgem*. GemStone recommends using the Bourne shell unless your standard environment is C shell and you also wish the user actions to depend on environment variables. Copy the appropriate GemStone script to your home directory:

For Bourne shell:

`$ cp $GEMSTONE/sys/gemnetobject $HOME/` *run_usrgem*

For C shell:

`$ cp $GEMSTONE/sys/gemnetobjcsh $HOME/` *run_usrgem*

Step 3. Modify *run_usrgem*.

❐ Modify the line that defines the script's **gemname** variable. Set its value to *usrgem*.

❐ Modify the line that defines the script's **gemdir** variable. Set its value to *usrgemdir*.

❐ Find the lines that define the script's **systemConfig** and **exeConfig** variables. If you wish to specify custom GemStone configuration files for the custom Gem, modify the appropriate line(s) accordingly.

The Gem service name associated with the private *usrgem* is now *run_usrgem*, because the script of that name is in your home directory. To run the custom Gem, supply its service name (NRS) as the application requires. From Topaz, you can use the **set gemnetid** command. If you are using GemBuilder for Smalltalk, add or edit the Gem service of a session entry from the GemStone Session Browser.

C.3 Name Conflicts with Dynamic User Actions

When user actions are installed in a process, they are given a name by which GemBuilder refers to them. These names must be unique. In case of a name conflict, a static user action in a custom Gem always takes precedence. If the custom Gem attempts to dynamically load a user action library containing a user

action with the same name as one the Gem's static user actions, the load operation fails. If an application attempts to load a user action library with a user action by the same name as one of the Gem's static user actions, the load operation succeeds, but the application user action with the conflicting name is ignored. The static Gem user action is always used.

Index

A

aborting transactions 1-7
access, structural (to objects),
  see structural access
adding
  OOPs to an NSC 1-20, 5-22, 5-23
alignment of traversal buffer 5-27, 5-192
application
  binding 2-3, 4-4
  improving performance 1-22, 1-25, 1-32, 5-103, 5-152, 5-173, 5-181, 5-183, 5-238, 5-239, 5-287, 5-292, 5-307
  linking 1-2
application user action 3-10
application user actions 3-2
authorization
  traversal 5-310
  violation, what to do 1-7

B

binding
  build-time 2-5, 4-7
binding to GemBuilder 2-3, 4-4
boolean
  converting to an object 5-32
  represented as a special object 1-9
break, see interrupt
  hard, see hard break
  soft, see soft break
build-time binding 2-5, 4-7
byte object
  fetching bytes from 1-17, 5-79, 5-81
  fetching the size 5-92, 5-108
  implementation type 1-17, 5-94
  storing bytes in 1-18, 5-266, 5-268, 5-270

C

C mnemonic
  sizes and offsets into objects 5-10

C representation of objects,
 see structural access
C types defined for GemBuilder functions 5-10, 5-11
call stack
 clearing 1-30, 5-45
calling
 the virtual machine 5-49, 5-68, 5-72, 5-74, 5-161, 5-165, 5-167, 5-169, 5-175, 5-216, 5-255
 user actions
  from GemStone 3-9
changed object, and re-reading 5-25, 5-34
changing class definitions 1-15
character
 converting to an object 5-36
 instance defined in GemStone A-2
 represented as a special object 1-9
checking for GemBuilder errors 1-29, 5-66
clamped object traversal 5-40, 5-137, 5-158
class
 compiling methods 1-3, 5-42
 fetching an object's 1-17, 5-85
 modifying 1-15
 object report 5-117
clearing the call stack 1-30
cluster bucket
 mnemonic for category A-2
committing transactions 1-6, 5-48, 5-160
compiling
 applications 4-3
 C code 4-2
 class methods 1-3, 5-42
 instance methods 1-3, 5-130
 methods 1-3, 5-130
 user actions 4-3
concurrency conflict 5-48, 5-160
 what to do 1-6
configuration files 5-129, 5-130
constraint violation, what to do 1-7
context
 call stack 5-45
 error handling 1-30
 of GemStone system 5-49, 5-51, 5-161, 5-162
continuable error 5-49, 5-51, 5-161, 5-162
continuing
 after an error 1-29, 5-49, 5-51, 5-161, 5-162
 traversal 5-153, 5-310
controlling
 sessions 5-48, 5-142, 5-146, 5-160, 5-260
 transactions 5-20, 5-48, 5-155, 5-160
converting between
 objects and booleans 5-32, 5-203, 5-204
 objects and characters 5-36, 5-205, 5-206
 objects and floating point numbers 5-119, 5-207
 objects and integers 5-148, 5-150, 5-209, 5-211, 5-313
 path representations 1-25, 5-213, 5-300
 special objects and C values 1-9
creating
 class methods 5-42
 database objects 1-3
 GemStone sessions 1-6, 5-142, 5-260
 instances of a GemStone class 1-16
 objects 1-21, 5-191
current session, defined 1-6
custom Gem executable
 user action C-2

D

debugging
 function, enabling 5-59
 information, finding 5-233, 5-258
 use GciRpc 2-2, 4-4
 user action 3-10
default
 directory, host file access 1-28
 login parameter value 5-128, 5-260

defining
  new methods 1-3
disabling
  error handlers 5-258
DLLs 2-3

E

enabling
  debugging functions 5-59
  error handlers 5-258
enumerating named instance variables 5-44, 5-140
error
  checking 1-29, 5-66
  continuing execution after 5-49, 5-51, 5-161, 5-162
  dictionary 1-8, A-2
  handling 1-29, 5-229, 5-233, 5-258
  jump buffer 1-29, 5-229, 5-233, 5-258
  mnemonics 5-10
  polling 1-29, 5-66
executing code in
  GemBuilder, advantages over GemStone 1-3
  GemStone 1-3, 1-13, 5-68, 5-70, 5-72, 5-74, 5-76, 5-165, 5-167, 5-169, 5-171, 5-179, 5-222
    advantages over GemBuilder 1-3
  host file access method 1-28
executing user action 3-10
export set 1-31
exporting objects to GemStone 1-3, 1-16

F

false, GemStone special object 1-8, 1-9, 1-10, 5-32, A-1
fetching
  bytes from a byte object 1-17, 5-79, 5-81
  class 1-17, 5-85
  object implementation format 1-17, 5-94
  object size 5-92, 5-108, 5-115
  objects by using paths 1-25, 5-103
  OOPs from a pointer object 1-19, 5-88, 5-90, 5-110, 5-113
  OOPs from an NSC 1-20, 5-99, 5-101
finding
  debugging information 5-233, 5-258
  object reports in a traversal buffer 5-117
  objects in a traversal buffer 1-24
floating point number
  as a byte object 1-18
  converting to an object 5-119
format of an object, fetching 1-17, 5-94

G

garbage collection 5-238, 5-239
  saving and releasing objects 1-31
GemBuilder 5-27, 5-32, 5-36, 5-137, 5-147, 5-150, 5-200, 5-201, 5-202, 5-204, 5-206, 5-211, 5-315
GCI_DECLARE_ACTION 3-4
GCI_LONGJMP 5-232
GCI_SETJMP 5-232
GCI_SIG_JMP_BUF_TYPE 5-232
GciAbort 5-20
GciAddOopsToNsc 5-23
GciAddOopToNsc 5-22
GciAddSaveObjsToReadSet 5-24
GciAlteredObjs 5-25
GciAppendBytes 5-28
GciAppendChars 5-29
GciAppendOops 5-30
GciBegin 5-31
GciCallInProgress 1-27, 5-33
GciCheckAuth 5-34
GciClampedTrav 5-37
GciClampedTraverseObjs 5-40

GciClassMethodForClass 5-42
GciClassNamedSize 5-44
GciClearStack 5-45
GciCommit 5-48
GciContinue 5-49
GciContinueWith 5-51
GciCreateByteObj 5-53
GciCreateOopObj 5-55
GciCTimeToDateTime 5-57
GciDateTimeSType 5-12
GciDateTimeToCTime 5-58
GciDbgEstablish 5-59
GciDirtyObjsInit 5-61
GciDirtySaveObjs 5-62
GciEnableSignaledErrors 5-64
GciErr 5-66
GciErrSType 5-12
GciExecute 5-68
GciExecuteFromContext 5-70
GciExecuteStr 5-72
GciExecuteStrFromContext 5-74
GciExecuteStrTrav 5-76
GciFetchByte 5-79
GciFetchBytes 5-81
GciFetchChars 5-84
GciFetchClass 5-85
GciFetchDateTime 5-87
GciFetchNamedOop 5-88
GciFetchNamedOops 5-90
GciFetchNamedSize 5-92
GciFetchNameOfClass 5-93
GciFetchObjectInfo 5-95
GciFetchObjImpl 5-94
GciFetchObjInfo 5-97
GciFetchOop 5-99
GciFetchOops 5-101
GciFetchPaths 5-103
GciFetchSize 5-108
GciFetchVaryingOop 5-110
GciFetchVaryingOops 5-113
GciFetchVaryingSize 5-115
GciFindObjRep 5-117
GciFltToOop 5-119
GciGetFreeOop 5-120
GciGetFreeOops 5-122
GciGetSessionId 5-124
GciHandleError 5-125
GciHardBreak 1-27, 5-127
GciInit 1-27, 5-128
GciInitAppName 5-129
GciInstallUserAction 5-132
GciInstMethodForClass 5-130
GciInUserAction 5-133
GciIsKindOf 5-134
GciIsKindOfClass 5-135
GciIsRemote 5-136
GciIsSubclassOf 5-138
GciIsSubclassOfClass 5-139
GciIvNameToIdx 5-140
GciLnk
  GciIsRemote 5-136
  Loading 5-250, 5-251, 5-253
  object traversal function 2-2
  path access function 2-2
  use only with debugged applications 4-4
  use to enhance performance 2-2
  user action 3-13
GciLoadUserActionLibrary 5-142
GciLogin 5-144
GciLogout 5-146
GciLongToOop 5-148
GciMoreTraversal 5-152
GciNbAbort 5-155
GciNbBegin 5-156
GciNbClampedTraverseObjs 5-158
GciNbCommit 5-160
GciNbContinue 5-161
GciNbContinueWith 5-162
GciNbEnd 5-163
GciNbExecute 5-165
GciNbExecuteStr 5-167
GciNbExecuteStrFromContext 5-169
GciNbExecuteStrTrav 5-144, 5-171
GciNbMoreTraversal 5-173

GciNbPerform 5-175
GciNbPerformTrav 5-179
GciNbTraverseObjs 5-183
GciNewByteObj 5-185
GciNewCharObj 5-186
GciNewDateTime 5-187
GciNewOop 5-188
GciNewOops 5-189
GciNewOopUsingObjRep 5-191
GciNewString 5-194
GciNewSymbol 5-195
GciObjExists 5-196
GciObjInCollection 5-197
GciObjInfoSType 5-13
GciObjRepHdrSType 5-15
GciObjRepSize 5-198
GciObjRepSType 5-14
GciOopToBool 5-203
GciOopToChr 5-205
GciOopToFlt 5-207
GciOopToLong 5-209
GciPathToStr 5-213
GciPerform 5-216
GciPerformNoDebug 5-218
GciPerformSym 5-220
GciPerformTrav 5-222
GciPollForSignal 5-227
GciPopErrJump 5-229
GciProcessDeferredUpdates 5-231
GciPushErrHandler 5-232
GciPushErrJump 5-233
GciRaiseException 5-237
GciReleaseAllOops 5-238
GciReleaseOops 5-239
GciRemoveOopFromNsc 5-241
GciRemoveOopsFromNsc 5-243
GciReplaceOops 5-245
GciReplaceVaryingOops 5-247
GciResolveSymbol 5-248
GciResolveSymbolObj 5-249
GciRpc
    loading 5-250, 5-251, 5-253
    multiple GemStone sessions 2-2
    object traversal function 2-2
    path access function 2-2
    use in debugging your application 2-2
GciRtlIsLoaded 5-250
GciRtlLoad 5-251
GciRtlUnLoad 5-253
GciSaveObjs 5-254
GciSendMsg 5-255
GciSessionIsRemote 5-257
GciSetErrJump 5-258
GciSetNet 5-260
GciSetSessionId 5-263
GciShutdown 5-264
GciSoftBreak 1-27, 5-265
GciStoreByte 5-266
GciStoreBytes 5-268
GciStoreBytesInstanceOf 5-270
GciStoreChars 5-272
GciStoreIdxOop 5-274
GciStoreIdxOops 5-276
GciStoreNamedOop 5-278
GciStoreNamedOops 5-280
GciStoreOop 5-282
GciStoreOops 5-284
GciStorePaths 5-287
GciStoreTrav 5-292
GciStrKeyValueDictAtObj 5-297
GciStrKeyValueDictAtObjPut 5-298
GciStrKeyValueDictAtPut 5-299
GciStrToPath 5-300
GciSymDictAt 5-303
GciSymDictAtObj 5-304
GciSymDictAtObjPut 5-305
GciSymDictAtPut 5-306
GciTraverseObjs 5-307
GciUnsignedLongToOop 5-313
GCIUSER_ACTION_INIT_DEF 3-4
GCIUSER_ACTION_SHUTDOWN_DEF 3-5
GciUserActionInit 3-4, 5-314

GciUserActionShutdown 3-4, 3-5
GciVersion 5-317
GemBuilder
  libraries 2-2, 4-5
  library file
    gcilnk50.* 2-3
    gcirpc50.* 2-3
  loading 5-250, 5-251, 5-253
  run-time binding 2-3, 4-4
GemRpc, user action 3-14, C-2
GemStone service name 5-260
GemStone-defined object, making available to applications 1-8
getting, see fetching, see finding

H

handling errors 5-229, 5-233, 5-258
hard break 5-20, 5-127, 5-155
  defined 1-14
host
  file access, default directory 1-28
  password 5-260
  username 5-260
host-specific C definition 5-10

I

implementation of an object
  fetching 1-17, 5-94
  object report 5-40, 5-117, 5-158
implementing a user action 3-3
importing objects
  from GemStone 1-3, 1-16
improving application performance 1-22, 1-25, 1-32, 5-103, 5-152, 5-173, 5-181, 5-183, 5-238, 5-239, 5-287, 5-292, 5-307
include file (GemBuilder)
  gci.ht 5-10
  gcioop.ht A-1
include file (GemBuilder)
  flag.ht 5-10
  gci.ht 1-8
  gcicmn.ht 5-10
  gcierr.ht 5-10
  gcioc.ht 5-10
  gcioop.ht 1-8, 1-10, 5-10
  gcirtl.hf 5-10
  gcirtl.ht 5-11
  gcirtlm.hf 5-10
  gciuser.hf 5-11
  staticua.hf 5-11
  version.ht 5-11
incomplete
  object report 5-310
indexable instance variable, fetching the value of an object's 5-113
initializing GemBuilder 1-5, 5-128
initiating a GemStone session 5-142
installing a user action 5-132
instance
  GemStone-defined A-1
  method, compiling 1-3, 5-130
  variable 1-18
    enumerating for a class 5-44, 5-140
integer, converting to an object 5-148, 5-150, 5-313
interrupt
  GemStone (hard break) 1-14
  handling 1-14, 1-27
  issuing 1-14, 5-20, 5-127, 5-155, 5-265
  virtual machine (soft break) 1-14, 5-49, 5-161, 5-265
invariance, object vs. class 5-16

J

jump buffer, error handling in GemBuilder 1-29, 5-229, 5-258
jump buffer, error handling in the GemBuilder 5-233

K

kernel class 5-188, 5-189
  mnemonics 1-8, A-2

L

level traversal 1-24, 5-152, 5-173, 5-183, 5-307
library
  GemBuilder 2-2, 4-5
  name conflicts C-4
  run-time loading 3-7
  search 2-4
  static C-1
  user action 3-4
linkable GemBuilder (GciLnk)
  GciIsRemote 5-136
  use to enhance performance 2-2
linking
  applications 1-2
  applications and user actions 3-12, 3-13
loading
  user action 3-6
  GemBuilder 5-250, 5-251, 5-253
logging in to GemStone 1-6, 5-142, 5-260
logging out from GemStone 1-6, 5-146
logical access to objects 1-3, 1-12
login parameter 5-142
longjmp, setjmp 1-29, 5-233, 5-258

M

macros defined 5-10
message
  GemBuilder function 1-13
  sending 1-12, 5-175, 5-216, 5-255
method
  calling C functions from 5-132
  compiling 1-3, 5-42, 5-130
mnemonic
  GemStone error 1-28, 5-10
modifying
  objects directly in C 1-3, 1-16
    caution 1-16
multiple GemStone sessions 3-14
  GciRpc 2-2
  switching among 5-124, 5-263
multiple objects
  defining 5-191
  exporting 5-117, 5-152, 5-173, 5-181, 5-183, 5-287, 5-292, 5-307
  importing 5-103, 5-117, 5-152, 5-173, 5-181, 5-183, 5-292, 5-307

N

named instance variable
  fetching 5-88, 5-90
  number of 5-44, 5-92, 5-140
  pointer object 1-18
network 5-260
  minimizing traffic 1-22, 1-25, 5-103, 5-152, 5-173, 5-181, 5-183, 5-287, 5-292, 5-307
  node 5-260
  parameter 5-142, 5-260
  resource string syntax B-1
  traffic, minimizing 1-25
nil, GemStone special object 1-8, 1-9, 1-10, A-1
node name, network 5-260
nonblocking functions 1-25
non-sequenceable collection (NSC) 1-20
  adding OOPs to 1-20, 5-22, 5-23
  fetching OOPs from 1-20, 5-99, 5-101
  fetching the size 5-92, 5-108
  implementation type 1-17, 1-20, 5-94
  removing OOPs from 1-20, 5-241, 5-243
NRS (network resource string)
  syntax B-1
number of an object's instance variables
  object report 5-40, 5-117, 5-158
number of named instance variables in a class 5-44

number, converting to an object 5-119, 5-148, 5-150, 5-313
numeric representation of a path 1-25, 5-103, 5-213, 5-287, 5-300

O

object
- byte implementation type 1-17
- control function 1-32, 5-238, 5-239, 5-254
- converting to
  - boolean 5-203, 5-204
  - character 5-205, 5-206
  - floating-point number 5-207
  - integer 5-209, 5-211
- creating 1-21, 5-191
- identity 1-8
- importing or exporting multiple 1-22
- mnemonic 1-8
- NSC implementation type 1-20
- pointer implementation type 1-18
- releasing 1-31, 5-146, 5-238, 5-239
- report 5-40, 5-117, 5-152, 5-158, 5-173, 5-181, 5-183, 5-191, 5-292, 5-307
  - finding in a traversal buffer 5-117
  - incomplete 5-310
  - size 5-198
  - special objects 5-309
  - structure summary 1-24
  - traversal buffer 1-23
  - word alignment 5-27
- representation in C 1-3, 1-16
- saving 1-31
- sending messages 5-175, 5-216, 5-255

object traversal, see traversal
obtaining, see fetching, see finding
OOP (object-oriented pointer)
- adding to an NSC 1-20, 5-22, 5-23
- defined 1-8
- fetching from an NSC 1-20, 5-99, 5-101, 5-110, 5-113
- removing from an NSC 1-20, 5-241, 5-243

operating system considerations 1-27

P

password
- GemStone 5-142, 5-260
- host 5-260

path access
- function, GciLnk 2-2
- function, GciRpc 2-2
- to objects 1-25, 5-103, 5-213, 5-287, 5-300

pause message 5-49, 5-51, 5-161, 5-162
performance, improving application 1-22, 1-25, 1-32, 5-103, 5-152, 5-173, 5-181, 5-183, 5-238, 5-239, 5-287, 5-292, 5-307
pointer object
- fetching OOPs from 1-19, 5-88, 5-90, 5-110, 5-113
- fetching the size 5-92, 5-108
- implementation type 1-17, 1-18, 5-94
- storing OOPs in 1-19, 5-274, 5-276, 5-278, 5-280, 5-282, 5-284

polling for GemBuilder errors 1-29, 5-66
primitive, user-defined 5-132
private method, compilation restrictions 5-42, 5-130

R

read set 5-24
- transaction 5-24

reclaiming storage 5-238, 5-239
releasing objects 1-31, 5-146, 5-238, 5-239
remote procedure call GemBuilder (GciRpc)
- GciIsRemote 5-136
- use in debugging your application 2-2

removing OOPs from an NSC 1-20, 5-241, 5-243
report, of an object 5-40, 5-117, 5-152, 5-158, 5-173, 5-181, 5-183, 5-191, 5-198, 5-292, 5-307
re-reading objects from the database 1-6
reserved OOP 1-8

resolving symbols 1-14, 5-42, 5-68, 5-72, 5-74, 5-76, 5-130, 5-165, 5-167, 5-169, 5-171
run-time binding
  GemBuilder 2-3, 4-4
run-time loading 3-7

S

saving objects 1-31
  export set 1-31
schema 1-2
segment of an object
  object report 5-117
sending messages to GemStone objects 1-3, 1-12, 5-175, 5-216, 5-255
service name, GemStone 5-260
session
  control 1-5, 5-142, 5-146, 5-260
  creating (logging in) 1-6, 5-142, 5-260
  current 1-6
  defined 1-5
  finding the current ID number 5-124
  setting the current ID number 5-263
  switching among multiple 5-124, 5-263
  terminating (logging out) 1-6, 5-146
session user action 3-10
session user actions 3-2
setjmp, longjmp 1-29, 5-233, 5-258
shared libraries
  GemBuilder 2-2
  user action 3-1
SIGIO 1-27
signal (system function) 1-27
signal errors 5-227
size of an object
  fetching 5-92, 5-108
  object report 5-117
size of an object report, calculating 5-198
SmallInteger
  represented as a special object 1-9
soft break 5-49, 5-161, 5-265
  defined 1-14
special object
  implementation type 1-17, 5-94
  object report 5-309
  traversal of 5-309
stack
  clearing the call 5-45
starting GemBuilder 1-5, 5-128
static user actions C-1
stopping GemBuilder 1-5, 5-264
storing
  bytes in a byte object 1-18, 5-266, 5-268, 5-270
  objects by using paths 1-25, 5-287
  OOPs in a pointer object 1-19, 5-274, 5-276, 5-278, 5-280, 5-282, 5-284
string
  as a byte object 1-17
  fetching 1-17, 5-79, 5-81, 5-84
  representation of a path 1-25, 5-213, 5-300
  storing 1-18, 5-266, 5-268, 5-272
structural access 1-16, 5-22, 5-23, 5-44, 5-79, 5-81, 5-85, 5-88, 5-90, 5-92, 5-94, 5-99, 5-101, 5-108, 5-110, 5-113, 5-140, 5-188, 5-189, 5-191, 5-241, 5-243, 5-266, 5-268, 5-270, 5-274, 5-276, 5-278, 5-280, 5-282, 5-284
  function 5-18
    caution when using 5-18
switching among multiple GemStone sessions 5-124, 5-263
symbol
  as a byte object 1-17
  resolution 1-14, 5-42, 5-68, 5-70, 5-72, 5-74, 5-76, 5-130, 5-165, 5-167, 5-169, 5-171

T

terminating GemStone sessions 5-146
testing an application
  use GciRpc 4-4
threshold, see traversal, threshold

tracing a GemBuilder call while debugging 5-59
transaction
- aborting 1-7, 5-20, 5-155
- committing 1-6, 5-48, 5-160
- control 5-20, 5-48, 5-155, 5-160
- management 5-25, 5-34
- workspace, creating 5-142, 5-260
- workspace, terminating 5-146

traversal 1-22, 5-37, 5-40, 5-117, 5-152, 5-158, 5-171, 5-173, 5-179, 5-181, 5-183, 5-222, 5-224, 5-292, 5-307
- buffer 1-23, 5-152, 5-173, 5-181, 5-183, 5-292, 5-307
  - finding object reports 1-24, 5-117
  - word alignment 5-27
- clamped object 5-137
- function
  - GciLnk 2-2
  - GciRpc 2-2
- inability to continue 5-153, 5-310
- level 1-24, 5-152, 5-173, 5-183, 5-307
- special object 5-309
- threshold 5-40, 5-152, 5-158, 5-173, 5-181, 5-183, 5-292, 5-307
- value buffer 5-315
- word alignment 5-27

true, GemStone special object 1-8, 1-9, 1-10, 5-32, A-1

U

uncommitted object, releasing 1-31, 5-146, 5-238, 5-239
underscore character, private method 5-42, 5-130
unnamed instance variable, fetching 5-110, 5-113
updating the C representation of database objects 5-25, 5-34

user action
- application 3-2, 3-10
- calling from GemStone 3-9
- configurations 3-10
- debugging 3-10
- defined 3-1
- executing 3-10
- implementing 3-3
- include file 5-11
- installation macro defined 5-11
- installation verified 3-9
- installing 5-132
- library 3-4
- linked application 3-13
- loading 3-6
- name conflicts C-4
- RPC application 3-12
- run-time loading 3-7
- session 3-2, 3-10
- static C-1
- userAction instance method 3-9

user action libraries 3-1
user name
- GemStone 5-142, 5-260
- host 5-260

user profile, searching the symbol list in 5-42, 5-68, 5-72, 5-74, 5-130, 5-165, 5-167, 5-169
user session
- creating 5-142, 5-260
- terminating 5-146

user, searching the symbol list for 1-14

V

value buffer
- object report 5-117, 5-152, 5-173, 5-181, 5-183, 5-191, 5-292, 5-307
- traversal 5-315
- word alignment 5-27

value of an instance variable, object report 5-40, 5-117, 5-158

version
GemBuilder 5-317
virtual machine
call stack 1-30
clearing 1-30, 5-45
control function 5-49, 5-51, 5-68, 5-72, 5-74, 5-161, 5-162, 5-165, 5-167, 5-169, 5-175, 5-216, 5-255

W

word alignment 5-27, 5-192