INTTELLECTUAL PROPERTY OWNERSHIP
This documentation is furnished for informational use only and is subject to change without notice. VMware, Inc., assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation. This documentation, 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 express written authorization from VMware, Inc.
Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.
The software installed in accordance with this documentation is copyrighted and licensed by VMware, 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.
Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of VMware, Inc.
This software is provided by VMware, Inc. and contributors “as is” and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall VMware, Inc. or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.
COPYRIGHTS
This software product, its documentation, and its user interface © 1986-2011 VMware, Inc., and GemStone Systems, Inc. All rights reserved by VMware, Inc.
PATENTS
GemStone software is covered by U.S. Patent Number 6,256,637 “Transactional virtual machine architecture”, Patent Number 6,360,219 “Object queues with concurrent updating”, Patent Number 6,567,905 “Generational garbage collector with persistent object cache”, and Patent Number 6,681,226 “Selective pessimistic locking for a concurrently updateable database”. GemStone software may also be covered by one or more pending United States patent applications.
TRADEMARKS
VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Sun, Sun Microsystems, and Solaris are trademarks or registered trademarks of Oracle and/or its affiliates. SPARC is a registered trademark of SPARC International, Inc.
HP, HP Integrity, and HP-UX are registered trademarks of Hewlett Packard Company.
Intel, Pentium, and Itanium are registered trademarks of Intel Corporation in the United States and other countries.
Microsoft, MS, Windows, Windows XP, Windows 2003, Windows 7 and Windows Vista are registered trademarks of Microsoft Corporation in the United States and other countries.
Linux is a registered trademark of Linus Torvalds and others.
Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.
SUSE is a registered trademark of Novell, Inc. in the United States and other countries.
AIX, POWER, and POWER6 are trademarks or registered trademarks of International Business Machines Corporation.
Apple, Mac, Mac OS, Macintosh, and Snow Leopard are trademarks of Apple Inc., in the United States and other countries. Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. VMware cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.
VMware, Inc.
15220 NW Greenbrier Parkway
Suite 150
Beaverton, OR 97006
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 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.

Prerequisites

This manual assumes you are familiar with the GemStone Smalltalk programming language, as described in the Programming Guide for GemStone/S 64 Bit. 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.

You should have the GemStone system installed correctly on your host computer, as described in the GemStone/S 64 Bit Installation Guide for your platform.
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.
- 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, “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 Files and Data Structures,” describes GemBuilder include files and the data structures used internally.
- Chapter 6, “GemBuilder C Functions,” provides a detailed description of each GemBuilder function, including syntax, parameters, return value, a general description of what the function does, and including examples of its use.
- Appendix A, “Reserved OOPs,” lists mnemonics for reserved OOPs.

Terminology Conventions

The term “GemStone” is used to refer to the server products GemStone/S 64 Bit and GemStone/S; the GemStone Smalltalk programming language; and may also be used to refer to the company, previously GemStone Systems, Inc., now a division of VMware, Inc.
Other GemStone Documentation

You may find it useful to look at other GemStone documentation:

- **Programming Guide** — a programmer’s guide to GemStone Smalltalk, GemStone’s object-oriented programming language.
- **Topaz Programming Environment** — describes Topaz, a scriptable command-line interface to GemStone Smalltalk.
- **System Administration Guide** — describes maintenance and administration of your GemStone/S system.

A description of the behavior of each GemStone kernel class is available in the class comments in the GemStone Smalltalk repository. Method comments include a description of the behavior of methods.

Technical Support

GemStone Website

http://support.gemstone.com

GemStone’s Technical Support website provides a variety of resources to help you use GemStone products:

- **Documentation** for released versions of all GemStone products, in PDF form.
- **Downloads** and **Patches**, including past and current versions of GemBuilder for Smalltalk.
- **Bugnotes**, identifying performance issues or error conditions you should be aware of.
- **TechTips**, providing information and instructions that are not otherwise included in the documentation.
- **Compatibility matrices**, listing supported platforms for GemStone product versions.

This material is updated regularly; we recommend checking this site on a regular basis.
Help Requests

You may need to contact Technical Support directly, if your questions are not answered in the documentation or by other material on the Technical Support site. Technical Support is available to customers with current support contracts.

Requests for technical support may be submitted online or by telephone. We recommend you use telephone contact only for serious requests that require immediate attention, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: http://techsupport.gemstone.com
Email: techsupport@gemstone.com
Telephone: (800) 243-4772 or (503) 533-3503

When submitting a request, please include the following information:

- Your name, company name, and GemStone server license number.
- The versions of all related GemStone products, and of any other related products, such as client Smalltalk products.
- The operating system and version you are using.
- A description of the problem or request.
- Exact error message(s) received, if any, including log files if appropriate.

Technical Support is available from 8am to 5pm Pacific Time, Monday through Friday, excluding VMware/GemStone holidays.

24x7 Emergency Technical Support

GemStone Technical Support offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 24 hours a day, 7 days a week, 365 days a year, for issues impacting a production system. For more details, contact your GemStone account manager.

Training and Consulting

Consulting is available to help you succeed with GemStone products. Training for GemStone software is available at your location, and training courses are offered periodically at our offices in Beaverton, Oregon. Contact your GemStone account representative for more details or to obtain consulting services.
Chapter 1. Introduction

1.1 GemBuilder Application Overview .............................. 19
   Deciding Where to Do the Work .................................. 21
   Representing GemStone Objects in C ................................. 21
   Smalltalk Access to Objects ....................................... 22
   Calling C Functions from Smalltalk Methods ..................... 22
   The GemBuilder Functions ......................................... 23

1.2 Session Control ................................................. 23
   Starting and Stopping GemBuilder ................................. 23
   Remote Login Setup ............................................... 23
   Logging In and Out ................................................ 24
   Transaction Management ........................................... 25
     Committing a Transaction ....................................... 25
     Aborting a Transaction ......................................... 25
     Controlling Transactions Manually .............................. 25

1.3 Representing Objects in C ..................................... 26
   GemStone-Defined Object Mnemonics ............................... 27
   Converting Between Special Objects and C Values ................ 27
   Byte-Swizzling of Binary Floating-Point Values ................. 29
Table of Contents

1.4 Manipulating Objects in GemStone ........................................ 30
   Sending Messages to GemStone Objects .................................. 30
   Executing Code in GemStone ............................................... 31
   Interrupting GemStone Execution ....................................... 32
   Modification of Classes .................................................... 33

1.5 Manipulating Objects Through Structural Access .................... 34
   Direct Access to Metadata ............................................... 35
   Byte Objects ....................................................................... 35
   Pointer Objects .................................................................... 36
   Nonsequenceable Collections (NSC Objects) ............................ 38

1.6 Creating Objects ............................................................... 40

1.7 Fetching and Storing Objects .............................................. 41
   Efficient Fetching and Storing with Object Traversal .............. 41
   How Object Traversal Works .............................................. 42
   The Object Traversal Functions ......................................... 43
   Efficient Fetching And Storing with Path Access .................... 44

1.8 Nonblocking Functions ....................................................... 44

1.9 Operating System Considerations ........................................ 46
   Signal Handling in Your GemBuilder Application .................. 46
   Executing Host File Access Methods .................................... 47
   Writing Portable Code ......................................................... 47

1.10 Error Handling and Recovery ............................................. 48
   Polling for Errors ................................................................ 48
   Error Jump Buffers ............................................................. 48
   The Call Stack .................................................................... 49
   GemStone System Errors ..................................................... 49

1.11 Garbage Collection ............................................................ 49

1.12 Preparing to Execute GemStone Applications ....................... 51
   GemStone Environment Variables ........................................ 51

Chapter 2. Building Applications with
GemBuilder for C ................................................................. 53

2.1 GciRpc and GciLnk .............................................................. 53
   Use GciRpc for Debugging .................................................... 54
   Use GciLnk for Performance ................................................. 54
   Multiple GemStone Sessions ............................................... 54

VMware, Inc.  June 2011
## Chapter 3. Writing C Functions

*To Be Called from GemStone*

<table>
<thead>
<tr>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Shared User Action Libraries</td>
<td>57</td>
</tr>
<tr>
<td>3.2 How User Actions Work</td>
<td>58</td>
</tr>
<tr>
<td>3.3 Developing User Actions.</td>
<td>59</td>
</tr>
<tr>
<td>Write the User Action Functions</td>
<td>59</td>
</tr>
<tr>
<td>Create a User Action Library</td>
<td>60</td>
</tr>
<tr>
<td>The gciua.hf Header File</td>
<td>60</td>
</tr>
<tr>
<td>The Initialization and Shutdown Functions</td>
<td>61</td>
</tr>
<tr>
<td>Compiling and Linking Shared Libraries</td>
<td>63</td>
</tr>
<tr>
<td>Using Existing User Actions in a User Action Library</td>
<td>63</td>
</tr>
<tr>
<td>Using Third-party C Code with a User Action Library</td>
<td>63</td>
</tr>
<tr>
<td>Loading User Actions</td>
<td>63</td>
</tr>
<tr>
<td>Loading User Action Libraries At Run Time</td>
<td>63</td>
</tr>
<tr>
<td>Specifying the User Action Library</td>
<td>64</td>
</tr>
<tr>
<td>Creating User Actions in Your C Application</td>
<td>65</td>
</tr>
<tr>
<td>Verify That Required User Actions Have Been Installed</td>
<td>65</td>
</tr>
<tr>
<td>Write the Code That Calls Your User Actions</td>
<td>66</td>
</tr>
<tr>
<td>Remote User Actions</td>
<td>66</td>
</tr>
<tr>
<td>Limit on Circular Calls Among User Actions and Smalltalk</td>
<td>66</td>
</tr>
<tr>
<td>Debug the User Action</td>
<td>67</td>
</tr>
<tr>
<td>3.4 Executing User Actions</td>
<td>67</td>
</tr>
<tr>
<td>Choosing Between Session and Application User Actions</td>
<td>67</td>
</tr>
<tr>
<td>Running User Actions with Applications</td>
<td>69</td>
</tr>
<tr>
<td>With an RPC Application</td>
<td>69</td>
</tr>
<tr>
<td>With a Linked Application</td>
<td>70</td>
</tr>
<tr>
<td>Running User Actions with Gems</td>
<td>71</td>
</tr>
<tr>
<td>Running User Actions with Applications and Gems</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 4. Compiling and Linking  

4.1 Development Environment and Standard Libraries 76  
4.2 Compiling C Source Code for GemStone 76  
   The C++ Compiler 76  
   Listing the Version of Your Compiler 77  
   Compilation Options 78  
   Compilation Command Lines 78  
4.3 Linking C/C++ Object Code with GemStone 81  
   Risk of Database Corruption 81  
   Linker 82  
   Link Options 82  
   Command Line Assumptions 82  
   Linking Applications That Bind to GemBuilder at Run Time 83  
   Linking User Actions into Shared Libraries 84  

Chapter 5. GemBuilder  
Files and Data Structures 87  

5.1 GemBuilder Include Files 87  
5.2 GemBuilder Data Types 89  
   The Structure for Representing the Date and Time 90  
   The Error Report Structure 90  
   The Object Information Structure 92  
   The Object Report Structure 94  
   The Object Report Header Class 95  
   The User Action Information Structure 99  
   The Traversal Buffer Type 100  
5.3 Structural Access Functions 101  
5.4 environmentId 101  
5.5 UNIX Signal Handling 102  

Chapter 6. GemBuilder  
C Functions 103  

6.1 Function Summary Tables 103
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciAbort</td>
<td>114</td>
</tr>
<tr>
<td>GciAddOopToNsc</td>
<td>115</td>
</tr>
<tr>
<td>GciAddOopsToNsc</td>
<td>117</td>
</tr>
<tr>
<td>GCI_ALIGN</td>
<td>119</td>
</tr>
<tr>
<td>GciAllocTravBuf</td>
<td>120</td>
</tr>
<tr>
<td>GciAlteredObjs</td>
<td>121</td>
</tr>
<tr>
<td>GciAppendBytes</td>
<td>124</td>
</tr>
<tr>
<td>GciAppendChars</td>
<td>125</td>
</tr>
<tr>
<td>GciAppendOops</td>
<td>126</td>
</tr>
<tr>
<td>GciBegin</td>
<td>127</td>
</tr>
<tr>
<td>GCI_BOOL_TO_OOP</td>
<td>128</td>
</tr>
<tr>
<td>GciByteArrayToPointer</td>
<td>130</td>
</tr>
<tr>
<td>GciCallInProgress</td>
<td>131</td>
</tr>
<tr>
<td>GciCheckAuth</td>
<td>132</td>
</tr>
<tr>
<td>GCI_CHR_TO_OOP</td>
<td>134</td>
</tr>
<tr>
<td>GciClampedTrav</td>
<td>135</td>
</tr>
<tr>
<td>GciClampedTraverseObjs</td>
<td>138</td>
</tr>
<tr>
<td>GciClassMethodForClass</td>
<td>141</td>
</tr>
<tr>
<td>GciClassNamedSize</td>
<td>143</td>
</tr>
<tr>
<td>GciClearStack</td>
<td>145</td>
</tr>
<tr>
<td>GciCommit</td>
<td>147</td>
</tr>
<tr>
<td>GciCompileMethod</td>
<td>149</td>
</tr>
<tr>
<td>GciCompress</td>
<td>151</td>
</tr>
<tr>
<td>GciContinue</td>
<td>154</td>
</tr>
<tr>
<td>GciContinueWith</td>
<td>156</td>
</tr>
<tr>
<td>GciCreateByteObj</td>
<td>158</td>
</tr>
<tr>
<td>GciCreateOopObj</td>
<td>160</td>
</tr>
<tr>
<td>GciCTimeToDateTime</td>
<td>162</td>
</tr>
<tr>
<td>GciDateTimeToCTime</td>
<td>163</td>
</tr>
<tr>
<td>GciDbgEstablish</td>
<td>164</td>
</tr>
<tr>
<td>GciDbgEstablishToFile</td>
<td>166</td>
</tr>
<tr>
<td>GciDbgLogString</td>
<td>167</td>
</tr>
<tr>
<td>GciDeclareAction</td>
<td>168</td>
</tr>
<tr>
<td>GciDecodeOopArray</td>
<td>170</td>
</tr>
<tr>
<td>GciDecSharedCounter</td>
<td>172</td>
</tr>
<tr>
<td>GciDirtyExportedObjs</td>
<td>174</td>
</tr>
<tr>
<td>GciDirtyObjInit</td>
<td>176</td>
</tr>
<tr>
<td>GciDirtySaveObjs</td>
<td>178</td>
</tr>
<tr>
<td>GciDirtyTrackedObjs</td>
<td>180</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Gci_doubleToSmallDouble</td>
<td>182</td>
</tr>
<tr>
<td>GciEnableFreeOopEncoding</td>
<td>183</td>
</tr>
<tr>
<td>GciEnableFullCompression</td>
<td>184</td>
</tr>
<tr>
<td>GciEnableSignaledErrors</td>
<td>185</td>
</tr>
<tr>
<td>GciEncodeOopArray</td>
<td>187</td>
</tr>
<tr>
<td>GciErr</td>
<td>189</td>
</tr>
<tr>
<td>GciExecute</td>
<td>191</td>
</tr>
<tr>
<td>GciExecuteFromContext</td>
<td>193</td>
</tr>
<tr>
<td>GciExecuteStr</td>
<td>195</td>
</tr>
<tr>
<td>GciExecuteStrFromContext</td>
<td>198</td>
</tr>
<tr>
<td>GciExecuteStrTrav</td>
<td>201</td>
</tr>
<tr>
<td>GciFetchByte</td>
<td>204</td>
</tr>
<tr>
<td>GciFetchBytes_</td>
<td>206</td>
</tr>
<tr>
<td>GciFetchChars_</td>
<td>209</td>
</tr>
<tr>
<td>GciFetchClass</td>
<td>211</td>
</tr>
<tr>
<td>GciFetchDateTime</td>
<td>213</td>
</tr>
<tr>
<td>GciFetchDynamicIv</td>
<td>214</td>
</tr>
<tr>
<td>GciFetchDynamicIvs</td>
<td>215</td>
</tr>
<tr>
<td>GciFetchNamedOop</td>
<td>216</td>
</tr>
<tr>
<td>GciFetchNamedOops</td>
<td>219</td>
</tr>
<tr>
<td>GciFetchNamedSize</td>
<td>222</td>
</tr>
<tr>
<td>GciFetchNameOfClass</td>
<td>223</td>
</tr>
<tr>
<td>GciFetchNumEncodedOops</td>
<td>224</td>
</tr>
<tr>
<td>GciFetchNumSharedCounters</td>
<td>225</td>
</tr>
<tr>
<td>GciFetchObjectInfo</td>
<td>226</td>
</tr>
<tr>
<td>GciFetchObjImpl</td>
<td>228</td>
</tr>
<tr>
<td>GciFetchObjInfo</td>
<td>229</td>
</tr>
<tr>
<td>GciFetchOop</td>
<td>231</td>
</tr>
<tr>
<td>GciFetchOops</td>
<td>234</td>
</tr>
<tr>
<td>GciFetchPaths</td>
<td>237</td>
</tr>
<tr>
<td>GciFetchSharedCounterValuesNoLock</td>
<td>244</td>
</tr>
<tr>
<td>GciFetchSize_</td>
<td>246</td>
</tr>
<tr>
<td>GciFetchVaryingOop</td>
<td>248</td>
</tr>
<tr>
<td>GciFetchVaryingOops</td>
<td>251</td>
</tr>
<tr>
<td>GciFetchVaryingSize_</td>
<td>253</td>
</tr>
<tr>
<td>GciFindObjRep</td>
<td>255</td>
</tr>
<tr>
<td>GciFloatKind</td>
<td>257</td>
</tr>
<tr>
<td>GciFltToOop</td>
<td>258</td>
</tr>
<tr>
<td>GciGetFreeOop</td>
<td>260</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>GciGetFreeOops</td>
<td>262</td>
</tr>
<tr>
<td>GciGetFreeOopsEncoded</td>
<td>264</td>
</tr>
<tr>
<td>GciGetSessionId</td>
<td>265</td>
</tr>
<tr>
<td>GciHardBreak</td>
<td>267</td>
</tr>
<tr>
<td>GciHiddenSetIncludesOop</td>
<td>268</td>
</tr>
<tr>
<td>GCI_I64_IS_SMALL_INT</td>
<td>269</td>
</tr>
<tr>
<td>GciI64ToOop</td>
<td>270</td>
</tr>
<tr>
<td>GciIncSharedCounter</td>
<td>271</td>
</tr>
<tr>
<td>GciInit</td>
<td>273</td>
</tr>
<tr>
<td>GciInitAppName</td>
<td>274</td>
</tr>
<tr>
<td>GciInitAppName_</td>
<td>275</td>
</tr>
<tr>
<td>GciInstallUserAction</td>
<td>276</td>
</tr>
<tr>
<td>GciInstMethodForClass</td>
<td>277</td>
</tr>
<tr>
<td>GciInUserAction</td>
<td>279</td>
</tr>
<tr>
<td>GciIsKindOf</td>
<td>280</td>
</tr>
<tr>
<td>GciIsKindOfClass</td>
<td>281</td>
</tr>
<tr>
<td>GciIsRemote</td>
<td>282</td>
</tr>
<tr>
<td>GciIsSubclassOf</td>
<td>283</td>
</tr>
<tr>
<td>GciIsSubclassOfClass</td>
<td>284</td>
</tr>
<tr>
<td>GciIvNameToldx</td>
<td>285</td>
</tr>
<tr>
<td>GciLoadUserActionLibrary</td>
<td>287</td>
</tr>
<tr>
<td>GciLogin</td>
<td>289</td>
</tr>
<tr>
<td>GciLogout</td>
<td>291</td>
</tr>
<tr>
<td>GciLongImp</td>
<td>292</td>
</tr>
<tr>
<td>GciMoreTraversal</td>
<td>293</td>
</tr>
<tr>
<td>GciNbAbort</td>
<td>296</td>
</tr>
<tr>
<td>GciNbBegin</td>
<td>297</td>
</tr>
<tr>
<td>GciNbClampedTrav</td>
<td>298</td>
</tr>
<tr>
<td>GciNbClampedTraverseObjs</td>
<td>299</td>
</tr>
<tr>
<td>GciNbCommit</td>
<td>301</td>
</tr>
<tr>
<td>GciNbContinue</td>
<td>302</td>
</tr>
<tr>
<td>GciNbContinueWith</td>
<td>303</td>
</tr>
<tr>
<td>GciNbEnd</td>
<td>304</td>
</tr>
<tr>
<td>GciNbExecute</td>
<td>306</td>
</tr>
<tr>
<td>GciNbExecuteStr</td>
<td>308</td>
</tr>
<tr>
<td>GciNbExecuteStrFromContext</td>
<td>310</td>
</tr>
<tr>
<td>GciNbExecuteStrTrav</td>
<td>312</td>
</tr>
<tr>
<td>GciNbMoreTraversal</td>
<td>314</td>
</tr>
<tr>
<td>GciNbPerform</td>
<td>315</td>
</tr>
</tbody>
</table>
GciNbPerformNoDebug .......................... 317
GciNbPerformTrav .............................. 319
GciNbStoreTrav ................................. 321
GciNbStoreTravDo ............................... 323
GciNbStoreTravDoTrav_ ......................... 324
GciNbStoreTravDoTravRefs_ ..................... 326
GciNbTraverseObjs .............................. 328
GciNewByteObj ................................. 330
GciNewCharObj ................................ 331
GciNewDateTime ................................ 332
GciNewOop ..................................... 333
GciNewOops .................................... 335
GciNewOopUsingObjRep ......................... 338
GciNewString .................................. 341
GciNewSymbol .................................. 342
GciNscIncludesOop .............................. 343
GciObjExists .................................... 345
GciObjIsCommitted .............................. 347
GciObjInCollection ............................. 346
GciObjIsCommitted .............................. 347
GciObjRepSize_ ................................ 348
GciOldOopToNewOop ............................. 350
GCI_OOP_IS_BOOL ............................... 351
GCI_OOP_IS_SMALL_INT ......................... 352
GCI_OOP_IS_SPECIAL ............................ 353
GciOopToBool .................................. 354
GCI_OOP_TO_BOOL ............................... 356
GciOopToChar16 ................................ 357
GciOopToChar32 ................................ 358
GciOopToChr .................................... 359
GCI_OOP_TO_CHR ................................. 361
GciOopToFlt .................................... 362
GciOopToI32 ................................... 364
GciOopToI32_ .................................. 365
GciPathToStr .................................... 368
GciPerform ..................................... 369
GciPerformNoDebug ............................. 373
GciPerformSymDbg .............................. 375
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciPerformTrav</td>
<td>377</td>
</tr>
<tr>
<td>GciPerformTraverse</td>
<td>379</td>
</tr>
<tr>
<td>GciPointerToByteArray</td>
<td>383</td>
</tr>
<tr>
<td>GciPollForSignal</td>
<td>384</td>
</tr>
<tr>
<td>GciPopErrJump</td>
<td>386</td>
</tr>
<tr>
<td>GciProcessDeferredUpdates</td>
<td>388</td>
</tr>
<tr>
<td>GciProduct</td>
<td>390</td>
</tr>
<tr>
<td>GciPushErrJump</td>
<td>391</td>
</tr>
<tr>
<td>GciRaiseException</td>
<td>393</td>
</tr>
<tr>
<td>GciReadSharedCounter</td>
<td>394</td>
</tr>
<tr>
<td>GciReadSharedCounterNoLock</td>
<td>395</td>
</tr>
<tr>
<td>GciRealloc</td>
<td>397</td>
</tr>
<tr>
<td>GciReleaseAllGlobalOops</td>
<td>398</td>
</tr>
<tr>
<td>GciReleaseAllOops</td>
<td>399</td>
</tr>
<tr>
<td>GciReleaseAllTrackedOops</td>
<td>400</td>
</tr>
<tr>
<td>GciReleaseGlobalOops</td>
<td>401</td>
</tr>
<tr>
<td>GciReleaseOoops</td>
<td>402</td>
</tr>
<tr>
<td>GciReleaseTrackedOoops</td>
<td>405</td>
</tr>
<tr>
<td>GciRemoveOopFromNsc</td>
<td>406</td>
</tr>
<tr>
<td>GciRemoveOopsFromNsc</td>
<td>408</td>
</tr>
<tr>
<td>GciReplaceOops</td>
<td>410</td>
</tr>
<tr>
<td>GciReplaceVaryingOops</td>
<td>412</td>
</tr>
<tr>
<td>GciResolveSymbol</td>
<td>413</td>
</tr>
<tr>
<td>GciResolveSymbolObj</td>
<td>414</td>
</tr>
<tr>
<td>GciRtlIsLoaded</td>
<td>415</td>
</tr>
<tr>
<td>GciRtlLoad</td>
<td>416</td>
</tr>
<tr>
<td>GciRtlUnload</td>
<td>418</td>
</tr>
<tr>
<td>GciSaveAndTrackObjs</td>
<td>419</td>
</tr>
<tr>
<td>GciSaveGlobalObjs</td>
<td>421</td>
</tr>
<tr>
<td>GciSaveObjs</td>
<td>422</td>
</tr>
<tr>
<td>GciServerIsBigEndian</td>
<td>423</td>
</tr>
<tr>
<td>GciSessionIsRemote</td>
<td>424</td>
</tr>
<tr>
<td>GciSetCacheName</td>
<td>425</td>
</tr>
<tr>
<td>GciSetDynLib</td>
<td>426</td>
</tr>
<tr>
<td>GciSetErrJump</td>
<td>427</td>
</tr>
<tr>
<td>GciSetHaltOnError</td>
<td>430</td>
</tr>
<tr>
<td>Gc_SETJMP</td>
<td>431</td>
</tr>
<tr>
<td>GciSetNet</td>
<td>432</td>
</tr>
<tr>
<td>GciSetSessionId</td>
<td>435</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>GciSetSharedCounter</td>
<td>437</td>
</tr>
<tr>
<td>GciSetTraversalBufSwizzling</td>
<td>438</td>
</tr>
<tr>
<td>GciSetVaryingSize</td>
<td>439</td>
</tr>
<tr>
<td>GciShutdown</td>
<td>440</td>
</tr>
<tr>
<td>GciSoftBreak</td>
<td>441</td>
</tr>
<tr>
<td>GciStep</td>
<td>444</td>
</tr>
<tr>
<td>GciStoreByte</td>
<td>445</td>
</tr>
<tr>
<td>GciStoreBytes</td>
<td>447</td>
</tr>
<tr>
<td>GciStoreBytesInstanceOf</td>
<td>449</td>
</tr>
<tr>
<td>GciStoreChars</td>
<td>451</td>
</tr>
<tr>
<td>GciStoreDynamicIv</td>
<td>453</td>
</tr>
<tr>
<td>GciStoreIdxOop</td>
<td>454</td>
</tr>
<tr>
<td>GciStoreIdxOops</td>
<td>456</td>
</tr>
<tr>
<td>GciStoreNamedOop</td>
<td>459</td>
</tr>
<tr>
<td>GciStoreNamedOops</td>
<td>462</td>
</tr>
<tr>
<td>GciStoreOop</td>
<td>465</td>
</tr>
<tr>
<td>GciStoreOoops</td>
<td>468</td>
</tr>
<tr>
<td>GciStorePaths</td>
<td>471</td>
</tr>
<tr>
<td>GciStoreTrav</td>
<td>478</td>
</tr>
<tr>
<td>GciStoreTravDo</td>
<td>482</td>
</tr>
<tr>
<td>GciStoreTravDoTrav_</td>
<td>486</td>
</tr>
<tr>
<td>GciStoreTravDoTravRefs_</td>
<td>488</td>
</tr>
<tr>
<td>GciStringToInteger</td>
<td>492</td>
</tr>
<tr>
<td>GciStrKeyValueDictAt</td>
<td>493</td>
</tr>
<tr>
<td>GciStrKeyValueDictAtObj</td>
<td>494</td>
</tr>
<tr>
<td>GciStrKeyValueDictAtObjPut</td>
<td>495</td>
</tr>
<tr>
<td>GciStrKeyValueDictAtPut</td>
<td>496</td>
</tr>
<tr>
<td>GciStrToPath</td>
<td>497</td>
</tr>
<tr>
<td>GciSwapBytesUint</td>
<td>500</td>
</tr>
<tr>
<td>GciSwapBytesUshort</td>
<td>501</td>
</tr>
<tr>
<td>GciSymDictAt</td>
<td>502</td>
</tr>
<tr>
<td>GciSymDictAtObj</td>
<td>504</td>
</tr>
<tr>
<td>GciSymDictAtObjPut</td>
<td>505</td>
</tr>
<tr>
<td>GciSymDictAtPut</td>
<td>506</td>
</tr>
<tr>
<td>GciTrackedObjsFetchAllDirty</td>
<td>507</td>
</tr>
<tr>
<td>GciTrackedObjsInit</td>
<td>509</td>
</tr>
<tr>
<td>GciTraverseObjs</td>
<td>510</td>
</tr>
<tr>
<td>GciUncompress</td>
<td>515</td>
</tr>
<tr>
<td>GciUserActionInit</td>
<td>517</td>
</tr>
</tbody>
</table>
Appendix A. Reserved OOPs

Appendix B. GemStone C Statistics Interface

B.1 Developing a GCSI Application
   Required Header Files
   The GCSI Shared Library
   Compiling and Linking
   Connecting to the Shared Page Cache
   The Sample Program

B.2 GCSI Data Types
   The Structure for Representing the GCSI Function Result
   GcsiAllStatsForMask
   GcsiAttachSharedCache
   GcsiAttachSharedCacheForStone
   GcsiDetachSharedCache
   GcsiFetchMaxProcessesInCache
   GcsiInit
   GcsiShrPcMonStatAtOffset
   GcsiStnStatAtOffset
   GcsiStatsForGemSessionId
   GcsiStatsForGemSessionWithName
   GcsiStatsForPgsvrSessionId
   GcsiStatsForProcessId
   GcsiStatsForShrPcMon
   GcsiStatsForStone
   GCSI Errors

Index
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.
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, “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. By import, we mean that memory is allocated within your C program to contain the C equivalent of the GemStone Smalltalk object. You could also say that these values are cached in your application; rather than having a reference to the object by identity (OOP), we have the contents of its instance variables. The object in its permanent form still exists in the repository, and the cached values in your application may become obsolete if other sessions commit changes to this object. Exporting is the reverse of importing - you create a GemStone Smalltalk object that holds the equivalent to your C data, or update an existing GemStone Smalltalk object with the C data in your application.

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 34 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 30 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 two 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. 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 System Administration Guide for GemStone/S 64 Bit 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.

```
GciInit must be called before the first GciLogin in the lifetime of a process.
```

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:

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

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

```c
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 “GciAbort” on page 114.
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)
- 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, Integer, or Boolean objects and the equivalent C values:

- `GCI_BOOL_TO_OOP` — (MACRO) Convert a C Boolean value to a GemStone Boolean object.
- `GciByteArrayToPointer` — Given a result from `GciPointerToByteArray`, return a C pointer.
- `GCI_CHR_TO_OOP` — (MACRO) Convert a C character value to a GemStone Character object.
- `Gcil64ToOop` — Convert a C 64-bit integer value to a GemStone object.
- `GciOopToBool` — Convert a Boolean object to a C Boolean value.
GCI_OOP_TO_BOOL — (MACRO) Convert a Boolean object to a C Boolean value.

GciOopToChar16 — Convert a Character object to a 16-bit C character 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.

GciOopToI32, GciOopToI32_ — Convert a GemStone object to a C 32-bit integer value.

GciOopToI64, GciOopToI64_ — Convert a GemStone object to a C 64-bit integer value.

GciPointerToArray — Given a C pointer, return a SmallInteger or ByteArray containing the value of the pointer.

GciStringToInteger — Convert a C string to a GemStone SmallInteger, LargePositiveInteger or LargeNegativeInteger object.

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 SmallDouble or Float object.

GciOopToFlt — Convert a SmallDouble, Float, or SmallFloat object to a C double.

The following macros are for testing OOPs:

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.h 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 \textit{address}. Finally, assume that the fifth instance variable represents the zip code of the address.

The fifth element of \textit{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 \textit{zip}.

\textit{This example assumes that you already have a valid session (obtained from the successful execution of GciLogin).}

**Example 1.1**

```c
int64 example1_1(OopType addressId)
{
    // returns the zipcode or -1 if an error occurred,
    enum { addr_num_instVars = 5 };

    OopType instVars[addr_num_instVars];

    int numRet = GciFetchOops(addressId, 1, instVars, addr_num_instVars);
    if (numRet != (int)addr_num_instVars)
        return -1;

    BoolType conversionError = FALSE;
    int64 zip = GciOopToI64_(instVars[4], &conversionError);
    if (! conversionError)
        return -1;

    // zip now contains an integer that has the same value as the GemStone object represented by address[4]

    return zip;
}
```

**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 non-special floating-point objects when they are
transferred between such machines, the bytes need to be reordered (swizzled) to match the machine to which they are transferred.

In GemStone, a binary float is an instance of class Float (eight bytes) or SmallFloat (four bytes), or an instance of SmallDouble (a special object identifier that has no body). Instances of Float and SmallFloat have byte-format bodies whose size is fixed by GemStone and cannot be changed. The programmer must supply all the bytes, or provide a C double, for a binary floating object when creating or storing it.

Most GemBuilder functions provide automatic byte swizzling for instances of Float and SmallFloat. The following GemBuilder functions raise an error if you pass a Float or SmallFloat object to them:

- **GciAppendBytes** — Append bytes to a byte object. (page 124)
- **GciStoreByte** — Store one byte in a byte object. (page 445)
- **GciStoreBytes** — (MACRO) Store multiple bytes in a byte object. (page 447)
- **GciStoreChars** — Store multiple ASCII characters in a byte object. (page 451)

The **GciFetchBytes** function does not raise an error if you pass an instance of Float or SmallFloat 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 Floats or SmallFloats, 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 the function **GciPerform**, which sends 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 this function.

*This example assumes that you already have a valid session (obtained from the successful execution of GciLogin).*
Example 1.2

```c
void example_1_2(void)
{
    OopType userGlobals = GciResolveSymbol("UserGlobals", OOP_NIL);
    OopType aKey = GciNewSymbol("myNumber");
    OopType aValue = GciI32ToOop(55);

    OopType argList[2];
    argList[0] = aKey;
    argList[1] = aValue;

    /* Two statements that have the same effect when executed */
    OopType result = GciSendMsg(userGlobals, 4, "at:", aKey, "put:", aValue);

    result = GciPerform(userGlobals, "at:put:", argList, 2);
}
```

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.
  (page 191)

- **GciExecuteFromContext** — Execute a Smalltalk expression contained in a String object as if it were a message sent to another object. (page 193)

- **GciExecuteStr** — Execute a Smalltalk expression contained in a C string.
  (page 195)

- **GciExecuteFromContext** — Execute a Smalltalk expression contained in a C string as if it were a message sent to an object. (page 198)

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.

This example assumes that you already have a valid session (obtained from the successful execution of GciLogin).

Example 1.3

```c
OopType example_1_3(void)
{
    // Pass the String to GemStone for compilation and execution.
    // If it succeeds, return the result of the expression shown
    // otherwise OOP_NIL will be returned.

    OopType objSize = GciExecuteStr(" ^ myObject size ",
                                    OOP_NIL/*use default symbolList*/);
    return objSize;
}
```

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 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 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 a signal handler 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. 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 GciPerform 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 *Programming Guide for GemStone/S 64 Bit*. 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. (page 124)
- `GciAppendChars` — Append a C string to a byte object. (page 125)
- `GciFetchByte` — Fetch one byte from an indexed byte object. (page 204)
- `GciFetchBytes_` — Fetch multiple bytes from an indexed byte object. (page 206)
**GciFetchChars** — Fetch multiple ASCII characters from an indexed byte object. (page 209)

**GciStoreByte** — Store one byte in a byte object. (page 445)

**GciStoreBytes** — (MACRO) Store multiple bytes in a byte object. (page 447)

**GciStoreChars** — Store multiple ASCII characters in a byte object. (page 451)

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 `suppld` contains an OOP representing an object of class String. Example 1.4 imports that String into the C variable `suppName`.

>This example assumes that you already have a valid session (obtained from the successful execution of GciLogin).

**Example 1.4**

```c
void example_1_4(OopType suppId)
{
    char suppName[1025];
    int64 size = GciFetchBytes_(suppId, 1L, (ByteType*)suppName,
    sizeof(suppName) - 1);
    suppName[size] = '\0';
    // suppName now contains the bytes of the GemStone object referenced
    // by suppId , or the first 1024 bytes whichever is less
}
```

**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. (page 216)

GciFetchNamedOops — Fetch the OOPs of one or more of an object’s named instance variables. (page 219)

GciStoreNamedOop — Store one OOP into an object’s named instance variable. (page 459)

GciStoreNamedOops — Store one or more OOPs into an object’s named instance variables. (page 462)

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 indexable pointer object or NSC. (page 248).

GciFetchVaryingOops — Fetch the OOPs of one or more unnamed instance variables from an indexable pointer object or NSC. (page 251).

GciStoreIdxOop — Store one OOP in an indexable pointer object’s unnamed instance variable. (page 454).

GciStoreIdxOops — Store one or more OOPs in an indexable pointer object’s unnamed instance variables. (page 456).
In each of the following functions, if the indexable object contains named instance variables, pointers to the named instance variables precede pointers to the indexable instance variables.

**GciFetchOop** — Fetch the OOP of one instance variable of an object. (page 231)

**GciFetchOops** — Fetch the OOPs of one or more instance variables of an object. (page 234)

**GciStoreOop** — Store one OOP into an object’s instance variable. (page 465)

**GciStoreOops** — Store one or more OOPs into an object’s instance variables. (page 468)

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`.

*This example assumes that you already have a valid session (obtained from the successful execution of GciLogin).*

**Example 1.5**

```c
void example_1_5(OopType currSup)
{
    enum { num_ivs = 7 };
    OopType instVars[num_ivs];

    int numRet = GciFetchNamedOops(currSup, 1L, instVars, num_ivs);
    if (numRet == 7) {
        // instVars now contains the OOPs of the seven instance
        // variables of the GemStone object referenced by currSup
    } else {
        // error occurred or currSup is not of expected class or size
    }
}
```

**Nonsequenceable Collections (NSC Objects)**

In addition to byte objects and pointer objects, GemStone exports objects implemented as nonsequenceable collections (NSCs). NSC objects (for example, instances of class IdentityBag and IdentitySet) 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 nonsequenceable collection. (page 115)

**GciAddOopsToNsc** — Add multiple OOPs to the unordered variables of a nonsequenceable collection. (page 117)

**GciFetchOop** — Fetch the OOP of one instance variable of an object. (page 231)

**GciFetchOops** — Fetch the OOPs of one or more instance variables of an object. (page 234)

**GciRemoveOopFromNsc** — Remove an OOP from an NSC. (page 406)

**GciRemoveOopsFromNsc** — Remove one or more OOPs from an NSC. (page 408)

**GciReplaceVaryingOops** — Replace all unnamed instance variables in an NSC object. (page 412)

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 the unnamed instance variables of an NSC. (You can use the **GciStoreOop**, **GciStoreOops**, **GciStoreNamedOop**, and **GciStoreNamedOops** functions to modify user-defined named instance variables of an NSC. You cannot, however, use these functions to modify the named instance variables defined in class IdentityBag.)

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.

This example assumes that you already have a valid session (obtained from the successful execution of **GciLogin**).
Example 1.6

```c
void example_1_6(OopType mySuppSet, OopType newSupp)
{
    GciAddOopToNsc(mySuppSet, newSupp);

    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        // an error occurred
    } else {
        // The instance of SupplierSet 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 53.

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

This example assumes that you already have a valid session (obtained from the successful execution of GciLogin).

Example 1.7

```noindent
Object subclass: 'Employee'

instVarNames: #('name' 'empNum' 'jobTitle'
               'department' 'address' 'favoriteTune')

classVars: #()

classInstVars: #()

poolDictionaries: #()

inDictionary: UserGlobals
```

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.

![Object Traversal and Paths](image)

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 [GciTraverseObj](#) traverses object trees rooted at a collection of one or more GemStone objects, gathering object reports on the specified objects into a traversal buffer.

- **Traversal buffers** are instances of the C++ class GciTravBufType, which is defined in `$GEMSTONE/include/gcicmn.ht`. (For details about GciTravBufType, see page 100.)

- **Object reports** within the traversal buffer are described by the C++ classes GciObjRepSType and GciObjRepHdrSType, which are defined in `$GEMSTONE/include/gci.ht`. (For details about these classes, see page 94.)

Each object report provides information about an object’s identity (its OOP), class, size (the number of instance variables, named plus unnamed), object security policy id, 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.

The function [GciStoreTravDo](#) is even more parsimonious of network resources. In a single network round trip, you can store values into any number of existing GemStone objects, then execute some code; the function returns a pointer to the resulting object. That function takes a structure as its argument, which defines traversal buffer of object reports and an execution string or message. After the function has completed, the structure also contains information describing the GemStone objects that have changed.
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.

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 \((\text{GciNbEnd})\) return \text{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:

\begin{verbatim}
GciCallInProgress
GciErr
GciGetSessionId
GciHardBreak
GciNbEnd
GciSetSessionId
GciShutdown
GciSoftBreak
\end{verbatim}

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 \text{GCI\_ERR\_OP\_IN\_PROGRESS} is generated.

Once a nonblocking operation is in progress, an application must call \text{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, \text{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, \text{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 \text{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.

Signal Handling in Your GemBuilder Application

Under UNIX, it is important that signals be enabled when your code calls GemBuilder functions. Disabling signals has the effect of disabling much of the error handling within GemBuilder. Because signal handlers can execute at arbitrary points during execution of your application, your signal handling code should not call any GemBuilder functions other than GciSoftBreak, GciHardBreak, or GciCallInProgress.

GciInit always installs a signal handler for SIGIO. This handler chains to any previous handler.

In the linkable (GciLnk) configuration, GciInit also does the following:

- Installs handlers to service and ignore these signals (if no previous handler is found): SIGPIPE, SIGHUP, SIGDANGER
- Installs handlers to treat the following signals as fatal errors if they are defined by the operating system: SIGTERM, SIGXCPU, SIGABRT, SIGXFSZ, SIGXCPU, SIGEMT, SIGLOST
- Installs a handler for SIGUSR1. If you have a valid linkable session, SIGUSR1 will cause the Smalltalk interpreter to print the current Smalltalk stack to stdout or to the Topaz output file. This handler chains to any previous handler.
- Installs a handler for SIGUSR2, which is used internally by a Gemstone session. This handler chains to any previous handler.
- Installs a handler to gracefully handle SIGCHLD if no previous handler is found.
- Installs a handler to treat SIGFPE as a fatal error if no previous handler is found.
- Installs handlers for SIGILL and SIGBUS. If the program counter is found to be in libgci*.so code, or if no previous handler is available to chain to, these are fatal errors.
- Installs a handler for SIGSEGV. A Smalltalk stack overflow produces a SIGSEGV, which is translated to a Smalltalk stack overflow error. If the
program counter is found to be in libgci*.so code, or if no previous handler is available to chain to, SEGV is a fatal error.

If your application installs a handler for SIGIO after calling GciInit, your handler must chain to the previously existing handler.

If your application uses Linkable GCI and installs any signal handlers after calling GciInit, you must chain to the previously existing handlers. If you install handlers for SIGSEGV, SIGILL or SIGBUS, your handler must determine if the program counter at the point of the signal is in your own C or C++ code and if not, must chain to the previously existing handler. You must only treat these signals as fatal if the program counter is in your own code.

If you are linking with other shared libraries, it is recommended that GciInit be called after all other libraries are loaded.

**Executing Host File Access Methods**

If you use GciPerform 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 class>>performOnServer: and the file accessing methods implemented in GsFile. See the file I/O information in the Programming Guide for GemStone/S 64 Bit.

**Writing Portable Code**

If you want to produce code that can run in both 32-bit and 64-bit environments, observe the following guidelines:

- Don’t hard-code size computations. Instead, use sizeof operations, so that if some structure changes, your code will still return the correct values.

- If you are using printf strings to print 64-bit integers, you might find it convenient to use the FMT_.* macros in $GEMSTONE/include/gcicmn.ht. Those macros help you to compose a format string for a printf that will be portable. In particular, use of the FMT_ macros make the printing of 64-bit integers portable between Windows and UNIX.
• To avoid discrepancies between 32-bit and 64-bit environments, avoid the use of `long` or `unsigned long` in your code. Instead, you can use the type `intptr_t`, which makes the variable the same size as a pointer, regardless of whether your application is running in 32-bit or 64-bit. Alternatively, you can use `int64` or `int` to fix the size of the variable explicitly.

1.10 Error Handling and Recovery

Your C program is responsible for processing any errors generated by GemBuilder function calls.

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, runtime (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` 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 three or more GemBuilder calls in sequence, jump buffers provide significantly faster performance than polling for errors.

When your C program calls `Gci_SETJMP`, the context of the current C environment is saved in a jump buffer designated by your program. 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 implementation calls `GciLongJmp` using the buffer currently at the top of GemBuilder’s error jump stack, and pops that buffer from the stack.

For functions with local error recovery, your program can call `GciSetErrJump` to temporarily disable the `GciLongJmp` mechanism (and to re-enable it afterwards).
Whenever the jump stack is empty, the application must use GciErr to poll for any GemBuilder errors.

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 GciLlk, 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 System Administration Guide for GemStone/S 64 Bit.

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 GciLlk again.

1.11 Garbage Collection

GemStone performs automatic garbage collection via several mechanisms, which are discussed more fully in the chapter “GemStone Garbage Collection” in the System Administration Guide for GemStone/S 64 Bit.

In-memory garbage collection of non-persistent temporary objects occurs regularly, to avoid low and out of memory issues. If newly created or temporary objects are not referenced, they run the risk of being garbage collected and disappearing prematurely during in-memory garbage collection. To avoid this
problem GemStone uses several internal sets: the PureExportSet, the GciTrackedObjs set, and the user action’s export set.

Before removing any objects, the GemStone in-memory garbage collector checks the PureExportSet and the GciTrackedObjs set in the user session’s workspace, and if in a user action, the user action’s export set. Any object in these sets is considered to be referenced. The garbage collector does not remove objects that are in these sets, 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.

Some functions will automatically add the objects which they return to the export sets. Objects may also be added and removed explicitly. Objects are automatically added to an export set in these cases:

- The results of GciNew*, GciCreate*, GciSend*, GciPerform*, GciExecute* and GciResolve* calls are automatically added to the applicable export set - either the PureExportSet, or if the function is called from within a user action, to the user action’s export set.
- Objects returned in the report buffer of a GciFetchObjectInfo or GciClampedTrav when GCI_RETRIEVE_EXPORT flag is set will be added to the PureExportSet or the user action’s export set.
- When the function
  
  GciErr(GciErrSType *errorReport);

  returns TRUE, values of type OopType in the *errorReport are added to the applicable export set.

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 applicable export set. GemBuilder does not automatically add other objects to the export sets; the application should be careful to explicitly call the GciSaveObjs or GciSaveGlobalObjs function when it needs to be sure to retain an object that is not already in an export set.

Objects that are the contents of instance variables, such as objects returned from a call to GciFetchOops, are not added to the export sets. These are already referenced from the object whose instance variable references them. Note however that these objects are cached in your C code, and the values may no longer be valid if the referencing object becomes dirty due to an abort or commit.
Persistent objects may be added to any of the three sets, in which case they are protected from garbage collection on persistent objects, such as markForCollection.

In a user action, some of these functions behave differently. When these functions are called from within a user action, the objects are added to the user action’s export set to prevent them from being garbage collected, rather than to the PureExportSet. When the user action comes to an end, the user action’s export set ceases to exist and the objects it contained may be garbage collected. This avoids the risk of objects not being released and consuming excess memory, for example if the user action exits with an unexpected error. In order to prevent objects saved from within a user action from being released prematurely, the user action can explicitly call GciSaveGlobalObjs, which will save them to the PureExportSet regardless of the user action context.

Once the objects in the GciTrackedObjs or in the export sets are no longer needed, the application can improve performance and avoid out of memory issues by calling the GciRelease... functions, to reduce the size of the set and permit garbage collection of obsolete temporaries.

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.

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 System Administration Guide for GemStone/S 64 Bit 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 System Administration Guide for GemStone/S 64 Bit for details.

GEMSTONE_NRS_ALL — A network resource string — a means for identifying certain GemStone file and process information. It can identify the name of the script to run to start an RPC Gem. See the System Administration Guide for GemStone/S 64 Bit for details.
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 81.

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. 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.

The GemBuilder library files libgcilnk.* and libgcirpc.* reside in $GEMSTONE/lib.

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 (libgcirpc.* and libgcilnk.*).

   Chapter 4, “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, Linux or AIX
   - LD_LIBRARY_PATH or SHLIB_PATH HP-UX
2. Any path recorded in the executable when it was built.
3. The global directory /usr/lib.
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`, `GciPerform`, or `GciContinue`.

2. Your GemStone Smalltalk code invokes a user action function (written in C) by sending a message of the form:
   ```c
   System userAction: aSymbol with: 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 63.)

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 class>>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 install as a user action called from Smalltalk is little different from writing other C functions. However, one important difference exists: user actions cannot reliably retain references to objects they create. The application that called the user action (whether written in C, Java, or Smalltalk) controls the export set—the set of OOPs to save after execution completes. Therefore, make sure your C application treats all argument and result objects of a user action as temporary objects. Don’t save the OOPs in static C variables for use by a subsequent invocation of the user action or by another C function.
Don’t rely on **GciSaveObjs** to make the objects persistent. The application that called the user action can still call **GciReleaseOops** on the object that the user action needs to retain (or **GciReleaseAllOops** to release all objects at once).

To make a newly created object a permanent part of the GemStone repository, the user action has two options:

- Store the OOP of the new object into an object known to be permanent, such as a collection created by the calling application (for example, a collection created in Smalltalk and committed to the repository).
- Return the OOP of the object as the function result.

After a user action returns, the persistence of the new object is determined by the normal semantics of the calling application.

If you are working in GemBuilder for Smalltalk, you can also explicitly save these user action objects by populating a collection in the user-definable portion of `System sessionState` using `System > sessionStateAt:put:`. Your user action can retain references to objects that you add to this collection in this way.

### 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` or `gcirtl.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.

Do not call GciInit, GciLogin, or GciLogout within a user action.

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 GciDeclareAction once for each function in the set of user actions.

Example 3.1

```c
static OopType doParse(void)
{
    return OOP_NIL;
}
static OopType doFetch(void)
{
    return OOP_NIL;
}

GCIUSER_ACTION_INIT_DEF()
{
    GciDeclareAction("doParse", doParse, 1, 0, TRUE);
    GciDeclareAction("doFetch", doFetch, 1, 0, TRUE);
    // ...
}
```

GciDeclareAction 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 of arguments that the function takes. A call to GciDeclareAction looks similar to this:

GciDeclareAction("userActionName", userActionFunction, 1, 0, TRUE)

The function installs 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 third argument to `GciDeclareAction` 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`.

The fourth argument to `GciDeclareAction` is rarely used. The final argument indicates whether to return an error if there is already a user action with the specified name.

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

### 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**

```c
#include "gciuser.hf"

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, “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 \texttt{gciua.hf} instead of \texttt{gci.hf} or \texttt{gcirtl.hf}. Define a \texttt{GciUserActionShutdown}, and a \texttt{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.
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:
  ```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 class>>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: `libua.so`
- HP-UX: `libua.sl`
- AIX: `libua.so`
- Linux (x86_64): `libua.so`
- Darwin: `libua.dylib`
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 56.

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.h or gci.h in your C source code, instead of gciua.h. (Your C source code should include exactly one of these three include files.)

The GciUserActionInit and GciUserActionShutdown functions are not required, but the application must call GciDeclareAction 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 GciDeclareAction. 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:

```smalltalk
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:

```smalltalk
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 function.

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.

CAUTION

Debug your C code in a process that does not include a Gem.
For more information, see “Risk of Database Corruption” on page 81.

Use the instructions for user actions in Chapter 4, “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 class>>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 management, 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 Topaz as 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.

CAUTION
Debug your user actions in a process that does not include a Gem.
For more information, see “Risk of Database Corruption” on page 81.

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.

CAUTION
Debug your user actions in a process that does not include a Gem.
For more information, see “Risk of Database Corruption” on page 81.

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

The focus is directly on operations for each compiling or linking alternative on each GemStone server platform. 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/S 64 Bit Release Notes, Installation Guide, and vendor publications for possible updates.*
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 and C++ libraries on some platforms. 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/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

C applications and user actions must be compiled and linked with a compiler that is compatible with GemStone libraries and object code.

The example compiler and linker command lines in this chapter assume that a compatible compiler has been installed and is in your path.

The following C++ compilers were used to produce the GemStone product, and have been tested for producing C/C++ applications and user action libraries.

- Solaris (SPARC) — Sun C++ 5.8 Patch 121017-05 2006/08/30
- Solaris (x86) — Sun C++ 5.10 SunOS_i386 128229-09 2010/06/24
- HP-UX (Itanium) — HP C/aC++ B3910B A.06.25.02 [Nov 5 2010]

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.
• AIX — IBM XL C/C++ for AIX, V11.1 (5724-X13)
  Version: 11.01.0000.0004
• Linux — g++ (GCC) 4.1.2 20070115 (prerelease) (SUSE Linux)
• Darwin — g++: i686-apple-darwin10-g++-4.2.1 (GCC) 4.2.1 (Apple Inc. build 5664)
• Windows — Microsoft Visual C++ 2008 9160-270-6514982-60495
  Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM

Other compilers, such as ANSI C++ compilers, are assumed to work, but have not been tested.

Listing the Version of Your Compiler

To list the version of your compiler, execute the appropriate command line.

**Solaris (SPARC):**

% CC -V
CC: Sun C++ 5.8 Patch 121017-05 2006/08/30

**Solaris (x86):**

% CC -V
CC: Sun C++ 5.10 SunOS_i386 128229-09 2010/06/24

**HP-UX (Itanium):**

% aCC -V
aCC: HP C/aC++ B3910B A.06.25.02 [Nov 25 2010]

**AIX (IBM):**

% /usr/vacpp/bin/xlC_r -qversion
IBM XL C/C++ for AIX, V11.1 (5724-X13)
Version: 11.01.0000.0004

**Linux:**

% g++ --version
 g++ (GCC) 4.1.2 20070115 (prerelease) (SUSE Linux)

**Darwin:**

% g++ --version
 g++: i686-apple-darwin10-g++-4.2.1 (GCC) 4.2.1 (Apple Inc. build 5664)
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.

For more information and details on the listed compiler options and other compiler flags, please consult your compiler documentation.

Compilation Command Lines

This section presents simple example command lines for compiling C source code on each platform.

The command lines for each platform illustrates how to compile a simple application program or user action file named usercode, whose source contains one code file, usercode.c. Its result is one object file, usercode.o.

For simplicity in compiling code for user actions, 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.

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

Solaris (SPARC):

$ CC -xO4 -xcode=pic32 -m64 -mt -xchip=ultra2 -D_REENTRANT
   -D_POSIX_PTHREAD_SEMANTICS -I$GEMSTONE/include -c userCode.c
   -o userCode.o

   To allow debugging of the resulting library, include the optional -g flag and omit the optimization flag -xO4.

Solaris (x86):

$ CC -xO4 -m64 -Kpic -mt -D_REENTRANT -D_POSIX_PTHREAD_SEMANTICS
   -I$GEMSTONE/include -c userCode.c -o userCode.o

   To allow debugging of the resulting library, include the optional -g flag and omit the optimization flag -xO4.
HP-UX (Itanium):

$ /opt/aCC/bin/aCC +O2 +Onolimit +Z +DD64 +DSitaminum2 -Aa
   -D_PSTAT64 -D_LARGEFILE64_SOURCE -mt -Wl,+vnocompatwarnings
   +W212,749,740,863,2225,2175,2177,4232,4189,4070,20011,20009,2368
   -D_HPUX -D_POSIX_C_SOURCE=199506L -D_HPUX_SOURCE
   -D_INCLUDE_LONGLONG -D_XOPEN_SOURCE=600
   -D_XOPEN_SOURCE_EXTENDED=1 +W2 -I$GEMSTONE/include -c
   -S userCode.c -o userCode.o

   The -Aa switch is required; it designates the ANSI C mode. The +Z switch is
   required for user action library code. To allow debugging of the resulting library,
   also include the optional -g flag and omit the optimization flag +O2.

AIX (IBM):

$ /usr/vacpp/bin/x1C_r -O3 -qstrict -qalias=noansi -q64 -qarch=pwr5
   -qthreaded -qminimaltoc -qmaxmem=-1
   -qnoeh -I$GEMSTONE/include -c userCode.c
   -o userCode.o

   Note that there is no space in the -qsuppress arguments that are continued on
   the following line.

   To allow debugging of the resulting library, also include the optional -g,
   -qdbxextra and -qfullpath flags, and omit the optimization flag -O3.

Linux:

$ /usr/bin/g++ -fmessage-length=0 -fcheck-new -O3 -ggdb -m64
   -pipe -D_REENTRANT -D_GNU_SOURCE -pthread -fPIC -m64
   -fno-strict-aliasing -fno-exceptions
   -I$GEMSTONE/include -x c++ -c userCode.c -o userCode.o

   The following warn flags are recommended for compilation:
   -Wformat -Wtrigraphs -Wcomment -Wsystem-headers -Wtrigraphs
   -Wno-aggregate-return -Wswitch -Wshadow -Wunused-value
   -Wunused-variable -Wunused-label -Wunused-function
   -Wunused-argument -Wunused-macros -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -Wunused-parameter
   -Wunused-result -Wunused-parameter -Wunused-struct -unused-paramete

   June 2011  VMware, Inc.  79
If you want to stop the compilation process when any of the above warnings are encountered, use the following flag:

-Werror

To allow debugging of the resulting library, also include the optional -g flag and omit the optimization flag -O3.

**Darwin:**

```bash
/usr/bin/g++ -fmessage-length=0 -fcheck-new -O3 -ggdb -DFLAG_FAST=1 -m64 -pipe -D_XOPEN_SOURCE -D_REENTRANT -D_GNU_SOURCE -fpic -m64 -fno-strict-aliasing -I$GEMSTONE/include -x c++ -c userCode.c -o userCode.o
```

The following warn flags are recommended for compilation:


To allow debugging of the resulting library, also include the optional -g flag and omit the optimization flag -O3.

**Windows:**

```bash
cl /W3 /MD /Zi /TP /nologo /DWIN32 /D_CONSOLE /D_DLL /DNATIVE /I 'VisualStudioInstallPath\VC\atlmfc\include'
/I 'VisualStudioInstallPath\VC\include' /I 'C:\Program Files\Microsoft SDKs\Windows\v6.0A\Include'
/I%GEMSTONE%\include -c userCode.c -ouserCode.obj
```
4.3 Linking C/C++ Object Code with GemStone

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

Run-time 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.

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**

**CAUTION**

*Debug your C/C++ code in a process that does not include a Gem.*

*Do not log into GemStone in a linked application or run a Gem with your user actions until your C/C++ code has been properly debugged.*

When your C/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.

**Linker**

Use the same C++ compiler to link your GemStone C/C++ code as you use to compile it.

**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**

This section presents simple example command lines for linking object code on each platform. Each command line illustrates how to link a simple application program with one application object file, `userCode.o`. Its result is one executable file, `userAppl` or `userAppl.exe`, depending on your platform.

In addition, this section illustrates how to link a user action object file named `userCode.o` with GemStone libraries to produce a user action library named `libuserAct.so`, `libuserAct.sl`, or `libuserAct.dylib`, depending on your platform.

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

Use the same C++ compiler to link your GemStone C/C++ code as you used to compile it.
Linking Applications That Bind to GemBuilder at Run Time

**Solaris (SPARC):**

```
$ CC -xildoff -xarch=v9 -i userCode.o
   $GEMSTONE/lib/gcirtlobj.o -z nodefs -Bdynamic -lc
      -lpthread -ldl -lrt -lsocket -lnsl -lm -lCrun -o userAppl
```

**Solaris (x86):**

```
$ CC -xildoff -m64 -i appl.o $GEMSTONE/lib/gcirtlobj.o
   -z nodefs -Bdynamic -lc -lpthread -ldl -lrt -lsocket
      -lnsl -lm -lCrun -o appl
```

**HP-UX (Itanium):**

```
$/opt/aCC/bin/aCC -v +DD64 +DSitanium2
   -Wl,+allowdups,+k,+n,+pd4M,+pi64K -z userCode.o
   $GEMSTONE/lib/gcirtlobj.o
   -Wl,+allowunsats,+vnoshlibunsats -lxnet -lrt -lsec
      -ldld -lm -l:libcres.a -lcl -lpthread -o userAppl
```

On HP-UX, to enable debugging of your application, issue the `pxdb` command on the application that loads the DLL:

```
$/opt/langtools/bin/pxdb -s enable userAppl
```

**AIX (IBM):**

```
$/usr/vacpp/bin/xlC_r -Wl,-bdatapsize:64K -q64 userCode.o
   $GEMSTONE/lib/gcirtlobj.o -Wl,-berok -L/usr/vacpp/lib
   -lpthreads -lc_r -lC_r -lm -ldl -lbsd -Wl,-brtllib -q64 -o userAppl
```

**Linux:**

```
$/usr/bin/g++ userCode.o $GEMSTONE/lib/gcirtlobj.o -m64
   -lpthread -lcrypt -ldl -lc -lm -lrt -Wl,-z,muldefs -o userAppl
   -Wl,--warn-unresolved-symbols
```

**Darwin:**

```
$/usr/bin/g++ userCode.o $GEMSTONE/lib/gcirtlobj.o -m64
   -lpthread -ldl -lc -lm -o userAppl -undefined
dynamic_lookup
```
Linking C/C++ Object Code with GemStone

84

 VMware, Inc.  June 2011

Windows:

$ link /LIBPATH:’VisualStudioInstallPath\VC\lib’
/LIBPATH:’VisualStudioInstallPath\VC\atlmfc\lib’
/LIBPATH:’C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib’
-INCREMENTAL:NO -nologo ’userCode.obj’
’%GEMSTONE%\lib\gcirpc.lib’  wsock32.lib netapi32.lib
advapi32.lib comdlg32.lib user32.lib gdi32.lib
kernel32.lib winspool.lib -out:’userAppl.exe’

Linking User Actions into Shared Libraries

Solaris (SPARC):

$ CC -xarch=v9 -G -Bsymbolic -h libuserAct.so -i userCode.o
$GEMSTONE/lib/gciualib.o -o libuserAct.so -Bdynamic -lc
-lpthread -ldl -lrt -lsocket -lnsl -lm -lCrun -z nodefs

Solaris (x86):

$ CC -m64 -G -Bsymbolic -h libuserAct.so -i userCode.o
$GEMSTONE/lib/gciualib.o -o libuserAct.so -Bdynamic -lc
-lpthread -ldl -lrt -lsocket -lnsl -lm -lCrun -z nodefs

HP-UX (Itanium):

$ /opt/aCC/bin/aCC -v +DD64 +DSitanium2
-Wl,+allowdups,+k,+n,+pd4M,+pi64K -v -b
-Wl,-B,symbolic -z userCode.o $GEMSTONE/lib/gciualib.o
-o libuserAct.so -Wl,-e,GciUserActionLibraryMain -lxnet
-lrt -lsec -ldld -lm -l:libcres.a -lc -lCsup -lunwind
-Wl,+allowunsats,+vnoshlibunsats

On HP-UX, to enable debugging of your user action code, issue the pxdb
command with whichever application will load the user action, Gem or Topaz:

$ /opt/langtools/bin/pxdb -s enable $GEMSTONE/sys/gem

or

$ /opt/langtools/bin/pxdb -s enable $GEMSTONE/bin/topaz

AIX (IBM):

$ /usr/vacpp/bin/xlC_r -G -q64 userCode.o
$GEMSTONE/lib/gciualib.o -o libuserAct.so
-e GciUserActionLibraryMain -L/usr/vacpp/lib
-lpthreads -lc_r -LC_r -lm -ldl -lbsd -Wl,-berok
Linux:
$ /usr/bin/g++ -shared -Wl,-Bdynamic,-hlib\texttt{userAct}.so \texttt{userCode}.o
\$\texttt{GEMSTONE/lib/gciualib.o} -o \texttt{libuserAct}.so -m64 -lpthread
-lcrypt -ldl -lc -lm -Wl,-z,muldefs
-\texttt{Wl,--warn-unresolved-symbols}

Darwin:
$ /usr/bin/g++ -dynamiclib \texttt{userCode}.o
\$\texttt{GEMSTONE/lib/gciualib.o} -o \texttt{libuserAct}.dylib -m64
- lpthread -ldl -lc -lm -undefined dynamic\_lookup
GemBuilder
Files and Data Structures

This chapter describes the GemBuilder include files, data structures, and other reference information that may be useful when writing your application.

5.1 GemBuilder Include Files

The following include files are provided for use with GemBuilder C functions. These files are in the $GEMSTONE/include directory.

Your C source code should include *exactly* one of these three include files:

- **gcirtl.hf**: Forward references to the GemBuilder functions, to be included in code that will bind to GemBuilder at run time. For a discussion of how to load a library that was compiled against gcirtl.hf, see “Binding to GemBuilder at Run Time” on page 55.

  For modules that define user actions, use gciua.hf instead of this file.

- **gciua.hf**: Used instead of gcirtl.hf in modules that define user actions.

- **gci.hf**: Forward references to the GemBuilder functions; indirectly included by gcirtl.hf and gciua.hf.
Used for a C application that will call Gcilnit and GciLogin, on platforms that allow shared libraries to be built containing references to unresolved symbols (which are defined in gci.hf and resolved at run time).

In addition, your code can include these files:

**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.

**gcisend.hf**  Inline implementations of deprecated GciSendMsg, which no longer uses variable arguments. New code should use GciPerform. For convenience, if you are using GciSendMsg in your GemBuilder application, include this file after the include of gcirtl.hf.

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

**flag.ht**  Contains host-specific C definitions for compilation.

**gci.hf**  Defines C types for use by GemBuilder functions. See “GemBuilder Data Types” on page 89.

**gcicmn.hf**  Defines common C types and macros used by gcirtl.hf, gci.hf, and gciua.hf.

**gcierr.hf**  Defines mnemonics for all GemStone errors.

**gcioc.hf**  Defines C mnemonics for sizes and offsets into objects.

**gcioop.hf**  Defines C mnemonics for predefined GemStone objects. See Appendix A, “Reserved OOPs,” for a list of constants defined in this file.

**gcirtl.hf**  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.

**version.hf**  Defines C mnemonics for version-dependent strings.
5.2 GemBuilder Data Types

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

**BoolType**  
An int.

**ByteType**  
An unsigned 8-bit integer.

**OopType**  
Object-oriented pointer, an unsigned 32-bit integer.

**FloatKindEType**  
Enumerated type that defines the possible kinds of an IEEE binary float.

**GciClampedTravArgsSType**  
A C++ class for clamped traversal arguments.

**GciDateTimeSType**  
A structure for representing GemStone dates and times.

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

**GciErrSType**  

**GciJumpBufSType**  
Jump buffer, defined in the `setjmp.h` file.

**GciObjInfoSType**  
A C++ class for a GemStone object information report (see “The Object Information Structure” on page 92).

**GciObjRepHdrSType**  
A C++ class for an object report header (see “The Object Report Header Class” on page 95).

**GciObjRepSType**  
A C++ class for an object report (see “The Object Report Structure” on page 94).

**GciSessionIdType**  
A signed 32-bit integer.

**GciStoreTravDoArgsSType**  
A C++ class for store traversal arguments.
GciTravBufType
A traversal buffer. See “The Traversal Buffer Type” on page 100.

GciUserActionSType
A structure for describing a user action (see “The User Action Information Structure” on page 99.

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:

```c
#if !defined(GCICMN_HT)
typedef struct {
    int year;
    int dayOfYear;
    int milliseconds;
    OopType timeZone;
} GciDateTimeSType;
#endif
```

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

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

```c
#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
Deprecated. The value is always OOP_GEMSTONE_ERROR_CAT.

OopType context
The OOP of a GsProcess that provides the state of the virtual machine for use in debugging. This GsProcess 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, your program can ignore this value.

OopType exceptionObj
Either an instance of Exception or nil (if the error was not signaled from Smalltalk execution).

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

int number
The GemStone error number (a positive integer).

int argCount
The number of arguments in the args array.

unsigned char fatal
Nonzero if this error is fatal.

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.

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++ class 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 GciFetchObjImpl (page 228).

OopType objId
OOP of the object.

OopType objClass
Class of the object; see GciFetchClass (page 211).

int64 objSize
Object's total size in bytes or OOPs; see GciFetchSize_ (page 246).

int namedSize
Number of named instance variables in the object.

unsigned short objectSecurityPolicyId
The ID of the object's security policy.

Functions

The object information class GciObjInfoSType provides the following functions:

enum {
    implem_mask = 0x03,
    indexable_mask = 0x04,
    invariant_mask = 0x08,
    partial_mask = 0x10,
    overlay_mask = 0x20
};

Defines bits to use in evaluating whether this instance is invariable, indexable, partial, or overlayed.

unsigned char isInvariant();
Returns non-zero if this object is invariant. Returns zero otherwise.

unsigned char isIndexable();
Returns non-zero if this object is indexable. Returns zero otherwise.

unsigned char isPartial();
Returns non-zero if the value buffer does not contain the entire
object; that is, the operation truncated the object’s instance variables. Returns zero otherwise.

unsigned char isOverlayed();
Returns non-zero if overlay semantics were used on this operation. Returns zero otherwise.
When the traversal is overlayed, you can use OOP_ILLEGAL to mask out instance variables that you don’t want to modify, and then store into the remaining instance variables.

unsigned char objImpl();
Returns an unsigned char in the range 0..3 that corresponds to the object’s implementation format. See the description on page 93.

void clearBits();
Sets the invariant, indexable, partial, and overlayed bits to FALSE.

void setBits(unsigned char bits);
Sets the invariant, indexable, partial, and overlayed bits.

void setObjImpl(unsigned char impl);
Defines the object’s implementation format. The argument must be an integer in the range 0..3 corresponding to the implementation format. See the description on page 93.

void setInvariant(unsigned char val);
If val is non-zero, make this object invariant.

void setIndexable(unsigned char val);
If val is non-zero, make this object indexable.

void setPartial(unsigned char val);
This function has no practical effect.

void setOverlayed(unsigned char val);
If val is non-zero, use overlay semantics on this store traversal.

Description
The gcioct.h 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 C++ class GciObjRepHdrSType) and a variable-size value buffer (an array of the values of the object’s instance variables):

```cpp
#if !defined(GCI_HT)
class GciObjRepSType { /* object report struct */
public:
    GciObjRepHdrSType hdr; /* object report header */
    union {
        ByteType        bytes[1]; /* Byte obj impl. obj’s value buff */
        OopType         oops[1]; /* Pointer obj impl. obj’s value buff*/
    } u;

    inline int64 usedBytes() const {
        return this->hdr.usedBytes();
    }

    inline GciObjRepSType* nextReport() const {
        return (GciObjRepSType*) this->hdr.nextReport();
    }

    inline ByteType* valueBufferBytes() const {
        return (ByteType*)this->u.bytes;
    }

    inline OopType* valueBuffer Oops() const {
        return (OopType*)this->u.oops;
    }
};
#endif
```

Functions

The object report class GciObjRepSType provides these functions:

```cpp
int64 usedBytes();
```

When constructing an object report buffer, returns the size of the object report, including any alignment considerations.
GciObjRepHdrSType * nextReport() ;
  Given a pointer to an object report in a traversal buffer, this
  function increments the pointer by usedBytes (the size of the object
  report).

ByteType* valueBufferBytes() ;
  Returns a pointer to the start of the body, as bytes.

OopType* valueBufferOops()
  Returns a pointer to the start of the body, as OOPs.

The Object Report Header Class

An object report header is a C++ class named GciObjRepHdrSType. This class
holds a general description of an object, and contains the following fields:

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

short namedSize
  Number of named instance variables in the object.

unsigned short objectSecurityPolicyId
  The ID of the object's security policy.

OopType objId
  OOP of the object.

OopType oclass
  Class of the object; see GciFetchClass (page 211).

int64 firstOffset
  Offset of first value to fetch or store.

Functions

The object report header class GciObjRepHdrSType provides the following
functions:

enum {
  implem_mask = 0x03,
  indexable_mask = 0x04,
  invariant_mask = 0x08,
  partial_mask   = 0x10,
  overlay_mask   = 0x20,
  no_read_auth_mask   = 0x40,
  clamped_mask = 0x80,
  unused_mask    = 0xFF00

}
all_bits_mask    = 0xFFFF
};
Defines bits to use in evaluating this instance’s implementation format, and whether this instance is indexable, invariable, partial, overlayed, readable, or clamped.

int64 idxSize();
Returns the number of indexable or varying instance variables.

void setIdxSize(int64 size);
Sets the number of indexable or varying instance variables.

void setIdxSizeBits(int64 size, unsigned char bits);
Sets both the indexable size and the eight bits defined by the enum of the mask values. Intended for GemStone use only.

int objImpl();
Returns an integer in the range 0..3 that corresponds to the object’s implementation format. See the description on page 98.

int setObjImpl(int impl);
Defines the object’s implementation format. The argument must be an integer in the range 0..3 corresponding to the implementation format. See the description on page 98.

int64 objSize();
Returns the total number of instance variables in the object (both indexable and named). See GciFetchSize_ (page 246).

void clearBits();
Sets indexable, invariable, partial, overlayed, non-readable, and clamped to FALSE.

unsigned char isClamped();
Returns non-zero if this object report is clamped. Returns zero otherwise.

unsigned char noReadAuthorization();
Returns non-zero if this object report is not readable. Returns zero otherwise.

unsigned char isInvariant();
Returns non-zero if this object report is invariant. Returns zero otherwise.
unsigned char isIndexable( );
    Returns non-zero if this object report is indexable. Returns zero otherwise.

unsigned char isPartial( );
    Returns non-zero if the value buffer does not contain the entire object; that is, the traversal operation truncated the object's instance variables. Returns zero otherwise.

unsigned char isOverlaid( );
    Returns non-zero if overlay semantics were used on this store traversal operation. Returns zero otherwise.

    When the traversal is overlayed, you can use OOP_ILLEGAL to mask out instance variables that you don't want to modify, and then store into the remaining instance variables.

void setIsClamped(unsigned char val);
    If val is non-zero, make this object report clamped.

void setNoReadAuth(unsigned char val);
    If val is non-zero, make this object report non-readable.

void setInvariant(unsigned char val);
    If val is non-zero, make this object report invariant.

void setIndexable(unsigned char val);
    If val is non-zero, make this object report indexable.

void setPartial(unsigned char val);
    This function has no practical effect.

void clearPartial(unsigned char val);
    This function has no practical effect.

void setOverlayed(unsigned char val);
    If val is non-zero, use overlay semantics on this store traversal.

ByteType* valueBufferBytes() ;
    Returns a pointer to the start of the body, as bytes.

OopType* valueBufferOops()
    Returns a pointer to the start of the body, as OOPs.

int64 usedBytes() ;
    Returns the size (in bytes) of this object report, including the size of the header, value buffer, and any padding bytes needed at the
GciObjRepHdrSType * nextReport() ;
Given a pointer to an object report in a traversal buffer, this
function increments the pointer by usedBytes (the size of the object
report).

Description
During a store traversal operation, if the specified idxSize is inadequate to
accommodate the contents of the value buffer (the values in u.bytes or u.oops), the
store operation will automatically increase idxSize (the number of the object’s
indexed variables) as needed. If the specified objClass is not indexable, then the
idxSize is ignored; in addition, if there are more OOPs in the value buffer than there
are named instance variables, and the object is not an NSC, an error will be
generated.

During a store traversal operation, 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).

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.

The `isInvariant()` value is true if the object itself is invariant. This can happen in one of three ways:

• The application program sends the message `immediateInvariant` to the object.

• The application program explicitly executes `setInvariant()` in the report header and then uses that report header in a call to `GciStoreTrav`.

• The object’s class was created with `instancesInvariant: true` and the object has been committed.

Table 5.1 Object Implementation Restrictions on Instance Variables

<table>
<thead>
<tr>
<th>Object Implementation</th>
<th>Named Instance Variables OK?</th>
<th>Unnamed Instance Variables OK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=Pointer</td>
<td>YES</td>
<td>YES (always indexed)</td>
</tr>
<tr>
<td>1=Byte</td>
<td>NO</td>
<td>YES (always indexed)</td>
</tr>
<tr>
<td>2=NSC</td>
<td>YES</td>
<td>YES (always unordered)</td>
</tr>
<tr>
<td>3=Special</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

For more information about object implementation types, see “Manipulating Objects Through Structural Access” on page 34.

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.

`int userActionNumArgs`
The number of arguments in the C function.

`GciUserActionFType userAction`
A pointer to the C user action function.
unsigned int  userActionFlags
    Mainly for internal use. If you use it, set it to 0 before passing a pointer to it.

The Traversal Buffer Type

The C++ class GciTravBufType describes a traversal buffer, and defines the following fields:

- int64  allocatedBytes
  The allocated size of the body.
- int64  usedBytes
  The used bytes of the body.
- ByteType  body[8]
  The actual body size is variable, with a minimum of GCI_MIN_TRAV_BUFF_SIZE.

Functions

The following function call is used to create an instance of GciTravBufType:

```
static GciTravBufType* malloc(size_t allocationSize);
```

Returns an instance obtained from ::malloc initialized with allocatedBytes equal to allocationSize and usedBytes== 0. (If allocationSize is not a multiple of 8 bytes, allocatedBytes is rounded up to the next 8-byte multiple.) Returns NULL if malloc fails.

The traversal buffer class GciTravBufType provides these functions:

- GciObjRepSType* firstReport() ;
  Returns a pointer to the first object report in the buffer.
- GciObjRepSType* readLimit() ;
  Used when reading object reports out of a buffer. Returns a pointer past the end of last object report in the buffer.
    If readLimit()==firstReport(), the buffer is empty.
- GciObjRepSType* writeLimit() ;
  Used when composing a buffer. Returns a pointer one byte past the end of the allocated buffer.
- GciObjRepHdrSType* firstReportHdr() ;
  Returns a pointer to the first object report in the buffer.
GciObjRepHdrSType* readLimitHdr();
    Used when reading object reports out of a buffer. Returns a
    pointer past the end of last object report in the buffer.

GciObjRepHdrSType* writeLimitHdr();
    Used when composing a buffer. Returns a pointer one byte past
    the end of the allocated buffer.

5.3 Structural Access Functions

A number of functions access Smalltalk objects structurally, rather than via
executing message sends. A list of these functions is in Table 6.8 on page 111.

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.

Structural access functions do not bypass authorization checks or other checks that
are not done in Smalltalk code.

5.4 environmentId

In GemStone/S 64 Bit 3.0, many new GCI functions are identical to existing GCI
functions, but with two key differences:

- The new function takes an environmentId argument.
- The name of the new function includes a trailing underscore.

The environmentId argument allows a GCI function to specify one of up to 256
execution environments, for use in Ruby applications.

Smalltalk applications do not need to know anything about environmentId. With
Smalltalk applications, it is preferable to use the existing GCI function (without
the trailing underscore).

For an example of this, see GciExecute (page 191). The syntax section on that page
shows both variants: GciExecute (used with Smalltalk applications) and
GciExecute_ (used with Ruby applications).
5.5 UNIX Signal Handling

Both versions of GemBuilder (GciLnk and GciRpc) use the SIGIO signal handler. GciLnk also uses the signals SIGSEGV and SIGVTALRM. SIGVTALRM is used by the ProfMonitor class.

If you must install your own signal handler (using signal or sigvec) for any of these signals, be sure that your application signal handler chains to the previous handler when done. Similar chaining is required for SIGVTALRM, if you intend to use ProfMonitor.

SIGSEGV occurs normally when a Smalltalk stack overflow occurs, and is translated to a Smalltalk stack overflow error by the GemStone SIGSEGV handler. If you use GciLnk and install handlers for this signal after calling GciLogin, your own SIGSEGV handler must determine whether the SIGSEGV was produced by your own C code, and if not, chain to the GemStone handler.

**CAUTION**

*Do not, under any circumstances, turn off SIGIO.*
This chapter describes the GemBuilder functions that may be called by your C application program.

6.1 Function Summary Tables

Tables 6.1 through 6.9 summarize the GemBuilder C functions and the services that they provide to your application.

Table 6.1 Functions for Controlling Sessions and Transactions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciAbort</td>
<td>Abort the current transaction.</td>
</tr>
<tr>
<td>GciAlteredObjs</td>
<td>Find all exported or dirty objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.</td>
</tr>
<tr>
<td>GciBegin</td>
<td>Begin a new transaction.</td>
</tr>
<tr>
<td>GciCommit</td>
<td>Write the current transaction to the database.</td>
</tr>
<tr>
<td>GciDeclareAction</td>
<td>An alternative way to associate a C function with a Smalltalk user action.</td>
</tr>
<tr>
<td>GciDirtyExportedObjs</td>
<td>Find all objects in the ExportedDirtyObjs set.</td>
</tr>
<tr>
<td>GciDirtyObjsInit</td>
<td>Begin tracking which objects in the session workspace change.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GciDirtySaveObjs</td>
<td>Find all exported or tracked objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.</td>
</tr>
<tr>
<td>GciDirtyTrackedObjs</td>
<td>Find all tracked objects that have changed and are therefore in the TrackedDirtyObjs set.</td>
</tr>
<tr>
<td>GciGetSessionId</td>
<td>Find the ID number of the current user session.</td>
</tr>
<tr>
<td>GciHardBreak</td>
<td>Interrupt GemStone and abort the current transaction.</td>
</tr>
<tr>
<td>GciInit</td>
<td>Initialize GemBuilder.</td>
</tr>
<tr>
<td>GciInitAppName</td>
<td>Override the default application configuration file name.</td>
</tr>
<tr>
<td>GciInitAppName_</td>
<td>Override the default application configuration file name and the size of temporary object memory.</td>
</tr>
<tr>
<td>GciInstallUserAction</td>
<td>Associate a C function with a Smalltalk user action.</td>
</tr>
<tr>
<td>GciIsRemote</td>
<td>Determine whether the application is running linked or remotely.</td>
</tr>
<tr>
<td>GciLogin</td>
<td>Load an application user action library.</td>
</tr>
<tr>
<td>GciLogin</td>
<td>Start a user session.</td>
</tr>
<tr>
<td>GciLogout</td>
<td>End the current user session.</td>
</tr>
<tr>
<td>GciNbAbort</td>
<td>Abort the current transaction (nonblocking).</td>
</tr>
<tr>
<td>GciNbBegin</td>
<td>Begin a new transaction (nonblocking).</td>
</tr>
<tr>
<td>GciNbCommit</td>
<td>Write the current transaction to the database (nonblocking).</td>
</tr>
<tr>
<td>GciNbEnd</td>
<td>Test the status of nonblocking call in progress for completion.</td>
</tr>
<tr>
<td>GciProcessDeferredUpdates_</td>
<td>Process deferred updates to objects that do not allow direct structural update.</td>
</tr>
<tr>
<td>GciReleaseAllGlobalOops</td>
<td>Remove all OOPS from the PureExportSet, making these objects eligible for garbage collection.</td>
</tr>
<tr>
<td>GciReleaseAllOops</td>
<td>Remove all OOPS from the PureExportSet, or if in a user action, from the user action’s export set, making these objects eligible for garbage collection.</td>
</tr>
<tr>
<td>GciReleaseAllTrackedOops</td>
<td>Clear the GciTrackedObjs set, making all tracked OOPs eligible for garbage collection.</td>
</tr>
<tr>
<td>GciReleaseGlobalOops</td>
<td>Remove an array of GemStone OOPs from the PureExportSet, making them eligible for garbage collection.</td>
</tr>
<tr>
<td>GciReleaseOops</td>
<td>Remove an array of GemStone OOPs from the PureExportSet, or if in a user action, remove them from the user action’s export set, making them eligible for garbage collection.</td>
</tr>
<tr>
<td>GciReleaseTrackedOops</td>
<td>Remove an array of OOPs from the GciTrackedObjs set, making them eligible for garbage collection.</td>
</tr>
</tbody>
</table>
Table 6.1  Functions for Controlling Sessions and Transactions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciRtlIsLoaded</td>
<td>Report whether a GemBuilder library is loaded.</td>
</tr>
<tr>
<td>GciRtlLoad</td>
<td>Load a GemBuilder library.</td>
</tr>
<tr>
<td>GciRtlUnload</td>
<td>Unload a GemBuilder library.</td>
</tr>
<tr>
<td>GciSaveAndTrackObjs</td>
<td>Add objects to GemStone’s internal GciTrackedObjs set to prevent them from being garbage collected.</td>
</tr>
<tr>
<td>GciSaveGlobalObjs</td>
<td>Add an array of OOPs to the PureExportSet, making them ineligible for garbage collection.</td>
</tr>
<tr>
<td>GciSaveObjs</td>
<td>Add an array of OOPs to the PureExportSet, or if in a user action to the user action’s export set, making them ineligible for garbage collection.</td>
</tr>
<tr>
<td>GciServerIsBigEndian</td>
<td>Determine whether or not the server process is big-endian.</td>
</tr>
<tr>
<td>GciSessionIsRemote</td>
<td>Determine whether or not the current session is using a Gem on another machine.</td>
</tr>
<tr>
<td>GciSetCacheName_</td>
<td>Set the name that a linked application will be known by in the shared cache.</td>
</tr>
<tr>
<td>GciSetNet</td>
<td>Set network parameters for connecting the user to the Gem and Stone processes.</td>
</tr>
<tr>
<td>GciSetSessionId</td>
<td>Set an active session to be the current one.</td>
</tr>
<tr>
<td>GciShutdown</td>
<td>Logout from all sessions and deactivate GemBuilder.</td>
</tr>
<tr>
<td>GciStep</td>
<td>Continue code execution in GemStone with specified single-step semantics.</td>
</tr>
<tr>
<td>GciTrackedObjsFetchAllDirty</td>
<td>Find all exported or tracked objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.</td>
</tr>
<tr>
<td>GciTrackedObjsInit</td>
<td>Reinitialize the set of tracked objects maintained by GemStone.</td>
</tr>
<tr>
<td>GciUserActionInit</td>
<td>Declare user actions for GemStone.</td>
</tr>
<tr>
<td>GciUserActionShutdown</td>
<td>Enable user-defined clean-up for user actions.</td>
</tr>
</tbody>
</table>

Table 6.2  Functions for Handling Errors and Interrupts and for Debugging

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciCallInProgress</td>
<td>Determine if a GemBuilder call is currently in progress.</td>
</tr>
<tr>
<td>GciClearStack</td>
<td>Clear the Smalltalk call stack.</td>
</tr>
<tr>
<td>GciContinue</td>
<td>Continue code execution in GemStone after an error.</td>
</tr>
<tr>
<td>GciContinueWith</td>
<td>Continue code execution in GemStone after an error.</td>
</tr>
<tr>
<td>GciDbgEstablish</td>
<td>Specify the debugging function for GemBuilder to execute before most calls to GemBuilder functions.</td>
</tr>
</tbody>
</table>
Table 6.2 Functions for Handling Errors and Interrupts and for Debugging (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciDbgEstablishToFile</td>
<td>Write trace information for most GemBuilder functions to a file.</td>
</tr>
<tr>
<td>GciDbgLogString</td>
<td>Pass a message to a trace function.</td>
</tr>
<tr>
<td>GciEnableSignaledErrors</td>
<td>Establish or remove GemBuilder visibility to signaled errors from GemStone.</td>
</tr>
<tr>
<td>GciErr</td>
<td>Prepare a report describing the most recent GemBuilder error.</td>
</tr>
<tr>
<td>GciInUserAction</td>
<td>Determine whether or not the current process is executing a user action.</td>
</tr>
<tr>
<td>GciLongJmp</td>
<td>Provides equivalent functionality to the corresponding longjmp() or _longjmp() function.</td>
</tr>
<tr>
<td>GciNbContinue</td>
<td>Continue code execution in GemStone after an error (nonblocking).</td>
</tr>
<tr>
<td>GciNbContinueWith</td>
<td>Continue code execution in GemStone after an error (nonblocking).</td>
</tr>
<tr>
<td>GciPopErrJump</td>
<td>Discard a previously saved error jump buffer.</td>
</tr>
<tr>
<td>GciPushErrJump</td>
<td>Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.</td>
</tr>
<tr>
<td>GciRaiseException</td>
<td>Signal an error, synchronously, within a user action.</td>
</tr>
<tr>
<td>GciSetErrJump</td>
<td>Enable or disable the current error handler.</td>
</tr>
<tr>
<td>GciSetHaltOnError</td>
<td>Halt the current session when a specified error occurs.</td>
</tr>
<tr>
<td>Gci_SETJMP (MACRO)</td>
<td>Save a jump buffer in GemBuilder’s error jump stack.</td>
</tr>
<tr>
<td>GciSoftBreak</td>
<td>Interrupt the execution of Smalltalk code, but permit it to be restarted.</td>
</tr>
</tbody>
</table>

Table 6.3 Functions for Compiling and Executing Smalltalk Code in the Database

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciClassMethodForClass</td>
<td>Compile a class method for a class.</td>
</tr>
<tr>
<td>GciCompileMethod</td>
<td>Compile a method.</td>
</tr>
<tr>
<td>GciExecute</td>
<td>Execute a Smalltalk expression contained in a String object.</td>
</tr>
<tr>
<td>GciExecuteFromContext</td>
<td>Execute a Smalltalk expression contained in a String object as if it were a message sent to another object.</td>
</tr>
<tr>
<td>GciExecuteStr</td>
<td>Execute a Smalltalk expression contained in a C string.</td>
</tr>
<tr>
<td>GciExecuteStrFromContext</td>
<td>Execute a Smalltalk expression contained in a C string as if it were a message sent to an object.</td>
</tr>
<tr>
<td>GciInstMethodForClass</td>
<td>Compile an instance method for a class.</td>
</tr>
<tr>
<td>GciNbExecute</td>
<td>Execute a Smalltalk expression contained in a String object (nonblocking).</td>
</tr>
</tbody>
</table>
### Table 6.3 Functions for Compiling and Executing Smalltalk Code in the Database

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciNbExecuteStr</td>
<td>Execute a Smalltalk expression contained in a C string (nonblocking).</td>
</tr>
<tr>
<td>GciNbExecuteStrFromContext</td>
<td>Execute a Smalltalk expression contained in a C string as if it were a message sent to an object (nonblocking).</td>
</tr>
<tr>
<td>GciNbPerform</td>
<td>Send a message to a GemStone object (nonblocking).</td>
</tr>
<tr>
<td>GciNbPerformNoDebug</td>
<td>Send a message to a GemStone object, and temporarily disable debugging (nonblocking).</td>
</tr>
<tr>
<td>GciPerform</td>
<td>Send a message to a GemStone object.</td>
</tr>
<tr>
<td>GciPerformNoDebug</td>
<td>Send a message to a GemStone object, and temporarily disable debugging.</td>
</tr>
<tr>
<td>GciPerformSymDbg</td>
<td>Send a message to a GemStone object, using a String object as a selector.</td>
</tr>
<tr>
<td>GciPerformTraverse</td>
<td>First send a message to a GemStone object, then traverse the result of the message.</td>
</tr>
</tbody>
</table>

### Table 6.4 Functions for Accessing Symbol Dictionaries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciResolveSymbol</td>
<td>Find the OOP of the object to which a symbol name refers, in the context of the current session’s user profile.</td>
</tr>
<tr>
<td>GciResolveSymbolObj</td>
<td>Find the OOP of the object to which a symbol object refers, in the context of the current session’s user profile.</td>
</tr>
<tr>
<td>GciStrKey VALUE Dict At</td>
<td>Find the value in a symbol KeyValue dictionary at the corresponding string key.</td>
</tr>
<tr>
<td>GciStrKey VALUE Dict At Obj Put</td>
<td>Store a value into a symbol KeyValue dictionary at the corresponding object key.</td>
</tr>
<tr>
<td>GciTraverseObjs</td>
<td>Traverse an array of GemStone objects.</td>
</tr>
</tbody>
</table>
### Table 6.5 Functions for Creating and Initializing Objects

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciCreateByteObj</td>
<td>Create a new byte-format object.</td>
</tr>
<tr>
<td>GciCreateOopObj</td>
<td>Create a new pointer-format object.</td>
</tr>
<tr>
<td>GciGetFreeOop</td>
<td>Allocate an OOP.</td>
</tr>
<tr>
<td>GciGetFreeOops</td>
<td>Allocate multiple OOPs.</td>
</tr>
<tr>
<td>GciGetFreeOopsEncoded</td>
<td>Allocate multiple OOPs.</td>
</tr>
<tr>
<td>GciNewByteObj</td>
<td>Create and initialize a new byte object.</td>
</tr>
<tr>
<td>GciNewCharObj</td>
<td>Create and initialize a new character object.</td>
</tr>
<tr>
<td>GciNewDateTime</td>
<td>Create and initialize a new date-time object.</td>
</tr>
<tr>
<td>GciNewOop</td>
<td>Create a new GemStone object.</td>
</tr>
<tr>
<td>GciNewOops</td>
<td>Create multiple new GemStone objects.</td>
</tr>
<tr>
<td>GciNewOopUsingObjRep</td>
<td>Create a new GemStone object from an existing object report.</td>
</tr>
<tr>
<td>GciNewString</td>
<td>Create a new String object from a C character string.</td>
</tr>
<tr>
<td>GciNewSymbol</td>
<td>Create a new Symbol object from a C character string.</td>
</tr>
</tbody>
</table>

### Table 6.6 Functions and Macros for Converting Objects and Values

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCI_BOOL_TO_OOP (MACRO)</td>
<td>Convert a C Boolean value to a GemStone Boolean object.</td>
</tr>
<tr>
<td>GciByteArrayToPointer</td>
<td>Given a result from GciPointerToByteArray, return a C pointer.</td>
</tr>
<tr>
<td>GCI_CHR_TO_OOP (MACRO)</td>
<td>Convert a C character value to a GemStone Character object.</td>
</tr>
<tr>
<td>GciCTimeToDateTime</td>
<td>Convert a C date-time representation to the equivalent GemStone representation.</td>
</tr>
<tr>
<td>GciDateTimeToCTime</td>
<td>Convert a GemStone date-time representation to the equivalent C representation.</td>
</tr>
<tr>
<td>Gci_doubleToSmallDouble</td>
<td>Convert a C double to a SmallDouble object.</td>
</tr>
<tr>
<td>GciFetchDateTime</td>
<td>Convert the contents of a DateTime object and place the results in a C structure.</td>
</tr>
<tr>
<td>GciFloatKind</td>
<td>Obtain the float kind corresponding to a C double value.</td>
</tr>
<tr>
<td>GciFltToOop</td>
<td>Convert a C double value to a SmallDouble or Float object.</td>
</tr>
<tr>
<td>GCI_I64_IS_SMALL_INT</td>
<td>Determine whether or not a C 64-bit integer value can be translated into a SmallInteger object.</td>
</tr>
<tr>
<td>GciI64ToOop</td>
<td>Convert a C 64-bit integer value to a GemStone object.</td>
</tr>
</tbody>
</table>
### Table 6.6  Functions and Macros for Converting Objects and Values (Continued)

<table>
<thead>
<tr>
<th>Function/Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCI_OOP_IS_BOOL</td>
<td>(MACRO) Determine whether or not a GemStone object represents a Boolean value.</td>
</tr>
<tr>
<td>GCI_OOP_IS_SMALL_INT</td>
<td>(MACRO) Determine whether or not a GemStone object represents a SmallInteger.</td>
</tr>
<tr>
<td>GCI_OOP_IS_SPECIAL</td>
<td>(MACRO) Determine whether or not a GemStone object has a special representation.</td>
</tr>
<tr>
<td>GciOopToBool</td>
<td>Convert a Boolean object to a C Boolean value.</td>
</tr>
<tr>
<td>GCI_OOP_TO_BOOL</td>
<td>(MACRO) Convert a Boolean object to a C Boolean value.</td>
</tr>
<tr>
<td>GciOopToChar16</td>
<td>Convert a Character object to a 16-bit C character value.</td>
</tr>
<tr>
<td>GciOopToChar32</td>
<td>Convert a Character object to a 32-bit C character value.</td>
</tr>
<tr>
<td>GciOopToChr</td>
<td>Convert a Character object to a C character value.</td>
</tr>
<tr>
<td>GCI_OOP_TO_CHR</td>
<td>(MACRO) Convert a Character object to a C character value.</td>
</tr>
<tr>
<td>GciOopToFlt</td>
<td>Convert a SmallDouble, Float, or SmallFloat object to a C double.</td>
</tr>
<tr>
<td>GciOopToI32</td>
<td>Convert a GemStone object to a C 32-bit integer value.</td>
</tr>
<tr>
<td>GciOopToI32_</td>
<td>Convert a GemStone object to a C 32-bit integer value, with error handling.</td>
</tr>
<tr>
<td>GciOopToI64</td>
<td>Convert a GemStone object to a C 64-bit integer value.</td>
</tr>
<tr>
<td>GciOopToI64_</td>
<td>Convert a GemStone object to a C 64-bit integer value, with error handling.</td>
</tr>
<tr>
<td>GciPointerToByteArray</td>
<td>Given a C pointer, return a SmallInteger or ByteArray containing the value of the pointer.</td>
</tr>
<tr>
<td>GciStringToInteger</td>
<td>Convert a C string to a GemStone SmallInteger, LargePositiveInteger or LargeNegativeInteger object.</td>
</tr>
</tbody>
</table>

### Table 6.7  Object Traversal and Path Functions and Macros

<table>
<thead>
<tr>
<th>Function/Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCI_ALIGN</td>
<td>(MACRO) Align an address to a word boundary.</td>
</tr>
<tr>
<td>GciClampedTrav</td>
<td>Traverse an array of objects, subject to clamps.</td>
</tr>
<tr>
<td>GciClamped TraverseObjs</td>
<td>Traverse an array of objects, subject to clamps.</td>
</tr>
<tr>
<td>GciExecuteStrTrav</td>
<td>Execute a string and traverse the result of the execution.</td>
</tr>
<tr>
<td>GciFetchPaths</td>
<td>Fetch selected multiple OOPs from an object tree.</td>
</tr>
<tr>
<td>GciFindObjRep</td>
<td>Fetch an object report in a traversal buffer.</td>
</tr>
<tr>
<td>GciMoreTraversal</td>
<td>Continue object traversal, reusing a given buffer.</td>
</tr>
<tr>
<td>GciNbClampedTrav</td>
<td>Traverse an array of objects, subject to clamps (nonblocking).</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>GciNbClampedTraverseObjs</td>
<td>Traverse an array of objects, subject to clamps (nonblocking).</td>
</tr>
<tr>
<td>GciNbExecuteStrTrav</td>
<td>Execute a string and traverse the result of the execution (nonblocking).</td>
</tr>
<tr>
<td>GciNbMoreTraversal</td>
<td>Continue object traversal, reusing a given buffer (nonblocking).</td>
</tr>
<tr>
<td>GciNbPerformTrav</td>
<td>First send a message to a GemStone object, then traverse the result of the message (nonblocking).</td>
</tr>
<tr>
<td>GciNbStoreTrav</td>
<td>Store multiple traversal buffer values in objects (nonblocking).</td>
</tr>
<tr>
<td>GciNbStoreTravDo_</td>
<td>Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object (non-blocking).</td>
</tr>
<tr>
<td>GciNbStoreTravDoTrav_</td>
<td>Combine in a single function the calls to GciNbStoreTravDo_ and GciNbClampedTrav, to store multiple traversal buffer values in objects, execute the specified code, and traverse the result object (non-blocking).</td>
</tr>
<tr>
<td>GciNbStoreTravDoTravRefs_</td>
<td>Combine in a single function modifications to session sets, traversal of objects to the server, optional Smalltalk execution, and traversal to the client of changed objects and (optionally) the result object (non blocking).</td>
</tr>
<tr>
<td>GciNbTraverseObjs</td>
<td>Traverse an array of GemStone objects (nonblocking).</td>
</tr>
<tr>
<td>GciObjRepSize_</td>
<td>Find the number of bytes in an object report.</td>
</tr>
<tr>
<td>GciPathToStr</td>
<td>Convert a path representation from numeric to string.</td>
</tr>
<tr>
<td>GciPerformTrav</td>
<td>First send a message to a GemStone object, then traverse the result of the message.</td>
</tr>
<tr>
<td>GciPerformTraverse</td>
<td>First send a message to a GemStone object, then traverse the result of the message.</td>
</tr>
<tr>
<td>GciSetTraversalBufSwizzling</td>
<td>Control swizzling of the traversal buffers.</td>
</tr>
<tr>
<td>GciStorePaths</td>
<td>Store selected multiple OOPs into an object tree.</td>
</tr>
<tr>
<td>GciStoreTrav</td>
<td>Store multiple traversal buffer values in objects.</td>
</tr>
<tr>
<td>GciStoreTravDo_</td>
<td>Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object.</td>
</tr>
<tr>
<td>GciStoreTravDoTrav_</td>
<td>Combine in a single function the calls to GciStoreTravDo_ and GciClampedTrav, to store multiple traversal buffer values in objects, execute the specified code, and traverse the result object.</td>
</tr>
<tr>
<td>GciStoreTravDoTravRefs_</td>
<td>Combine in a single function modifications to session sets, traversal of objects to the server, optional Smalltalk execution, and traversal to the client of changed objects and (optionally) the result object.</td>
</tr>
<tr>
<td>GciStrToPath</td>
<td>Convert a path representation from string to numeric.</td>
</tr>
</tbody>
</table>
Table 6.7 Object Traversal and Path Functions and Macros (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciTraverseObjs</td>
<td>Traverse an array of GemStone objects.</td>
</tr>
</tbody>
</table>

CAUTION

Exercise caution when using the following 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 authorization violations or concurrency conflicts.

Table 6.8 Structural Access Functions and Macros

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciAddOopToNsc</td>
<td>Add an OOP to the unordered variables of a nonsequenceable collection.</td>
</tr>
<tr>
<td>GciAddOopsToNsc</td>
<td>Add multiple OOPs to the unordered variables of a nonsequenceable collection.</td>
</tr>
<tr>
<td>GciAppendBytes</td>
<td>Append bytes to a byte object.</td>
</tr>
<tr>
<td>GciAppendChars</td>
<td>Append a C string to a byte object.</td>
</tr>
<tr>
<td>GciAppendOops</td>
<td>Append OOPs to the unnamed variables of a collection.</td>
</tr>
<tr>
<td>GciClassNamedSize</td>
<td>Find the number of named instance variables in a class.</td>
</tr>
<tr>
<td>GciFetchByte</td>
<td>Fetch one byte from an indexed byte object.</td>
</tr>
<tr>
<td>GciFetchBytes_</td>
<td>Fetch multiple bytes from an indexed byte object.</td>
</tr>
<tr>
<td>GciFetchChars_</td>
<td>Fetch multiple ASCII characters from an indexed byte object.</td>
</tr>
<tr>
<td>GciFetchClass</td>
<td>Fetch the class of an object.</td>
</tr>
<tr>
<td>GciFetchNamedOop</td>
<td>Fetch the OOP of one of an object’s named instance variables.</td>
</tr>
<tr>
<td>GciFetchNamedOops</td>
<td>Fetch the OOPs of one or more of an object’s named instance variables.</td>
</tr>
<tr>
<td>GciFetchNamedSize</td>
<td>Fetch the number of named instance variables in an object.</td>
</tr>
<tr>
<td>GciFetchNameOfClass</td>
<td>Fetch the class name object for a given class.</td>
</tr>
<tr>
<td>GciFetchObjectInfo</td>
<td>Fetch information and values from an object.</td>
</tr>
<tr>
<td>GciFetchObjImpl</td>
<td>Fetch the implementation of an object.</td>
</tr>
<tr>
<td>GciFetchObjInfo</td>
<td>Fetch information and values from an object.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GciFetchOop</td>
<td>Fetch the OOP of one instance variable of an object.</td>
</tr>
<tr>
<td>GciFetchOops</td>
<td>Fetch the OOPs of one or more instance variables of an object.</td>
</tr>
<tr>
<td>GciFetchSize_</td>
<td>Fetch the size of an object.</td>
</tr>
<tr>
<td>GciFetchVaryingOop</td>
<td>Fetch the OOP of one unnamed instance variable from an indexable pointer object or NSC.</td>
</tr>
<tr>
<td>GciFetchVaryingOops</td>
<td>Fetch the OOPs of one or more unnamed instance variables from an indexable pointer object or NSC.</td>
</tr>
<tr>
<td>GciFetchVaryingSize_</td>
<td>Fetch the number of unnamed instance variables in a pointer object or NSC.</td>
</tr>
<tr>
<td>GciHiddenSetIncludesOop</td>
<td>Determines whether the given OOP is present in the specified hidden set.</td>
</tr>
<tr>
<td>GciIsKindOf</td>
<td>Determine whether or not an object is some kind of a given class or class history.</td>
</tr>
<tr>
<td>GciIsKindOfClass</td>
<td>Determine whether or not an object is some kind of a given class.</td>
</tr>
<tr>
<td>GciIsSubclassOf</td>
<td>Determine whether or not a class is a subclass of a given class or class history.</td>
</tr>
<tr>
<td>GciIsSubclassOfClass</td>
<td>Determine whether or not a class is a subclass of a given class.</td>
</tr>
<tr>
<td>GciIVNameToIdx</td>
<td>Fetch the index of an instance variable name.</td>
</tr>
<tr>
<td>GciNscIncludesOop</td>
<td>Determines whether the given OOP is present in the specified unordered collection.</td>
</tr>
<tr>
<td>GciObjExists</td>
<td>Determine whether or not a GemStone object exists.</td>
</tr>
<tr>
<td>GciObjInCollection</td>
<td>Determine whether or not a GemStone object is in a Collection.</td>
</tr>
<tr>
<td>GciObjsCommitted</td>
<td>Determine whether or not an object is committed.</td>
</tr>
<tr>
<td>GciRemoveOopFromNsc</td>
<td>Remove an OOP from an NSC.</td>
</tr>
<tr>
<td>GciRemoveOopsFromNsc</td>
<td>Remove one or more OOPs from an NSC.</td>
</tr>
<tr>
<td>GciReplaceOoops</td>
<td>Replace all instance variables in a GemStone object.</td>
</tr>
<tr>
<td>GciReplaceVaryingOops</td>
<td>Replace all unnamed instance variables in an NSC object.</td>
</tr>
<tr>
<td>GciSetVaryingSize</td>
<td>Set the size of a collection.</td>
</tr>
<tr>
<td>GciStoreByte</td>
<td>Store one byte in a byte object.</td>
</tr>
<tr>
<td>GciStoreBytes</td>
<td>(MACRO) Store multiple bytes in a byte object.</td>
</tr>
<tr>
<td>GciStoreBytesInstanceOf</td>
<td>Store multiple bytes in a byte object.</td>
</tr>
<tr>
<td>GciStoreChars</td>
<td>Store multiple ASCII characters in a byte object.</td>
</tr>
<tr>
<td>GciStoreIdxOop</td>
<td>Store one OOP in an indexable pointer object’s unnamed instance variable.</td>
</tr>
</tbody>
</table>
### Table 6.8 Structural Access Functions and Macros (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciStoreIdxOops</td>
<td>Store one or more OOPs in an indexable pointer object’s unnamed instance variables.</td>
</tr>
<tr>
<td>GciStoreNamedOop</td>
<td>Store one OOP into an object’s named instance variable.</td>
</tr>
<tr>
<td>GciStoreNamedOops</td>
<td>Store one or more OOPs into an object’s named instance variables.</td>
</tr>
<tr>
<td>GciStoreOop</td>
<td>Store one OOP into an object’s instance variable.</td>
</tr>
<tr>
<td>GciStoreOops</td>
<td>Store one or more OOPs into an object’s instance variables.</td>
</tr>
</tbody>
</table>

### Table 6.9 Utility Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciCompress</td>
<td>Compress the supplied data, which can be uncompressed with GciUncompress.</td>
</tr>
<tr>
<td>GciDecodeOopArray</td>
<td>Decode an OOP array that was previously run-length encoded.</td>
</tr>
<tr>
<td>GciDecSharedCounter</td>
<td>Decrement the value of a shared counter.</td>
</tr>
<tr>
<td>GciEnableFreeOopEncoding</td>
<td>Enable run-length encoding of free OOPs.</td>
</tr>
<tr>
<td>GciEnableFullCompression</td>
<td>Enable full compression between the client and the RPC version of GemBuilder.</td>
</tr>
<tr>
<td>GciEncodeOopArray</td>
<td>Encode an array of OOPs, using run-length encoding.</td>
</tr>
<tr>
<td>GciFetchNumEncodedOops</td>
<td>Obtain the size of an encoded OOP array.</td>
</tr>
<tr>
<td>GciFetchNumSharedCounters</td>
<td>Obtain the number of shared counters available on the shared page cache used by this session.</td>
</tr>
<tr>
<td>GciFetchSharedCounterValuesNoLock</td>
<td>Fetch the value of multiple shared counters without locking them.</td>
</tr>
<tr>
<td>GciIncSharedCounter</td>
<td>Increment the value of a shared counter.</td>
</tr>
<tr>
<td>GciProduct</td>
<td>Return an 8-bit unsigned integer that indicates the GemStone/S product.</td>
</tr>
<tr>
<td>GciReadSharedCounter</td>
<td>Lock and fetch the value of a shared counter.</td>
</tr>
<tr>
<td>GciReadSharedCounterNoLock</td>
<td>Fetch the value of a shared counter without locking it.</td>
</tr>
<tr>
<td>GciSetSharedCounter</td>
<td>Set the value of a shared counter.</td>
</tr>
<tr>
<td>GciUncompress</td>
<td>Uncompress the supplied data, assumed to have been compressed with GciCompress.</td>
</tr>
<tr>
<td>GciVersion</td>
<td>Return a string that describes the GemBuilder version.</td>
</tr>
</tbody>
</table>
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 32.

See Also

GCI_CHR_TO_OOP, page 134
GciCommit, page 147
GciNbAbort, page 296
GciNbCommit, page 301
GciAddOopToNsc

Add an OOP to the unordered variables of a nonsequenceable collection.

Syntax

```c
void GciAddOopToNsc(
    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

```c
OopType GciAddOopToNsc_example(void)
{
    // return an IdentityBag containing the SmallIntegers with value 0..99
    OopType oNsc = GciNewOop(OOP_CLASS_IDENTITY_BAG);
    for (int i = 0; i < 100; i++) {
        OopType oNum = GciI32ToOop(i);
        GciAddOopToNsc(oNsc, oNum);
    }
    return oNsc;
}
```
See Also

GciAddOopsToNsc, page 117
GciNscIncludesOop, page 343
GciRemoveOopFromNsc, page 406
GciRemoveOopsFromNsc, page 408
GciAddOopsToNsc

Add multiple OOPs to the unordered variables of a nonsequenceable collection.

Syntax

```c
void GciAddOopsToNsc(
    OopType theNsc,
    const OopType theOops[],
    int 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

OopType GciAddOopsToNsc_example(void)
{
    // return an IdentityBag containing the SmallIntegers with value 0..99

    enum { AddOopsToNsc_SIZE = 100 };  

    OopType oNsc = GciNewOop(OOP_CLASS_IDENTITY_BAG);

    OopType values[AddOopsToNsc_SIZE];  
    for (int i = 0; i < AddOopsToNsc_SIZE; i++) {
        values[i] = GciI32ToOop(i);
    }
    GciAddOopsToNsc(oNsc, values, AddOopsToNsc_SIZE);
    return oNsc;
}

See Also

GciAddOopToNsc, page 115  
GciNscIncludesOop, page 343  
GciRemoveOopFromNsc, page 406  
GciRemoveOopsFromNsc, page 408
GCI_ALIGN

(MACRO) Align an address to a word boundary.

Syntax

uintptr_t * GCI_ALIGN(argument)

Input Arguments

argument The pointer or integer to be aligned.

Result Value

The first multiple of 8 that is greater than or equal to the input argument.

Description

This macro can be used to round up a pointer or size to be a multiple of sizeof(OopType).

Provided for compatibility. New code should use the accessor functions in GciObjRepHdrSType (page 94).

See Also

GciMoreTraversal, page 293
GciNewOopUsingObjRep, page 338
GciTraverseObjs, page 510
GciAllocTravBuf

Allocate and initialize a new GciTravBufType structure.

Syntax

(GciTravBufType *) GciAllocTravBuf(
    size_t allocationSize);

Input Arguments

allocationSize The size of the traversal buffer.

Description

This function allocates and initializes a new GciTravBufType structure.

See Also

“The Traversal Buffer Type” on page 100
GciAlteredObjs

Find all exported or dirty objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.

Syntax

```
BoolType GciAlteredObjs(
    OopType theOops[],
    int * numOops);
```

Input Arguments

- `theOops`: An array that will contain the OOPs of the objects in the ExportedDirtyObjs or TrackedDirtyObjs sets.
- `numOops`: Pointer to the maximum number of OOPs that can be returned in this call, that is, the size (in OOPs) of the buffer specified by `theOops`.

Result Arguments

- `theOops`: The resulting array of OOPs of objects that are in either the ExportedDirtyObjs or TrackedDirtyObjs sets.
- `* numOops`: The number of actual OOPs in the result array `theOops`.

Return Value

The function result indicates whether all dirty objects have been returned. If the operation is not complete, `GciAlteredObjs` returns FALSE, and it is expected that the application will make repeated calls to this function until it returns TRUE, indicating that all of the dirty objects have been returned. If repeated calls are not made, then the unreturned objects persist in the list until the next time `GciAlteredObjs`, or another call that destructively accesses the ExportedDirtyObjs or TrackedDirtyObjs sets, is called.

Description

Typically, a GemStone C application program caches some database objects in its local object space, generally in the PureExportSet or if in a user action, in the user action's export
set (see `GciSaveObjs`). It may also track them by storing them in the `GciTrackedObjs` set (see `GciSaveAndTrackObjs`). After an abort or a successful commit, the user’s session is resynchronized with the most recent version of the database. The values of instance variables cached in 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.

This function returns a list of all objects that are in the `PureExportSet` and are “dirty”. An object is considered dirty (changed) under one or more of the following conditions:

- The object was changed by Smalltalk execution from this session.
- The object was changed by a call from this session to any GemBuilder function from within a user action.
- The object was changed by a call from this session to one or more of the following functions: `GciStorePaths`, `GciSymDictAtObjPut`, `GciSymDictAtPut`, `GciStrKeyValueDictAtObjPut`, or `GciStrKeyValueDictAtPut`.
- The object was read by this session, and after this session did a commit, begin, or abort transaction, the session now has visibility to changes to the object committed by another session.
- The object is persistent, and this session aborted its changes to the object, thus rolling back the Smalltalk in-memory state to the previously committed state.

Calls to `GciStore...` (other than `GciStorePaths`), `GciAppend...`, `GciReplace...`, and `GciCreate...` do not put the modified object into the set of dirty objects (unless the call is from within a user action). The assumption is that the client does not want the dirty set to include modifications that the client has explicitly made.

You must call `GciDirtyObjsInit` once after `GciLogin` before you can use `GciAlteredObjs`.

Note that `GciAlteredObjs` removes OOPs from the `ExportedDirtyObjs` set and `TrackedDirtyObjs` sets as it enumerates.

**See Also**

- `GciAbort`, page 114
- `GciCommit`, page 147
- `GciDirtyObjsInit`, page 176
- `GciReleaseAllOops`, page 399
- `GciReleaseOops`, page 402
- `GciSaveAndTrackObjs`, page 419
GciSaveGlobalObjs, page 421
GciSaveObjs, page 422
GciAppendBytes

Append bytes to a byte object.

Syntax

```c
void GciAppendBytes(
    OopType theObject,
    int64 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 Float or SmallFloat. Float and SmallFloat objects are of a fixed and unchangeable size.

See Also

GciAppendChars, page 125
GciAppendChars

Append a C string to a byte object.

Syntax

    void GciAppendChars(
        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 124
GciAppendOops

Append OOPs to the unnamed variables of a collection.

Syntax

```c
void GciAppendOops(
    OopType        theObject,
    int            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:

```c
GciStoreOops(theObject, GciFetchSize_(theObject)+1, theOops, numOops);
```

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

```c
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 114  
GciExecuteStr, page 195  
GciNbAbort, page 296  
GciNbBegin, page 297  
GciNbExecuteStr, page 308
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

```c
int GCI_BOOL_TO_OOP_example(void)
{
    int z = 0;
    int nonZ = 99;

    OopType Fa = GCI_BOOL_TO_OOP(z);
    // any non-zero argument will produce a result of OOP_TRUE
    OopType Tr = GCI_BOOL_TO_OOP(nonZ);
    // the following will always be true
    return Fa == OOP_FALSE && Tr == OOP_TRUE;
}
```

See Also

GciOopToBool, page 354
GciByteArrayToPointer

Given a result from GciPointerToByteArray, return a C pointer.

Syntax

void * GciByteArrayToPointer(
    OopType arg);

Input Arguments

arg A GemStone SmallInteger or ByteArray that was returned by GciPointerToByteArray.

Description

Given an argument that was the result of GciPointerToByteArray, this function returns the corresponding C pointer.

See Also

GciPointerToByteArray, page 383
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 signal 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

GcilnUserAction, page 279
GciCheckAuth

Gather the current authorizations for an array of database objects.

Syntax

```c
void GciCheckAuth(
    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 object security policy using the **GciPerform** 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**

```c
OopType GCI_CHR_TO_OOP_example(void)
{
    // return the OOP for the ASCII character ‘a’
    OopType theOop = GCI_CHR_TO_OOP(‘a’);
    return theOop;
}
```

**See Also**

GciOopToChr, page 359
GciClampedTrav

Traverse an array of objects, subject to clamps.

Syntax

```c
BoolType GciClampedTrav(
    const OopType * theOops,
    int numOops,
    GciClampedTravArgsSType *travArgs);
```

Input Arguments

- `theOops`: An array of OOPs representing the objects to traverse.
- `numOops`: The number of elements in `theOops`.
- `travArgs`: Pointer to an instance of `GciClampedTravArgsSType` 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.
  - `int 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.
  - `GciTravBufType * travBuff`: A pointer to the traversal buffer.
  - `int 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 PureExportSet or the user action’s export set (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 PureExportSet or the user action’s export set before it adds any OOPs to the traverse buffer.

Result Arguments

\[ travArgs \]

Pointer to an instance of \textbf{GciClampedTravArgsSType} containing the following result argument field:

\[ GciTravBufType \ast \text{travBuff} \]

The buffer for the results of the traversal. The first element placed in the buffer is the \text{actualBufferSize}, an 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 \textbf{GciObjRepSType}.

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 \textbf{GciMoreTraversal} (that is, an object report was constructed for each object, minus the special objects).

Description

The \textbf{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 510 for a detailed discussion of object traversal.

GemBuilder clamped traversal functions are used by the GemBuilder for Smalltalk implementation of object replication and are intended for similar sophisticated client applications.
See Also

GciMoreTraversal, page 293
GciSaveObjs, page 422
GciClampedTraverseObjs

Traverse an array of objects, subject to clamps.

This function is provided for compatibility with prior releases. New code should use GciClampedTrav.

Syntax

```c
BoolType GciClampedTraverseObjs(
    OopType               clampSpec,
    const OopType         theOops[ ],
    int                   numOops,
    GciTravBufType *      travBuff,
    int                   level );
```

Input Arguments

- `clampSpec` The OOP of the Smalltalk ClampSpecification to be used.
- `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*, an 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*. If a given object report represents a clamped object, the *valueBuffSize* of the report is zero. The *idxSize* of the report is filled in with the *varyingSize* for simple objects such as Array, String, IdentityBag, IdentitySet, and some kinds of ObjectDictionary. For details about the object information and object report structures, see the discussion beginning on page 94. If the report array would otherwise 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 Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

GemBuilder clamped traversal functions are intended primarily for GemStone internal use.
See Also

GciTraverseObjs, page 510
GciNbClampedTraverseObjs, page 299
GciNbTraverseObjs, page 328
GciClassMethodForClass

Compile a class method for a class.

Syntax

OopType GciClassMethodForClass(
    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 adds 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).

Return Value

Returns OOP_NIL, unless there were compiler warnings (such as variables declared but not used, etc.), in which case the return will be the OOP of a string containing the warning messages.

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 Programming Guide for GemStone/S 64 Bit for a list of the optimized selectors.

To remove a class method, use GciExecuteStr instead.

Example

```c
void GciClassMethodForClass_example(void)
{
    // Assumes the topaz code for GciFetchVaryingOop example
    // has been executed.

    OopType theClass = GciResolveSymbol("Component", OOP_NIL);
    OopType oCateg = GciNewString("instance creation");
    // method to create a new instance with a specified part number
    OopType oMethodSrc = GciNewString(
        "newWithNumber: aNum . | o | o := self new . o partNumber: aNum. ^ o");

    GciClassMethodForClass(oMethodSrc, theClass, oCateg, OOP_NIL);
    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        printf("error category "FMT_OID" number %d, %s\n",
                errInfo.category, errInfo.number, errInfo.message);
    }
}
```

See Also

GciInstMethodForClass, page 277
**GciClassNamedSize**

Find the number of named instance variables in a class.

**Syntax**

```c
int GciClassNamedSize(  
    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

```c
int namedSizeExample(void)
{
    // find the class named Employee in the current symbolList
    OopType empClass = GciResolveSymbol("Employee", OOP_NIL);
    if (empClass == OOP_NIL) {
        return -1;  // class not found or other error.
    }
    int numIvs = GciClassNamedSize(empClass);
    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        return -1; // error occurred
    }

    // return the number of named instance variables which will
    // be >= 0
    return numIvs;
}
```

See Also

GciIvNameToIdx, page 285
GciClearStack

Clear the Smalltalk call stack.

Syntax

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

Input Arguments

- `process`: The OOP of a GsProcess 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 GsProcess object, and raises an error. The GsProcess 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 shows how an application can handle an error and either continue or terminate Smalltalk execution.

```c
void clearStackExample(void)
{
    OopType result = GciExecuteStr(
        "| a | a := 10 + 10. nil halt . ^ a + 100",
        OOP_NIL/*use default symbolList for execution*/);
```
// halt method is expected to generate error number

RT_ERR_GENERIC_ERROR
GciErrSType errInfo;
if (! GciErr(&errInfo)) {
    printf("expected an error but none found\n");
    return;
}
if (errInfo.number == ERR_Halt) {
    // now continue the execution to finish the computation
    result = GciContinue(errInfo.context);
} else {
    // FMT_OID format string is defined in gci.ht
    printf("unexpected error category "FMT_OID" number %d, %s\n",
           errInfo.category, errInfo.number, errInfo.message);
    // terminate the execution
    GciClearStack(errInfo.context);
    return;
}
int val = GciOopToI32(result);
if (GciErr(&errInfo)) {
    printf("unexpected error category "FMT_OID" number %d, %s\n",
           errInfo.category, errInfo.number, errInfo.message);
} else {
    if (val != 120) {
        printf("Wrong answer = %d\n", val);
    } else {
        printf("result = %d\n", val);
    }
}

See Also

GciContinue, page 154
GciSoftBreak, page 441
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

void GciCommit_example(void)
{
    // Call GciCommit and see if there was an error
    if ( ! GciCommit()) {
        GciErrSType errInfo;
        if (GciErr(&errInfo)) {
            printf("commit failed with error %d , %s \n", 
                   errInfo.number, errInfo.message );
        } else {
            printf("commit failed due to transaction conflicts\n");
        }
    }
}
See Also

GciAbort, page 114
GCI_CHR_TO_OOP, page 134
GciNbAbort, page 296
GciNbCommit, page 301
GciCompileMethod

Compile a method.

Syntax

OopType GciCompileMethod(
    OopType source,
    OopType oclass,
    OopType category,
    OopType symbolList,
    OopType overrideSelector,
    int compileFlags,
    ushort environmentId );

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 adds 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).
overrideSelector If not nil, this is a string that is converted to a symbol and used in precedence to the selector pattern in the method source when installing the method in the method dictionary. Sending 'selector' to the resulting method will also reflect the overrideSelector argument.
compileFlags Compiler flags used for bootstrapping the Ruby environment.
environmentId The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.
Return Value

Returns OOP_NIL, unless there were compiler warnings (such as variables declared but not used, etc.), in which case the return will be the OOP of a string containing the warning messages.

Description

This function is used for compiling a method. Replaces both GciInstMethodForClass and GciClassMethodForClass, and adds the environmentId argument.

This function compiles a 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 Programming Guide for GemStone/S 64 Bit for a list of the optimized selectors.

To remove a method, use GciExecuteStr instead.

See Also

GciClassMethodForClass, page 141
GciInstMethodForClass, page 277
GciCompress

Compress the supplied data, which can be uncompressed with GciUncompress.

Syntax

```c
int GciCompress(
    char * dest,
    uint * destLen,
    const char * source,
    uint sourceLen);
```

Input Arguments

- `dest`: Pointer to the buffer intended to hold the resulting compressed data.
- `destLen`: Length, in bytes, of the buffer intended to hold the compressed data.
- `source`: Pointer to the source data to compress.
- `sourceLen`: Length, in bytes, of the source data.

Result Arguments

- `dest`: The resulting compressed data.

Return Value

GciCompress returns Z_OK (equal to 0) if the compression succeeded, or various error values if it failed; see the documentation for the compress function in the GNU zlib library at http://www.gzip.org.

Description

GciCompress passes the supplied inputs unchanged to the compress function in the GNU zlib library Version 1.2.3, and returns the result exactly as the GNU compress function returns it.
Example

#include <limits.h>

OopType compressByteArray(OopType byteArray) 
{
    // given an input ByteArray, return a new ByteArray with
    // the contents of the input compressed.

    if (!GciIsKindOfClass(byteArray, OOP_CLASS_BYTE_ARRAY) )
        return OOP_NIL; /* error: input arg is not a ByteArray */

    int64 inputSize = GciFetchSize_(byteArray);
    if (inputSize > INT_MAX) {
        return OOP_NIL; // GciCompress supports max 2G bytes input
    }

    int64 outputSize = inputSize;

    ByteType *inputBuffer = (ByteType*)malloc( inputSize);
    if (inputBuffer == NULL) {
        return OOP_NIL; // malloc failure
    }
    ByteType *outputBuffer = (ByteType*)malloc( outputSize);
    if (outputBuffer == NULL) {
        free(inputBuffer);
        return OOP_NIL; // malloc failure
    }

    OopType resultOop = OOP_NIL;

    int64 numRet = GciFetchBytes_(byteArray, 1/* start at first element */,
    inputBuffer, inputSize /* max bytes to fetch */ );
    if (numRet == inputSize) {
        uint compressedSize;
        int status = GciCompress( (char *)outputBuffer, &compressedSize,
            (char *) inputBuffer, inputSize);
        if (status == 0) {
            // compress ok
        }
    }
}
resultOop = GciNewByteObj(OOP_CLASS_BYTE_ARRAY,
    outputBuffer, (int64)compressedSize );
} else {
    // compress failed
}
} else {
    // error during FetchBytes
}
free(inputBuffer);
free(outputBuffer);
return resultOop;

See Also

GciUncompress, page 515
**GciContinue**

Continue code execution in GemStone after an error.

**Syntax**

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

**Input Arguments**

`process` The OOP of a GsProcess 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 32.

It may also be possible to continue Smalltalk execution if the virtual machine detects a nonfatal error during a `GciExecute`... 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 145.
See Also

GciClearStack, page 145
GciErr, page 189
GciExecute, page 191
GciNbContinue, page 302
GciNbExecute, page 306
GciContinueWith

Continue code execution in GemStone after an error.

Syntax

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

Input Arguments

process The OOP of a GsProcess 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 154
GciErr, page 189
GciExecute, page 191
GciNbContinueWith, page 303
GciNbExecute, page 306
GciPerformNoDebug, page 373
GciCreateByteObj

Create a new byte-format object.

Syntax

OopType GciCreateByteObj(
    OopType oclass,
    OopType objId,
    const ByteType * values,
    int64 numValues,
    int 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.
        If you are trying to create a Symbol or DoubleByteSymbol, objId must be OOP_ILLEGAL. You cannot use the result of GciGetFreeOop to create a type of Symbol object.
values  Array of instance variable values.
numValues Number of elements in values.
clusterId ID of the cluster bucket in which to place the object. If clusterId is 0, use the cluster bucket (System currentClusterId). Otherwise, clusterId must be a positive integer <= GciFetchSize_(OOP_ALL_CLUSTER_BUCKETS).
makePermanent Has no effect.

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`. For more about the semantics of such object identifiers, see the `GciGetFreeOop` function on page 260.

The object is created in temporary object space, and the garbage collector makes it permanent if the object is referenced, or becomes referenced, by another permanent object.

Values are stored into the object starting at the first named instance variable (if any) and continuing to the indexable (or NSC) instance variables if `oclass` is indexable or NSC. 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, `numValues` must not exceed the number of named instance variables in the class. If `numValues` is less than number of named instance variables, then the size of the newly-created object is the number of named instance variables and any instance variables beyond `numValues` are initialized to OOP_NIL.

For certain classes of byte format, namely DateTime, Float, LargePositiveInteger, and LargeNegativeInteger, additional size restrictions apply.

For an indexable object, if `numValues` is greater than zero and `values` is NULL, then the object is created of size `numValues`, and is initialized to logical size `numValues`. (This is equivalent to `new: aSize` for classes Array or String.)

If `GciCreateByteObj` is being used to create an instance of OOP_CLASS_FLOAT or OOP_CLASS_SMALL_FLOAT, then the correct number of `value` bytes must be supplied at the time of creation.

If you are trying to create a Symbol or DoubleByteSymbol, `objId` must be OOP_ILLEGAL.

See Also

- GciCreateOopObj, page 160
- GciGetFreeOop, page 260
- GciGetFreeOops, page 262
GciCreateOopObj

Create a new pointer-format object.

Syntax

OopType GciCreateOopObj(
    OopType oclass,
    OopType objId,
    const OopType * values,
    int numValues,
    int 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. If clusterId is 0, use the cluster bucket (System currentClusterId). Otherwise, clusterId must be a positive integer <= GciFetchSize_(OOP_ALL_CLUSTER_BUCKETS).
makePermanent Has no effect.

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. For more about the semantics of such object identifiers, see the GciGetFreeOop function on page 260.
The object is created in temporary object space, and the garbage collector makes it permanent if the object is referenced, or becomes referenced, by another permanent object.

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

Because it is illegal to create a forward reference to a Symbol, any GciCreate... call that creates a Symbol will fail if the client’s objId of the created object was already used as a forward reference.

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, numValues must not exceed the number of named instance variables in the class. If numValues is less than number of named instance variables and any 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 is created of size numValues, and is initialized to logical size numValues. (This is equivalent to new: aSize for classes Array or String.)

See Also

GciCreateByteObj, page 158
GciGetFreeOop, page 260
GciGetFreeOops, page 262
GciCTimeToDateTime

Convert a C date-time representation to the equivalent GemStone representation.

Syntax

_BoolType GciCTimeToDateTime(
    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 the equivalent C representation.

**Syntax**

```c
#include <time.h>

time_t GciDateTimeToCTime(
    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(
    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, 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

```c
void traceGciFunct(const char* gciFname)
{
    printf("trace gci call %s \n", gciFname);
}

void debugEstablishExample(void)
{
    GciDbgEstablish(traceGciFunct);  // enable tracing
    GciFetchSize_(OOP_CLASS_STRING);  // this call will be traced
    GciDbgEstablish(NULL);  // shut off tracing
}
```

See Also

GciDbgEstablishToFile, page 166
GciErr, page 189
GciDbgEstablishToFile

Write trace information for most GemBuilder functions to a file.

Syntax

```
BoolType GciDbgEstablishToFile(
    const char * fileName);
```

Input Arguments

`fileName` The file to which trace information is to be written.

Return Value

Returns TRUE if the file operation was successful.

Description

This function causes trace information for most GemBuilder functions to be written to a file. If the file already exists, it is opened in append mode. If `fileName` is NULL and tracing to a file is not currently active, trace information will be written to stdout.

Calling `GciDbgEstablishToFile` supersedes the effect of any previous calls to `GciDbgEstablish` or `GciDbgEstablishToFile`.

To terminate tracing to an active file, your program can use the following statement:

```
GciDbgEstablishToFile(NULL);
```

Alternatively, your program can call `GciShutdown`.

For details about the trace information generated, see `GciDbgEstablish`.

See Also

GciDbgEstablish, page 164
GciErr, page 189
**GciDbgLogString**

Pass a message to a trace function.

**Syntax**

```c
void GciDbgLogString(
    const char * message);
```

**Input Arguments**

*message*: A message to be passed to `GciDbgEstablish` or `GciDbgEstablishToFile`.

**Description**

If either `GciDbgEstablish` or `GciDbgEstablishToFile` has been called to activate tracing of GemBuilder calls, this function passes the argument to the trace function.

If tracing is not active, this function has no effect.

**See Also**

- `GciDbgEstablish`, page 164
- `GciDbgEstablishToFile`, page 166
GciDeclareAction

An alternative way to associate a C function with a Smalltalk user action.

NOTE
In previous GemStone/S 64 Bit releases, similar behavior was provided by the macro GCI_DECLARE_USER_ACTION.

Syntax

```c
void GciDeclareAction( 
    const char*    name, 
    void*           func, 
    int             nargs, 
    uint           flags, 
    BoolType   errorIfDuplicate );
```

Input Arguments

- **name**: The user action name (a case-insensitive, null-terminated string).
- **func**: A pointer to the C user action function.
- **nargs**: The number of arguments in the C function.
- **flags**: Flags that are rarely used. Mainly for internal use.
- **errorIfDuplicate**: If True, return an error if there is already a user action with the specified name. If False, leave the existing user action in place and ignore the current call.

Description

This function associates a user action name (declared in Smalltalk) with a user-written C function. GciDeclareAction allows you to declare a user action by passing each field of the user action structure to the function as a separate argument. Because the user action structure is encapsulated within the function itself, there’s no need to explicitly allocate and free memory, as is required with GciInstallUserAction (which uses the data structure defined by GciUserActionSType).
See Also

“The User Action Information Structure” on page 99
“GciInstallUserAction” on page 276
GciDecodeOopArray

Decode an OOP array that was previously run-length encoded.

Syntax

```c
int GciDecodeOopArray(
    OopType * encodedOopArray,
    const int numEncodedOops,
    OopType * decodedOopArray,
    const int decodedOopArraySize);
```

Input Arguments

- `encodedOopArray`: An OOP array that was encoded by a call to `GciEncodeOopArray`.
- `numEncodedOops`: The number of OOPs in `encodedOopArray`.
- `decodedOopArraySize`: The maximum number of OOPs in `decodedOopArray`.

Result Arguments

- `decodedOopArray`: The decoded OOP array that had been run-length encoded.

Return Value

Returns the number of OOPs placed in `decodedOopArray`.

Description

This function decodes the OOPs in `encodedOopArray` that were run-length encoded using `GciEncodeOopArray` and places the result in `decodedOopArray`.

The `decodedOopArraySize` must be large enough to hold all decoded OOPs. If it is not, no decode is performed and `decodedOopArraySize` is set to -1.
See Also

- GciFetchNumEncodedOops, page 224
- GciEnableFreeOopEncoding, page 183
- GciEncodeOopArray, page 187
- GciGetFreeOopsEncoded, page 264
GciDecSharedCounter

Decrement the value of a shared counter.

Syntax

```c
BoolType GciDecSharedCounter(
    int64_t counterIdx,
    int64_t * value,
    int64_t * floor);
```

Input Arguments

- `counterIdx`: The offset into the shared counters array of the value to decrement.
- `value`: Pointer to a value that indicates how much to decrement the shared counter by.
- `floor`: The minimum possible value for the shared counter. The counter cannot be decremented below this value. If `floor` is NULL, then a `floor` value of INT_MIN (-2147483647) will be used.

Result Arguments

- `value`: Pointer to a value that indicates the new value of the shared counter, after the decrement.

Return Value

Returns a C Boolean value indicating if the shared counter was successfully decremented by the given amount. Returns TRUE if successful, FALSE if an error occurred.

Description

This function decrements the value of a particular shared counter by a specified amount. The shared counter is specified by index. The value of this shared counter cannot be decremented to a value lower than `floor`.

This function is not supported for remote GCI interfaces, and will always return FALSE.
See Also

GciFetchNumSharedCounters, page 225
GciIncSharedCounter, page 271
GciSetSharedCounter, page 437
GciReadSharedCounter, page 394
GciReadSharedCounterNoLock, page 395
GciFetchSharedCounterValuesNoLock, page 244
GciDirtyExportedObjs

Find all objects in the ExportedDirtyObjs set.

Syntax

```c
BoolType GciDirtyExportedObjs(
    OopType theOops[],
    int * numOops);
```

Input Arguments

- `numOops` The maximum number of objects that can be put into theOops buffer.

Result Arguments

- `theOops` An array of the dirty exported objects found.
- `numOops` The number of dirty exported 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

This function returns a list of all objects that are in the ExportedDirtyObjs set, which includes all objects in the PureExportSet that have been made “dirty” since the ExportedDirtyObjs set was last initialized or retrieved using GciDirtyAlteredObjs, GciDirtyExportedObjs, GciDirtyObjsInit, GciDirtySaveObjs, or GciTrackedObjsFetchAllDirty. Object are added to the PureExportSet using
GciSaveObjs or by other functions that invoke GciSaveObjs. An object is considered dirty (changed) under one or more of the following conditions:

- The object was changed by Smalltalk execution from this session.
- The object was changed by a call from this session to any GemBuilder function from within a user action.
- The object was changed by a call from this session to one or more of the following functions: GciStorePaths, GciSymDictAtObjPut, GciSymDictAtPut, GciStrKeyValDictAtObjPut, or GciStrKeyValDictAtPut.
- The object was read by this session, and after this session did a commit, begin, or abort transaction, the session now has visibility to changes to the object committed by another session.
- The object is persistent, and this session aborted its changes to the object, thus rolling back the Smalltalk in-memory state to the previously committed state.

Calls to GciStore... (other than GciStorePaths), GciAppend..., GciReplace..., and GciCreate... do not put the modified object into the set of dirty objects (unless the call is from within a user action). The assumption is that the client does not want the dirty set to include modifications that the client has explicitly made.

The function GciDirtyObjsInit must be executed once after GciLogin before this function can be called, because it depends upon GemStone’s set of dirty objects.

The user is expected to call this function repeatedly while it returns FALSE, until it finally returns TRUE. When this function returns TRUE, it first clears the set of dirty objects.

Note that GciDirtyExportedObjs removes OOPs from the ExportedDirtyObjs set as they are enumerated.

See Also

“Garbage Collection” on page 49
“GciDirtyObjsInit” on page 176
“GciDirtySaveObjs” on page 178
“GciDirtyTrackedObjs” on page 180
“GciTrackedObjsFetchAllDirty” on page 507
“GciHiddenSetIncludesOop” on page 268
“GciReleaseAllGlobalOops” on page 398
“GciSaveGlobalObjs” on page 421
“GciSaveObjs” on page 422
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, the ExportedDirtyObjects, when it is needed. Once initialized, GemStone tracks dirty objects until GciLogout is executed.

GciDirtyObjsInit must be called once after GciLogin before GciDirtyExportedObjs, GciDirtySaveObjs, or GciTrackedObjsFetchAllDirty 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:

- The object was changed by Smalltalk execution from this session.
- The object was changed by a call from this session to any GemBuilder function from within a user action.
- The object was changed by a call from this session to one or more of the following functions: GciStorePaths, GciSymDictAtObjPut, GciSymDictAtPut, GciStrKeyValDictAtObjPut, or GciStrKeyValDictAtPut.
- The object was read by this session, and after this session did a commit, begin, or abort transaction, the session now has visibility to changes to the object committed by another session.
- The object is persistent, and this session aborted its changes to the object, thus rolling back the Smalltalk in-memory state to the previously committed state.

See Also

GciDirtyExportedObjs, page 174
GciDirtySaveObjs, page 178
GciTrackedObjsFetchAllDirty, page 507
GciHiddenSetIncludesOop, page 268
GciDirtySaveObjs

Find all exported or tracked objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.

Syntax

BoolType GciDirtySaveObjs(
    OopType     theOops[],
    int *       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 that are in the ExportedDirtyObjs or TrackedDirtyObjs sets. The ExportedDirtyObjs set includes all objects in PureExportSet that have been made “dirty” since the ExportedDirtyObjs set was last reset, and the TrackedDirtyObjs set includes all objects in the GciTrackedObjs set that have been made “dirty” since the TrackedDirtyObjs set was last reset.

The ExportedDirtyObjs set is initialized by GciDirtyObjsInit; it is cleared by calls to GciDirtyAlteredObjs, GciDirtyExportedObjs, GciDirtySaveObjs (this function), or GciTrackedObjsFetchAllDirty. The TrackedDirtyObjs set is initialized by
GciTrackedObjsInit and cleared by calls to GciDirtyAlteredObjs, GciDirtySaveObjs (this function), GciDirtyTrackedObjs, or GciTrackedObjsFetchAllDirty.

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

- The object was changed by Smalltalk execution from this session.
- The object was changed by a call from this session to any GemBuilder function from within a user action.
- The object was changed by a call from this session to one or more of the following functions: GciStorePaths, GciSymDictAtObjPut, GciSymDictAtPut, GciStrKeyValueDictAtObjPut, or GciStrKeyValueDictAtPut.
- The object was read by this session, and after this session did a commit, begin, or abort transaction, the session now has visibility to changes to the object committed by another session.
- The object is persistent, and this session aborted its changes to the object, thus rolling back the Smalltalk in-memory state to the previously committed state.

Calls to GciStore... (other than GciStorePaths), GciAppend..., GciReplace..., and GciCreate... do not put the modified object into the set of dirty objects (unless the call is from within a user action). The assumption is that the client does not want the dirty set to include modifications that the client has explicitly made.

GciDirtyObjsInit must be called once after GciLogin before GciDirtySaveObjs can be 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.

For details about the PureExportSet, see GciSaveObjs. For details about the GciTrackedObjs set, see GciSaveAndTrackObjs.

Note that GciDirtySaveObjs removes OOPs from the ExportedDirtyObjs and TrackedDirtyObjs sets.

See Also

“Garbage Collection” on page 49
“GciDirtyExportedObjs” on page 174
“GciDirtyObjsInit” on page 176
“GciDirtyTrackedObjs” on page 180
“GciTrackedObjsFetchAllDirty” on page 507
“GciSaveObjs” on page 422
GciDirtyTrackedObjs

Find all tracked objects that have changed and are therefore in the TrackedDirtyObjs set.

Syntax

```
BoolType GciDirtyTrackedObjs(
    OopType          theOops[ ],
    int *            numOops);
```

Input Arguments

- `numOops` The maximum number of objects that can be put into `theOops` buffer.

Result Arguments

- `theOops` An array of the dirty tracked objects found.
- `numOops` The number of dirty tracked objects found.

Return Value

This function returns a C Boolean value indicating whether or not the complete set of dirty tracked 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

This function returns a list of all objects that are in the TrackedDirtyObjs set, which includes all objects that are in the GciTrackedObj set and have been made “dirty” since the GciTrackedObj set was initialized or cleared. Functions that initialize or remove objects from the TrackedDirtyObj set are `GciDirtyAlteredObjs`, `GciDirtySaveObjs`, `GciDirtyTrackedObjs` (this function), `GciTrackedObjFetchAllDirty` and `GciTrackedObjInit`. 
An object is considered dirty (changed) under one or more of the following conditions:

- The object was changed by Smalltalk execution from this session.
- The object was changed by a call from this session to any GemBuilder function from within a user action.
- The object was changed by a call from this session to one or more of the following functions: GciStorePaths, GciSymDictAtObjPut, GciSymDictAtPut, GciStrKeyValDictAtObjPut, or GciStrKeyValDictAtPut.
- The object was read by this session, and after this session did a commit, begin, or abort transaction, the session now has visibility to changes to the object committed by another session.
- The object is persistent, and this session aborted its changes to the object, thus rolling back the Smalltalk in-memory state to the previously committed state.

Calls to GciStore... (other than GciStorePaths), GciAppend..., GciReplace..., and GciCreate... do not put the modified object into the set of dirty objects (unless the call is from within a user action). The assumption is that the client does not want the dirty set to include modifications that the client has explicitly made.

This function may only be called after GciTrackedObjsInit has been executed, because it depends upon GemStone’s set of tracked objects. The user is expected to call this function repeatedly while it returns FALSE, until it finally returns TRUE. When this function returns TRUE, it first clears the set of dirty objects.

Note that GciDirtyTrackedObjs removes OOPs from the TrackedDirtyObjs set.

See Also

“Garbage Collection” on page 49
“GciDirtySaveObjs” on page 178
“GciHiddenSetIncludesOop” on page 268
“GciReleaseAllTrackedOops” on page 400
“GciSaveAndTrackObjs” on page 419
“GciTrackedObjsFetchAllDirty” on page 507
“GciTrackedObjsInit” on page 509
Gci_doubleToSmallDouble

Convert a C double to a SmallDouble object.

Syntax

OopType Gci_doubleToSmallDouble(
    double aFloat);

Return Value

Returns the OOP of the GemStone SmallDouble object that corresponds to the C value. If the C value is not representable as a GemStone SmallDouble, return OOP_ILLEGAL.

Description

This function translates a C double into the equivalent GemStone SmallDouble object.

See Also

GciFltToOop, page 258
GciEnableFreeOopEncoding

Enable run-length encoding of free OOPs.

Syntax

void GciEnableFreeOopEncoding( )

Description

This function enables run-length encoding of free OOPs sent between the Gem and the
GemBuilder client. This function increases CPU consumption on both the client and the
Gem, and decreases the number of bytes passed on the network.

See Also

GciDecodeOopArray, page 170
GciEncodeOopArray, page 187
GciFetchNumEncodedOops, page 224
GciGetFreeOopsEncoded, page 264
GciEnableFullCompression

Enable full compression between the client and the RPC version of GemBuilder.

Syntax

void GciEnableFullCompression( )

Description

This function enables full compression (in both directions) between the client and GciRpc (the “remote procedure call” version of GemBuilder). This function has no effect for linked sessions.

See Also

GciIsRemote, page 282
GciEnableSignaledErrors

Establish or remove GemBuilder visibility to signalled errors from GemStone.

Syntax

```c
BoolType GciEnableSignaledErrors(
    BoolType(newState);
```

Input Arguments

`newState` The new state of signalled error visibility: TRUE for visible.

Return Value

This function returns TRUE if signalled 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 GciLongImp 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 189
GciPollForSignal, page 384
GciEncodeOopArray

Encode an array of OOPs, using run-length encoding.

Syntax

```c
int GciEncodeOopArray(
    OopType * oopArray,
    const int numOops,
    OopType * encodedOopArray,
    BoolType needsSorting);
```

Input Arguments

- `oopArray`: An OOP array to be encoded.
- `numOops`: The number of OOPs in `oopArray`.
- `needsSorting`: If `oopArray` is known to be in ascending order, set this to FALSE; otherwise set it to TRUE.

Result Arguments

- `encodedOopArray`: The encoded OOP array.

Return Value

Returns the number of elements in the encoded array. Returns -1 indicating an error if the input array was found to be out of sequence and `needsSorting` was set to FALSE.

Description

This function encodes the OOPs in `oopArray` using run-length encoding and places the result in `encodedOopArray`. Both `oopArray` and `encodedOopArray` must have the size `numOops`. 
See Also

GciDecodeOopArray, page 170
GciEnableFreeOopEncoding, page 183
GciFetchNumEncodedOops, page 224
GciGetFreeOopsEncoded, page 264
GciErr

Prepare a report describing the most recent GemBuilder error.

Syntax

```
BoolType GciErr(
    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.

If the result is TRUE, all objects referenced from `errorReport` have been added to the PureExportSet, unless the error occurred during a `GciStoreTravDoTravRefs_`, in which case all objects referenced from `errorReport` have been added to the ReferencedSet rather than the PureExportSet.

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 386 for information about using general-purpose error traps in GemBuilder. The section “The Error Report Structure” on page 90 describes the C structure for error reports.
See Also

GciClearStack, page 145
GciContinue, page 154
GciExecute, page 191
GciPopErrJump, page 386
GciExecute

Execute a Smalltalk expression contained in a String object.

Syntax

OopType GciExecute(
    OopType source,
    OopType symbolList);

OopType GciExecute_(
    OopType source,
    OopType symbolList,
    ushort environmentId);

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).

environmentId
The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 32.

**Example**

```c
void executeExample(void)
{
    OopType oString = GciNewString(" ^ 3 + 4 ");

    OopType result = GciExecute(oString, OOP_NIL);
    if (result == OOP_NIL) {
        printf("error from execution\n");
    } else {
        BoolType conversionErr = FALSE;
        int val = GciOopToI32_(result, &conversionErr);
        if (conversionErr) {
            printf("Error converting result to C int\n");
        } else {
            printf("result = %d\n", val);
        }
    }
}
```

**See Also**

- **GciContinue**, page 154
- **GciErr**, page 189
- **GciExecuteFromContext**, page 193
- **GciExecuteStr**, page 195
- **GciExecuteStrFromContext**, page 198
- **GciNbContinue**, page 302
- **GciNbExecute**, page 306
- **GciNbExecuteStr**, page 308
- **GciNbExecuteStrFromContext**, page 310
GciExecuteFromContext

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

Syntax

```c
OopType GciExecuteFromContext(
    OopType  source,
    OopType  contextObject,
    OopType  symbolList);
```

```c
OopType GciExecuteFromContext_(
    OopType  source,
    OopType  contextObject,
    OopType  symbolList,
    ushort   environmentId);
```

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).
- **environmentId**: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 32.

See Also

"GciContinue" on page 154
"GciErr" on page 189
"GciExecute" on page 191
"GciExecuteStr" on page 195
"GciExecuteStrFromContext" on page 198
"GciNbContinue" on page 302
"GciNbExecute" on page 306
"GciNbExecuteStr" on page 308
"GciNbExecuteStrFromContext" on page 310
**GciExecuteStr**

Execute a Smalltalk expression contained in a C string.

**Syntax**

```c
OopType GciExecuteStr(
    const char source[],
    OopType symbolList);
```

```c
OopType GciExecuteStr_(
    const char source[],
    OopType symbolList,
    ushort environmentId);
```

**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`).
- `environmentId`: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “`environmentId`” on page 101.

**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 32.

**Example**

```c
void executeStrExample(void)
{
    // get the symbolList for UserProfile named ‘romeo’
    OopType symbolList = GciExecuteStr(
        "(AllUsers userWithId: ‘romeo’) symbolList", OOP_NIL);

    // get the value associated with key “nativeLanguage” in that symbolList ; values expected to be a kind of String
    OopType lang = GciExecuteStr("nativeLanguage", symbolList);

    // fetch characters of the String
    char buf[1024];
    GciFetchChars_(lang, 1, buf, sizeof(buf));

    GciErrSType errInfo;
    if ( GciErr(&errInfo)) {
        // FMT_OID format string is defined in gci.ht
        printf("unexpected error category “FMT_OID” number %d, %s\n",
            errInfo.category, errInfo.number, errInfo.message);
    } else {
        printf("nativeLanguage is %s \n", buf);
    }
}
```

**See Also**

- `GciContinue`, page 154
- `GciErr`, page 189
- `GciExecute`, page 191
- `GciExecuteFromContext`, page 193
- `GciExecuteStrFromContext`, page 198
- `GciNbContinue`, page 302
- `GciNbExecute`, page 306
GciNbExecuteStr, page 308
GciNbExecuteStrFromContext, page 310
GciExecuteStrFromContext

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

Syntax

OopType GciExecuteStrFromContext(
    const char source[],
    OopType contextObject,
    OopType symbolList);

OopType GciExecuteStrFromContext_(
    const char source[],
    OopType contextObject,
    OopType symbolList,
    ushort environmentId);

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).

environmentId The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 32.

Example

```c
void executeFromContextExample(void)
{
    // get the Association with key UserProfileSet in dictionary
    Globals
        OopType oAssoc = GciExecuteStr("Globals associationAt:
            UserProfileSet",
            OOP_NIL);

        OopType oResult = GciExecuteStrFromContext(" ^ value ", oAssoc,
            OOP_NIL);

        if (oResult != OOP_CLASS_USERPROFILE_SET) {
            printf("unexpected result"FMT_OID”\n", oResult);
        }
}
```

See Also

GciContinue, page 154
GciErr, page 189
GciExecute, page 191
GciExecuteFromContext, page 193
GciExecuteStr, page 195
GciNbContinue, page 302
GciNbExecute, page 306
GciNbExecuteStr, page 308
GciNbExecuteStrFromContext, page 310
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

```c
BoolType GciExecuteStrTrav(
    const char    source[],
    OopType       contextObject,
    OopType       symbolList,
    GciClampedTravArgsSType *travArgs);

BoolType GciExecuteStrTrav_(
    const char    source[],
    OopType       contextObject,
    OopType       symbolList,
    GciClampedTravArgsSType *travArgs,
    ushort        environmentId);
```

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 an instance of `GciClampedTravArgsSType` 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.
  - **int 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.

int 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 PureExportSet or the user action’s export set before it adds any OOPs to the traverse buffer.

environmentId
The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

Result Arguments

travArgs
Pointer to an instance of GciClampedTravArgsSType 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, an 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 195
- `GciMoreTraversal`, page 293
- `GciPerformTrav`, page 377
GciFetchByte

Fetch one byte from an indexed byte object.

Syntax

```
ByteType GciFetchByte(
    OopType theObject,
    int64 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

```
void fetchByteExample(void)
{
    OopType oString = GciNewString("abc");

    ByteType theChar = GciFetchByte(oString, 2);
    if (theChar != 'b') {
        printf("unexpected result \%d \n", theChar);
    }
}
```
See Also

GciFetchBytes, page 206
GciStoreByte, page 445
GciStoreBytes, page 447
GciFetchBytes_

Fetch multiple bytes from an indexed byte object.

Syntax

```c
int64 GciFetchBytes_(
    OopType theObject,
    int64 startIndex,
    ByteType theBytes[],
    int64 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

NOTE
In previous GemStone/S 64 Bit releases, this function was named `GciFetchBytes` (without the underscore). Customers must ensure that the variables that receive this function’s result are large enough to accommodate an int64 value.
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 the object to be a Float or SmallFloat, 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. (For more about byte swizzling, see page 29.)

**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.

```c
void displayByteObject(OopType oObject)
{
    enum { BUF_SIZE = 5000 };  // Buffer size
    char displayBuff[BUF_SIZE];  // Buffer
    BoolType done = FALSE;
    int idx = 1;
    while (! done) {
        int64 numRet = GciFetchBytes_(oObject, idx,
            (ByteType*)displayBuff, BUF_SIZE - 1);
        if (numRet == 0) {
            done = TRUE;  // Hit end of object or error
            GciErrSType errInfo;
            if (GciErr(&errInfo)) {
                printf("unexpected error category "\n
                    errInfo.category, errInfo.number, errInfo.message);
            }
        } else {
            displayBuff[numRet] = '\0';
            printf("\n", displayBuff);
            idx += numRet;
        }
    }
}
```
See Also

GciFetchByte, page 204
GciFetchObjInfo, page 229
GciStoreByte, page 445
GciStoreBytes, page 447
GciFetchChars_

Fetch multiple ASCII characters from an indexed byte object.

Syntax

```c
int64 GciFetchChars_(
    OopType theObject,
    int64 startIndex,
    char * cString,
    int64 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

**NOTE**

In previous GemStone/S 64 Bit releases, this function was named GciFetchChars (without the underscore). Customers must ensure that the variables that receive this function’s result are large enough to accommodate an int64 value.

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
\texttt{cString} are undefined.

\textbf{See Also}

GciFetchBytes_, page 206
GciFetchClass

Fetch the class of an object.

Syntax

OopType GciFetchClass(
    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, SmallDouble, or UndefinedObject.
Example

#include <stdlib.h>

void fetchClassExample(void)
{
    // random double to Oop conversion producing a Float or SmallFloat
    double rand = drand48() * 1.0e38;
    OopType oFltObj = GciFltToOop(rand);

    OopType oClass = GciFetchClass(oFltObj);
    const char* kind;
    if (oClass == OOP_CLASS_SMALL_DOUBLE) {
        kind = "SmallDouble";
    } else if (oClass == OOP_CLASS_FLOAT) {
        kind = "Float";
    } else {
        kind = "Unexpected";
    }
    printf("result is a %s, class oop = \"FMT_OID\"\n", kind, oClass);
}

See Also

GciFetchNamedSize, page 222
GciFetchObjImpl, page 228
GciFetchSize_, page 246
GciFetchVaryingSize_, page 253
GciFetchDateTime

Convert the contents of a DateTime object and place the results in a C structure.

Syntax

```c
void GciFetchDateTime(
    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.
GciFetchDynamicIv

Fetch the OOP of one of an object’s dynamic instance variables.

Syntax

OopType GciFetchDynamicIv(
    OopType theObject,
    OopType aSymbol);

Input Arguments

theObject The OOP of the GemStone object.

aSymbol Specifies the dynamic instance variable to fetch.

Return Value

Returns the OOP of the specified dynamic instance variable. If no such dynamic instance variable exists in the object, this function returns OOP_NIL.

Description

This function fetches the contents of an object’s dynamic instance variable, as specified by aSymbol.

See Also

GciFetchDynamicIvs, page 215
GciStoreDynamicIv, page 453
GciFetchDynamicIvs

Fetches the OOPs of one or more of an object’s dynamic instance variables.

Syntax

int GciFetchDynamicIvs(
    OopType       theObject,
    OopType *     buf,
    int           numOops);

Input Arguments

theObject    The OOP of the source GemStone object.
numOops     The maximum number of elements to return.

Result Arguments

buf          C pointer to the buffer that will contain the object’s dynamic
             instance variables.

Return Value

Returns the number of OOPs fetched. (This may be less than numOops, depending upon the
size of theObject.)

Description

The number of dynamic instance variable pairs returned is (function result / 2). To obtain
all dynamic instance variables in one call, use a buffer.

See Also

GciFetchDynamicIv, page 214
GciStoreDynamicIv, page 453
**GciFetchNamedOop**

Fetch the OOP of one of an object’s named instance variables.

**Syntax**

```c
OopType GciFetchNamedOop(
    OopType theObject,
    int 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 248.

```c
void fetchNamedOopExample(void)
{
    // C constants to match Smalltalk class definition
    enum { COMPONENT_OFF_PARTNUMBER = 1,
        COMPONENT_OFF_NAME       = 2,
        COMPONENT_OFF_COST       = 3 };

    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr("AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        // error during execution or detect found nothing
        return;
    }

    // fetch the name instance variable of aComponent
    OopType oName = GciFetchNamedOop(aComponent, COMPONENT_OFF_NAME);

    // fetch name instance variable without fixing its offset at C compile time
    int ivOffset = GciIvNameToIdx(GciFetchClass(aComponent), "name");
    oName = GciFetchNamedOop(aComponent, ivOffset);
}
```

See Also

- `GciFetchNamedOops`, page 219
- `GciFetchVaryingOop`, page 248
- `GciFetchVaryingOops`, page 251
- `GciIvNameToIdx`, page 285
- `GciStoreIdxOop`, page 454
- `GciStoreIdxOops`, page 456
GciStoreNamedOop, page 459
GciStoreNamedOops, page 462
**GcgiFetchNamedOops**

Fetch the OOPs of one or more of an object’s named instance variables.

**Syntax**

```c
int GcgiFetchNamed Oops(
    OopType theObject,
    int startIndex,
    OopType theOops[],
    int 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 248.

```c
void fetchNamedOops_example(void)
{
    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr(“AllComponents detect:[i|i partNumber = 1234]”, OOP_NIL);
    if (aComponent == OOP_NIL) {
        // execution error, or detect: found nothing
        return;
    }

    // fetch name instance variables without knowing offset at C
    compile time
    int namedSize = GciFetchNamedSize(aComponent);
    if (namedSize == 0) {
        // error during fetch
        return;
    }
    OopType *oBuffer = (OopType*) malloc( sizeof(OopType) * namedSize);
    if (oBuffer != NULL) {
        int numRet = GciFetchNamedOops(aComponent, 1, oBuffer, namedSize);
        if (numRet != namedSize) {
            // error during fetch
        } else {
            // do something with contents of oBuffer
        }
        free(oBuffer);
    } else {
        // malloc failure
    }
}
```
See Also

GciFetchNamedOop, page 216
GciFetchVaryingOop, page 248
GciIvNameToldx, page 285
GciStoreIdxOop, page 454
GciStoreNamedOop, page 459
GciFetchNamedSize

Fetch the number of named instance variables in an object.

Syntax

\[
\text{int GciFetchNamedSize(} \\
\text{OopType theObject);}\\n\]

Input Arguments

\text{theObject} \quad \text{The OOP of the specified object.}

Return Value

Returns the number of named instance variables in \text{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 219.
GciFetchNameOfClass

Fetch the class name object for a given class.

Syntax

OopType GciFetchNameOfClass(
    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.
### GciFetchNumEncodedOops

Obtain the size of an encoded OOP array.

#### Syntax

```c
int GciFetchNumEncodedOops(
    OopType * encodedOopArray,
    const int numEncodedOops);
```

#### Input Arguments

- `encodedOopArray` An OOP array that was encoded by a call to `GciEncodeOopArray`.

#### Result Arguments

- `numEncodedOops` The number of OOPs in `encodedOopArray`.

#### Return Value

Returns the number of OOPs that will be decoded by a call to `GciDecodeOopArray`.

#### Description

This function returns the total number of OOPs in an OOP array that was encoded by a call to `GciEncodeOopArray`.

#### See Also

- `GciDecodeOopArray`, page 170
- `GciEnableFreeOopEncoding`, page 183
- `GciEncodeOopArray`, page 187
- `GciGetFreeOopsEncoded`, page 264
**GciFetchNumSharedCounters**

Obtain the number of shared counters available on the shared page cache used by this session.

**Syntax**

```c
int GciFetchNumSharedCounters();
```

**Return Value**

Returns the number of shared counters available on the shared page cache used by this session, or -1 if the session is not logged in.

**Description**

This function returns the total number of shared counters available on the shared page cache used by this session.

Not supported for remote GCI interfaces.

**See Also**

- GciDecSharedCounter, page 172
- GciIncSharedCounter, page 271
- GciSetSharedCounter, page 437
- GciReadSharedCounter, page 394
- GciReadSharedCounterNoLock, page 395
- GciFetchSharedCounterValuesNoLock, page 244
**GciFetchObjectInfo**

Fetch information and values from an object.

**Syntax**

```c
BoolType GciFetchObjInfo(
    OopType theObject,
    GciFetchObjInfoArgsSType *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:

  - `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`.

  - `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/8`. If greater than zero, and if a Float or BinaryFloat is being fetched, it must be large enough to fetch the complete object.

  - `retrievalFlags` 
    If `(retrievalFlags & GCI_RETRIEVE_EXPORT) != 0` then if `theObject` is non-special, `theObject` is automatically added to the PureExportSet or the user action’s export set (see the `GciSaveObjs` function).

**Result Arguments**

- `args` 
  Pointer to an instance of `GciFetchObjInfoArgsSType` with the following result argument fields:
GciObjInfoSType *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.

int64 numReturned
    Number of logical elements (bytes or OOPs) returned in buffer. Remember that the size of (OopType) is 8 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 234
GciFetchBytes, page 206
GciFetchObjInfo, page 229
GciSaveObjs, page 422
GciFetchObjImpl

Fetch the implementation of an object.

Syntax

```c
int GciFetchObjImpl(
    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 35.

See Also

- GciFetchClass, page 211
- GciFetchNamedSize, page 222
- GciFetchSize_, page 246
- GciFetchVaryingSize_, page 253
GciFetchObjInfo

Fetch information and values from an object.

Syntax

```c
BoolType GciFetchObjInfo(
    OopType theObject,
    int64 startIndex,
    int64 bufSize,
    GciObjInfoSType * info,
    ByteType * buffer,
    int64 * 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 `Float` and `SmallFloat` objects. (For more about byte swizzling, see page 29.) If `theObject` is a `Float` or `SmallFloat`, 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(
    OopType theObject,
    int64 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 248.

```c
void fetchOop_example(void)
{
    // C constant to match Smalltalk class definition
    enum { COMPONENT_OFF_NAME = 2 };

    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:([i|i partNumber = 1234])", OOP_NIL);
    if (aComponent == OOP_NIL) {
        // error during execution, select: found nothing
        return ;
    }

    // Two ways to fetch the name instance variable of aComponent */
    OopType oName = GciFetchOop(aComponent, COMPONENT_OFF_NAME);
    oName = GciFetchNamedOop(aComponent, COMPONENT_OFF_NAME);

    // Fetch the 3rd element of aComponent’s partsList,
    // without knowing exactly how many named instance variables
    // exist.
    int namedSize = GciFetchNamedSize(aComponent);
    if (namedSize == 0) {
        // error during fetch
        return ;
    }
    OopType aSubComponent = GciFetchOop(aComponent, namedSize + 3);

    // alternate way to Fetch the 3rd element of aComponent’s
    // partsList
    aSubComponent = GciFetchVaryingOop(aComponent, 3);
}
```
See Also

GciFetchOops, page 234
GciStoreOop, page 465
GciStoreOops, page 468
### GciFetchOops

**Description**

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

**Syntax**

```c
int GciFetchOops(
    OopType theObject,
    int64 startIndex,
    OopType theOops[],
    int 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 greater than the size of the `theObject`, 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.
- `numOops`: 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 unnamed 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 248.

```c
void fetchOops_example(void)
{
    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr("AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        // error during execution, or detect: found nothing
        return;
    }

    enum { BUF_SIZE = 60 };  
    OopType oBuf[BUF_SIZE];

    int namedSize = GciFetchNamedSize(aComponent);
    if (namedSize == 0 || namedSize > 50) {
        // error during fetch, or too many named instVars for buffer
        return;
    }

    // Two ways to fetch first 5 elements of aComponent’s partsList
    GciFetchOops(aComponent, namedSize + 1, oBuf, 5);
    GciFetchVaryingOops(aComponent, 1, oBuf, 5);

    // Fetch the named instance variables PLUS
    // the first 5 elements of partsList
    GciFetchOops(aComponent, 1, oBuf, namedSize + 5);
    // oBuf[0..namedSize-1] are named instVar values,
    // oBuf[namedSize] is first varying instVar value
}
```

See Also

GciFetchOop, page 231
GciFetchVaryingOop, page 248
GciStoreOop, page 465
GciStoreOops, page 468
GciFetchPaths

Fetch selected multiple OOPs from an object tree.

Syntax

```
BoolType GciFetchPaths(
    const OopType   theOops[],
    int             numOops,
    const int       paths[],
    const int       pathSizes[],
    int             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 53.

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).

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:

\[
\text{results[ ((i-1) * numPaths) + (j-1) ]}
\]

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**

```c
void fetchPath1(void)
{
    enum { path_size = 5 };  
    int    aPath[path_size]; /* the path itself */
    int    aSize = path_size; /* the size of the path */

    for (int j = 0; j < path_size; j++) {
        aPath[j] = j + 1;  // arbitrary offsets
    }

    OopType anOop; // the OOP to use as the root of the path
    anOop = GciExecuteStr("AllComponents detect:[1i1 partNumber = 1234]", OOP_NIL);
    if (anOop == OOP_NIL) {
        return; // error during resolve
    }

    OopType result;
    GciFetchPaths(&anOop, 1, aPath, &aSize, 1, &result);
}
```
Example 2: Calling sequence for multiple objects with a single path

```c
void fetchPath2(void)
{
    OopType coll = GciResolveSymbol(“AllComponents”, OOP_NIL);
    if (coll == OOP_NIL) {
        return ; // error during resolve
    }
    enum { num_roots = 3 ,
            path_size = 5 };
    OopType oops[num_roots];
    int numRet = GciFetchVaryingOops(coll, 1, oops, num_roots);
    if (numRet != num_roots) {
        return; // error during fetch or collection too small
    }

    int aPath[path_size];
    int aSize = path_size;
    for (int j = 0; j < path_size; j++) {
        aPath[j] = 1;  // arbitrary offsets
    }
    OopType results[num_roots];
    GciFetchPaths(oops, num_roots, aPath, &aSize, 1, results);
}
```
Example 3: Calling sequence for a single object with multiple paths

void fetchPath3(void)
{
    OopType anOop; // the OOP to use as the root of the path
    anOop = GciExecuteStr("AllComponents detect:[i:i partNumber = 1234]", OOP_NIL);
    if (anOop == OOP_NIL) {
        return; // error during execution
    }

    enum { num_paths = 10,
          path_size = 5};

    int pathSizes[num_paths];
    int paths[path_size * num_paths ];
    int idx = 0;
    for (int j = 0; j < num_paths; j++) {
        for (int k = 0; k < path_size; k++) {
            paths[idx++] = k + 1; // arbitrary offset
        }
    }
    OopType results[num_paths];
    GciFetchPaths (&anOop, 1, paths, pathSizes, num_paths, results);
}
Example 4: Calling sequence for multiple objects with multiple paths

```c
void fetchPath4(void)
{
    OopType coll = GciResolveSymbol("AllComponents", OOP_NIL);
    if (coll == OOP_NIL) {
        return; // error during resolve
    }

    enum { num_roots = 10,
           num_paths = 3,
           path_size = 5 };}

    OopType oops[num_roots];
    int numRet = GciFetchVaryingOops(coll, 1, oops, num_roots);
    if (numRet != num_roots) {
        return; // error during fetch or collection too small
    }

    int pathSizes[num_paths];
    int paths[path_size * num_paths];
    int idx = 0;
    for (int j = 0; j < num_paths; j++) {
        for (int k = 0; k < path_size; k++) {
            paths[idx++] = k + 1; // arbitrary offset
        }
    }

    OopType results[num_roots * num_paths];
    GciFetchPaths(oops, num_roots, paths, pathSizes, num_paths, 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 248.

```c
void fetchPath5(void)
{
    // retrieve a random instance of class Component */
    OopType aComponent = GciExecuteStr("AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        return; // error in execute, or detect: found nothing
    }

    // fetch name instVar of 5th element of aComponent’s partsList */
    enum { path_size = 2 };  
    int path[path_size];
    path[0] = -5;  // 5th varying instVar  
    path[1] = GciIvNameToIdx(GciFetchClass(aComponent), "name");
    int pathSizes = path_size;
    OopType oName;
    GciFetchPaths(&aComponent, 1, path, &pathSizes, 1, &oName);
}
```

See Also

GciStorePaths, page 471
GciFetchSharedCounterValuesNoLock

Fetch the value of multiple shared counters without locking them.

Syntax

```c
int GciFetchSharedCounterValuesNoLock(
    int startIndex,
    int64_t buffer[],
    size_t * maxReturn);
```

Input Arguments

- `startIndex` The offset into the shared counters array of the first shared counter value to fetch.
- `maxReturn` Pointer to a value that indicates the maximum number of shared counters to fetch.

Result Arguments

- `buffer` Pointer to a buffer where the shared counter values will be stored. The buffer must be large enough to accommodate `maxReturn` 8-byte values, and the address must be aligned on an 8-byte boundary.

Return Value

Returns an int indicating the number of shared counter values successfully stored in the buffer. Returns -1 if a bad argument is detected.

Description

Fetch the values of multiple shared counters in a single call, without locking any of them. The values of the `maxReturn` count of shared counters starting at the offset indicated by `startIndex` (0-based) are put into the buffer `buffer`. `buffer` must be large enough to accommodate `maxReturn` 8-byte values, and be aligned on an 8-byte boundary.

Not supported for remote GCI interfaces.
See Also

GciFetchNumSharedCounters, page 225
GciDecSharedCounter, page 172
GciIncSharedCounter, page 271
GciSetSharedCounter, page 437
GciReadSharedCounter, page 394
GciReadSharedCounterNoLock, page 395
GciFetchSize_

Fetch the size of an object.

Syntax

    int64 GciFetchSize_(
       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.

    NOTE
    In previous GemStone/S 64 Bit releases, this function was named GciFetchSize
    (without the underscore). Customers must ensure that the variables that receive this
    function’s result are large enough to accommodate an int64 value.

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 6.10:
Table 6.10 Differences in Reported Object Size

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Object&gt;&gt;size (Smalltalk)</th>
<th>GciFetchSize_</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=Pointer</td>
<td>Number of indexed elements in the object (0 if not indexed)</td>
<td>Number of indexed elements PLUS number of named instance variables</td>
</tr>
<tr>
<td>1=Byte</td>
<td>Number of indexed elements in the object</td>
<td>Same as Smalltalk message &quot;size&quot;</td>
</tr>
<tr>
<td>2=NSC</td>
<td>Number of elements in the object</td>
<td>Same as Smalltalk message &quot;size&quot;</td>
</tr>
<tr>
<td>3=Special</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

```c
void fetchSize_example(void) {
    const char* str = "abcdef";
    OopType oString = GciNewString(str);

    int64 itsSize = GciFetchSize_(oString);
    if (itsSize != (int64)strlen(str)) {
        printf("error during fetch size\n");
    }
}
```

See Also

GciFetchClass, page 211
GciFetchNamedSize, page 222
GciFetchObjImpl, page 228
GciFetchVaryingOop, page 248
GciFetchVaryingOop

Fetch the OOP of one unnamed instance variable from an indexable pointer object or NSC.

Syntax

OopType GciFetchVaryingOop(  
    OopType theObject,  
    int64 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:

```
run
“define the class Component and compile accessor methods”
| cls |
cls := Array subclass: #Component
   instVarNames: #(#partNumber #name #cost
   “varying instVars form the partsList”)
classVars: #()
classInstVars: #()
poolDictionaries: #()
cls compileAccessingMethodsFor: cls instVarNames .
^ cls
%
run
“create and populate the set of all Components”
| allC |
allC := IdentitySet new .
UserGlobals at: #AllComponents put: allC .
1 to: 100 do:
  [:j | aComp |
    aComp := Component new .
    aComp partNumber: 1200 + j .
    aComp name: ‘part’ + j asString .
    aComp cost: j asFloat .
    allC add: aComp .
  ] .
^ allC size
%
run
“build a parts list for each part.”
| allC idx |
allC := Array withAll: AllComponents .
idx := 1 .
AllComponents do:
  [:aComp | | list |
    list := Array new: (idx \ 10 ) . “list size varies from 0 to 9”
    idx > 75 ifTrue:
      [ idx := 1 ].
  1 to: list size do:
      [:k |
list at: k put: (allC at: idx + (k * 2)).
].
}. aComp addAll: list.
idx := idx + 1.
].
%

Now execute this C code:

```c
OopType fetchVaryingOopExample(void)
{
  // retrieve a random instance of class Component */
  OopType aComponent = GciExecuteStr(
      "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
  if (aComponent == OOP_NIL) {
    return OOP_NIL; // error in execute, or detect: found nothing
  }

  /* fetch 3rd element of aComponent’s parts list */
  OopType aSubComponent = GciFetchVaryingOop(aComponent, 3);
  return aSubComponent;
}
```

See Also

GciFetchNamedOop, page 216
GciFetchNamedOops, page 219
GciFetchVaryingOops, page 251
GciStoreIdxOop, page 454
GciStoreIdxOops, page 456
GciStoreNamedOop, page 459
GciStoreNamedOops, page 462
**GciFetchVaryingOops**

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

**Syntax**

```c
int GciFetchVaryingOops(
    OopType theObject,
    int64 startIndex,
    OopType theOops[],
    int 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 248.

```c
int fetchVaryingOopsExample(void)
{
    // retrieve a random instance of class Component */
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        return -1; // error in execute, or detect: found nothing
    }

    /* fetch the up to the first 5 elements of aComponent’s parts list */
    enum { num_oops = 5 };  
    OopType oBuf[num_oops];

    int numRet = GciFetchVaryingOops(aComponent, 1, oBuf, num_oops);
    // at this point we have   0 <= numRet <= 5
    return numRet;
}
```

See Also

GciFetchNamedOop, page 216
GciFetchNamedOops, page 219
GciFetchVaryingOop, page 248
GciStoreIdxOop, page 454
GciStoreIdxOops, page 456
GciStoreNamedOop, page 459
GciStoreNamedOops, page 462
GciFetchVaryingSize_

Fetch the number of unnamed instance variables in a pointer object or NSC.

Syntax

```c
int64 GciFetchVaryingSize_
    (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

NOTE
In previous GemStone/S 64 Bit releases, this function was named GciFetchVaryingSize (without the underscore). Customers must ensure that the variables that receive this function’s result are large enough to accommodate an int64 value.

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 248.

```c
int64 fetchVaryingSizeExample(void)
{
    // retrieve a random instance of class Component */
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        return -1; // error in execute, or detect: found nothing
    }

    /* fetch the size of aComponent’s partsList */
    int64 theSize = GciFetchVaryingSize_(aComponent);
    return theSize;
}
```

See Also

GciFetchClass, page 211
GciFetchNamedSize, page 222
GciFetchObjImpl, page 228
GciFetchSize_, page 246
GciSetVaryingSize, page 439
**GciFindObjRep**

Fetch an object report in a traversal buffer.

**Syntax**

```c
GciObjRepHdrSType * GciFindObjRep(
    GciTravBufType * 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 53.
Example

```c
GciObjRepHdrSType* findObjRepExample(GciTravBufType *buf, OopType objId)
{
    GciObjRepHdrSType *theReport = GciFindObjRep(buf, objId);
    if (theReport == NULL) {
        GciErrSType errInfo;
        if (GciErr(&errInfo)) {
            printf("error category "FMT_OID" number %d, %s\n",
                errInfo.category, errInfo.number, errInfo.message);
        }
    }
    return theReport;
}
```

See Also

GciMoreTraversal, page 293
GciObjRepSize_, page 348
GciTraverseObjs, page 510
GciFloatKind

Obtain the float kind corresponding to a C double value.

Syntax

```c
GciFloatKindEType GciFloatKind(
    double                     aReal
);
```

Input Arguments

- `aReal` A floating point value.

Return Value

Returns the type of GemStone Float object that corresponds to the C value.

Description

This function obtains the kind of GemStone Float object that corresponds to the C floating point value `aReal`.

See Also

- GciFltToOop, page 258
- GciOopToFlt, page 362
GciFltToOop

Convert a C double value to a SmallDouble or Float object.

Syntax

OopType GciFltToOop(  
    double  aReal );

Input Arguments

aReal  The floating point value to be translated into an object.

Return Value

Returns the OOP of the GemStone SmallDouble or 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

#include <stdlib.h>

void fltToOopExample(void)
{
    // random double to Oop conversion producing a Float or SmallFloat
    double rand = drand48() * 1.0e38;
    OopType oFltObj = GciFltToOop(rand);

    OopType oClass = GciFetchClass(oFltObj);
    const char* kind;
    if (oClass == OOP_CLASS_SMALL_DOUBLE) {
        kind = "SmallDouble";
    } else if (oClass == OOP_CLASS_FLOAT) {
        kind = "Float";
    } else {
        kind = "Unexpected";
    }
    printf("result is a %s, class oop = "FMT_OID"\n", kind, oClass);
}

See Also

GciOopToFlt, page 362
GciGetFreeOopsEncoded, page 264
GciGetFreeOop

Allocate an OOP.

Syntax

OopType GciGetFreeOop()

Return Value

Returns an unused object identifier (OOP).
You cannot use the result of GciGetFreeOop to create a Symbol object.

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 
\texttt{GciGetFreeOop} can be used only as a parameter to the following function calls, under the 
given restrictions:

- As the \texttt{objID} of the object to be created
  \begin{verbatim}
  GciCreateByteObj
  \end{verbatim}

- As the \texttt{objID} of the object to be created, or as an element of the value buffer
  \begin{verbatim}
  GciCreateOopObj
  \end{verbatim}

- As an element of the value buffer only
  \begin{verbatim}
  GciStoreOop
  GciStoreOops
  GciStoreIdxOop
  GciStoreIdxOops
  GciStoreNamedOop
  GciStoreNamedOops
  GciStoreTrav
  GciAppendOops
  GciAddOopToNsc
  GciAddOopsToNsc
  GciNewOopUsingObjRep
  \end{verbatim}

- As an element of \texttt{newValues} only
  \begin{verbatim}
  GciStorePaths
  \end{verbatim}

\textbf{See Also}

\texttt{GciCreateByteObj}, page 158  
\texttt{GciCreateOopObj}, page 160  
\texttt{GciGetFreeOops}, page 262  
\texttt{GciGetFreeOopsEncoded}, page 264
GciGetFreeOops

Allocate multiple OOPs.

Syntax

```c
void GciGetFreeOops(
   int 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
\texttt{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**
  \begin{verbatim}
  GciCreateByteObj
  \end{verbatim}

- **As the objID of the object to be created, or as an element of the value buffer**
  \begin{verbatim}
  GciCreateOopObj
  \end{verbatim}

- **As an element of the value buffer, only**
  \begin{verbatim}
  GciStoreOop
  GciStoreOoops
  GciStoreIdxOop
  GciStoreIdxOoops
  GciStoreNamedOop
  GciStoreNamedOoops
  GciStoreTrav
  GciAppendOops
  GciAddOopToNsc
  GciAddOoopsToNsc
  GciNewOopUsingObjRep
  \end{verbatim}

- **As an element of newValues, only**
  \begin{verbatim}
  GciStorePaths
  \end{verbatim}

\textbf{See Also}

- \texttt{GciCreateByteObj}, page 158
- \texttt{GciCreateOopObj}, page 160
- \texttt{GciGetFreeOop}, page 260
- \texttt{GciGetFreeOopsEncoded}, page 264
GciGetFreeOopsEncoded

Allocate multiple OOPs.

Syntax

```c
void GciGetFreeOopsEncoded(
    int * count,
    OopType * encodedOops);
```

Input Arguments

- `count` The number of OOPs to allocate.
- `encodedOops` A pointer to memory for holding encoded oops. Must be large enough to hold at least the input value of `count`.

Result Arguments

- `count` The number of OOPs returned in the encoded OOP array.
- `encodedOops` An array to hold the returned encoded oops. Must be large enough to hold at least the input value of `count`.

Description

This function is identical to `GciGetFreeOop`, except that it returns OOPs in an encoded array that is more compact for less network I/O. Before the OOPs can be used, the encoded array must be decoded by calling `GciDecodeOopArray()`.

See Also

- GciGetFreeOop, page 260
- GciGetFreeOops, page 262
- GciFetchNumEncodedOops, page 224
- GciEnableFreeOopEncoding, page 183
- GciEncodeOopArray, page 187
- GciDecodeOopArray, page 170
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

void getSessionExample(const char* userId, const char* password)
{
    if (GciLogin(userId, password)) {
        GciSessionIdType sessId = GciGetSessionId();
        printf("sessionId is %d \n", sessId);
    }
    GciLogout();
    GciSessionIdType sessId = GciGetSessionId();
    if (sessId != GCI_INVALID_SESSION_ID) {
        printf("unexpected sessionId %d after logout \n", sessId);
    }
}

See Also

GciLogin, page 289
GciSetSessionId, page 435
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 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 a signal handler 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 a signal handler.

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 441
GciHiddenSetIncludesOop

Determines whether the given OOP is present in the specified hidden set.

Syntax

```
BoolType GciHiddenSetIncludesOop(
    OopType theOop,
    int hiddenSetId );
```

Input Arguments

- **theOop**  The OOP to search for.
- **hiddenSetId**  The index to the hidden set to search.

Return Value

True if the OOP was found; false otherwise.

Description

The Gem holds objects in a number of sets ordinarily hidden from the user. **GciHiddenSetIncludesOop** allows you to pass in an index to a specified hidden set to determine if the set includes an specific object. For indexes of available hidden sets, see the GemStone Smalltalk method `System Class >> HiddenSetSpecifiers`.

Example

```
OopType TrackedSetContainsOop(OopType anOop)
{
    if (GciHiddenSetIncludesOop(anOop, 40/* GciTrackedObjs */))
        return OOP_TRUE;
    else
        return OOP_FALSE;
}
```
GCI_I64_IS_SMALL_INT

Determine whether or not a C 64-bit integer value can be translated into a SmallInteger object.

Syntax

static inline BoolType GCI_I64_IS_SMALL_INT(anInt)

Input Arguments

anInt A C 64-bit signed integer.

Result Value

A C Boolean value. Returns TRUE if anInt is within SmallInteger range, FALSE otherwise. A SmallInteger has a 61-bit two’s-complement integer and three tag bits.

For a positive argument to be within the range of the GemStone SmallInteger class, its top four bits must be 2r0000. For a negative argument, the top four bits must be 2r1111.

Description

This macro tests to see if anInt can be represented as a SmallInteger.

See Also

GCI_OOP_IS_SMALL_INT, page 352
**GciI64ToOop**

Convert a C 64-bit integer value to a GemStone object.

**Syntax**

```c
OopType GciI64ToOop(
    int64    anInt
);```

**Input Arguments**

*anInt*  
A C 64-bit signed integer.

**Return Value**

The `GciI64ToOop` function returns the OOP of a GemStone object whose value is equivalent to `anInt`.

**Description**

The `GciI64ToOop` function translates a C 64-bit integer (int64_t) value into the equivalent GemStone object. If the result is not a SmallInteger, the result is automatically saved by a GciSaveObjs() call.

**See Also**

- GciOopToI64, page 366
- GciOopToI64_, page 367
- GciSaveObjs, page 422
GciIncSharedCounter

Increment the value of a shared counter.

Syntax

```
BoolType GciIncSharedCounter(
    int64_t counterIdx,
    int64_t * value);
```

Input Arguments

- `counterIdx` The offset into the shared counters array of the value to increment.
- `value` Pointer to a value that indicates how much to increment the shared counter by. Shared counters cannot be incremented to a value greater than INT_MAX (2147483647). Attempt to do so will not cause an error, but will set the counter to a value of INT_MAX.

Result Arguments

- `value` Pointer to a value that indicates the new value of the shared counter, after incrementing.

Return Value

Returns a C Boolean value indicating if the shared counter was successfully incremented. Returns TRUE if successful, FALSE if an error occurred.

Description

This function increments the value of a particular shared counter by a specified amount. The shared counter is specified by index. The maximum value of this shared counter is INT_MAX (2147483647), attempts to increase a shared counter to higher values is not an error, but does not cause the value to increase further.

This function is not supported for remote GCI interfaces, and will always return FALSE.
See Also

GciFetchNumSharedCounters, page 225
GciDecSharedCounter, page 172
GciSetSharedCounter, page 437
GciReadSharedCounter, page 394
GciReadSharedCounterNoLock, page 395
GciFetchSharedCounterValuesNoLock, page 244
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.

(Note that when doing run-time binding, you would call GciRtlLoad before calling GciInit. For details, see “Building the Application” on page 55.)

When GemBuilder is initialized on UNIX platforms, it establishes its own handler for SIGIO interrupts. See “Signal Handling in Your GemBuilder Application” on page 46 for information on GciInit’s handling of interrupts and pointers on making GemBuilder, application, and third-party handlers work together.

See Also

GciInitAppName, page 274
GciInitAppName

Override the default application configuration file name.

Syntax

void GciInitAppName(
    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 System Administration Guide for GemStone/S 64 Bit.

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.
GciInitAppName

Override the default application configuration file name and the size of temporary object memory.

Syntax

```c
void GciInitAppName_
const char * applicationName,
BoolType logWarnings,
unsigned int gemTempObjCacheOverrideKB
```

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.
- `gemTempObjCacheOverrideKB`: If non-zero, defines the maximum size (in KB) of temporary object memory for this application. This value overrides any GEM_TEMPOBJ_CACHE_SIZE settings in configuration files read by GciInit.

Description

This function is similar to the GciInitAppName function (page 274), but with one exception: you can override any GEM_TEMPOBJ_CACHE_SIZE settings in configuration files read by GciInit.

If your application calls this function, it must not call GciInitAppName.
GciInstallUserAction

Associate a C function with a Smalltalk user action.

Syntax

void GciInstallUserAction(
    GciUserActionSType * userAction);

void GciInstallUserAction_(
    GciUserActionSType * userAction,
    BoolType errorIfDuplicate);

Input Arguments

userAction A pointer to a C structure that describes the user-written C function.
errorIfDuplicate If True, return an error if there is already a user action with the specified name. If False, leave the existing user action in place and ignore the current call.

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 99
“GciUserActionShutdown” on page 518
GciInstMethodForClass

Compile an instance method for a class.

Syntax

OopType GciInstMethodForClass(
    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`).

Return Value

Returns OOP_NIL, unless there were compiler warnings (such as variables declared but not used, etc.), in which case the return will be the OOP of a string containing the warning messages.

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 Programming Guide for GemStone/S 64 Bit for a list of the optimized selectors.

To remove a class method, use GciExecuteStr instead.

Example

```c
void instanceMethodExample(void)
{
    // Assumes the topaz code for GciFetchVaryingOop example
    // has been executed.

    OopType theClass = GciResolveSymbol("Component", OOP_NIL);
    OopType oCateg = GciNewString("printing");
    // method to return the part number as a String
    OopType oMethodSrc = GciNewString("partNumString ^ partNumber
        asString ");

    GciInstMethodForClass(oMethodSrc, theClass, oCateg, OOP_NIL);
    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        printf("error category "FMT_OID" number %d, %s
",
            errInfo.category, errInfo.number, errInfo.message);
    }
}
```

See Also

GciClassMethodForClass, page 141
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 signal 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 131
GciIsKindOf

Determine whether or not an object is some kind of a given class or class history.

Syntax

```
BoolType GciIsKindOf(
    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 281
- GciIsSubclassOf, page 283
- GciIsSubclassListOfClass, page 284
GcilsKindOfClass

Determine whether or not an object is some kind of a given class.

Syntax

```
BoolType GcilsKindOfClass(
    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

GcilsKindOfClass returns TRUE when the class of `anObj` or any of its superclasses is `givenClass`. It returns FALSE otherwise.

Description

GcilsKindOfClass 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 GcilsKindOfClass 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 GcilsKindOf function.

See Also

- GcilsKindOf, page 280
- GcilsSubclassOf, page 283
- GcilsSubclassOfClass, page 284
GcilsRemote

Determine whether the application is running linked or remotely.

Syntax

BoolType GcilsRemote()

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.
GcIsSubclassOf

Determine whether or not a class is a subclass of a given class or class history.

Syntax

```c
BoolType GcIsSubclassOf(
    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

GcIsSubclassOf returns TRUE when `aClass` or any of its superclasses is in the class history of `givenClass`. It returns FALSE otherwise.

Description

GcIsSubclassOf 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), GcIsSubclassOf uses `givenClass`'s class history. When `givenClass` is itself a class history, GcIsSubclassOf uses `givenClass` directly.

Since GcIsSubclassOf 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 GcIsSubclassOfClass function.

See Also

- GcIsKindOf, page 280
- GcIsKindOfClass, page 281
- GcIsSubclassOfClass, page 284
GcIsSubclassOfClass

Determine whether or not a class is a subclass of a given class.

Syntax

```
BoolType GcIsSubclassOf(
    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

GcIsSubclassOf returns TRUE when `aClass` or any of its superclasses is `givenClass`. It returns FALSE otherwise.

Description

GcIsSubclassOfClass 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 GcIsSubclassOfClass 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 GcIsSubclassOf function.

See Also

GcIsKindOf, page 280
GcIsKindOfClass, page 281
GcIsSubclassOf, page 283
GciIvNameToIdx

Fetch the index of an instance variable name.

Syntax

    int GciIvNameToIdx(
        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

```c
int nameToIdx_example(void)
{
    // Assumes topaz code for GciFetchVaryingOop example has run

    OopType theClass = GciResolveSymbol("Component", OOP_NIL);
    int idx = GciIvNameToIdx(theClass, "cost");
    if (idx < 1) {
        printf("error during GciIvNameToIdx\n");
    }
    return idx;
}
```

See Also

- GciClassNamedSize, page 143
- GciFetchNamedOop, page 216
- GciFetchNamedOops, page 219
- GciStoreNamedOop, page 459
- GciStoreNamedOops, page 462
GciLoadUserActionLibrary

Load an application user action library.

Syntax

```c
BoolType GciLoadUserActionLibrary(
    const char * uaLibraryName[],
    BoolType mustExist,
    void ** libHandlePtr,
    char infoBuf[],
    int64 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 cannot 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 276  
GciInUserAction, page 279  
GciUserActionShutdown, page 518
GciLogin

Start a user session.

Syntax

```c
BoolType GciLogin(
    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

```c
BoolType login_example(void)
{
    // assume the netldi on machine lichen been started with -a -g
    // so that host userId and host password are not required.
    const char* StoneName    = "!tcp@lichen!gs64stone";
    const char* HostUserId   = "";
    const char* HostPassword = "";
    const char* GemService   = "!tcp@lichen!gemnetobject";
    const char* gsUserName   = "isaacNewton";
    const char* gsPassword   = "pomme";

    // GciInit required before first login
    if (!GciInit()) {
        printf("GciInit failed\n");
        return FALSE;
    }

    GciSetNet(StoneName, HostUserId, HostPassword, GemService);
    BoolType success = GciLogin(gsUserName, gsPassword);
    if (!success) {
        GciErrSType errInfo;
        if (GciErr(&errInfo)) {
            printf("error category "%s, number %d, %s\n",
                    errInfo.category, errInfo.number, errInfo.message);
        }
    }
    return success;
}
```

See Also

- GciGetSessionId, page 265
- GciLogout, page 291
- GciSetNet, page 432
- GciSetSessionId, page 435
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 265 for more information.)

See Also

GciGetSessionId, page 265
GciLogin, page 289
GciSetSessionId, page 435
GciLongJmp

Provides equivalent functionality to the corresponding longjmp() or _longjmp() function.

Syntax

```c
void GciLongJmp(
    GciJmpBufSType * jumpBuffer,
    int val);
```

Input Arguments

- `jumpBuffer` A pointer to a jump buffer.

Description

Except for the difference in the first argument type, the semantics of this function are the same as for longjmp() on Solaris and _longjmp() on HP-UX.

See Also

- GciErr, page 189
- GciPopErrJump, page 386
- GciPushErrJump, page 391
- GciSetErrJump, page 427
- Gci_SETJMP, page 431
GciMoreTraversal

Continue object traversal, reusing a given buffer.

Syntax

```
BoolType GciMoreTraversal(
    GciTravBufType * travBuff);
```

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 53.
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 `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 511 describes the organization of traversal buffers in detail.

`GciMoreTraversal` provides automatic byte swizzling for Float and SmallFloat objects. (For more about byte swizzling, see page 29.)
Example

```c
void moreTraversalExample(void)
{
    // Assumes topaz code for GciFetchVaryingOops example has run

    OopType rootObj = GciResolveSymbol("AllComponents", OOP_NIL);
    GciTravBufType *buf = GciTravBufType::malloc(8000);

    int totalCount = 0;
    // traverse the AllComponents collection to 10 levels deep
    BoolType done = GciTraverseObjs(&rootObj, 1, buf, 10);
    while (! done) {
        int objCount = 0;
        GciObjRepHdrSType *rpt = buf->firstReportHdr();
        GciObjRepHdrSType *limit = buf->readLimitHdr();
        while (rpt < limit) {
            objCount++;
            rpt = rpt->nextReport();
        }
        totalCount += objCount;
        done = GciMoreTraversal(buf);
    }
    buf->free();
    printf("traversal returned %d total objects\n", totalCount);
}
```

See Also

- GCI_ALIGN, page 119
- GciFindObjRep, page 255
- GciNbMoreTraversal, page 314
- GciNbTraverseObjs, page 328
- GciObjRepSize_, page 348
- GciTraverseObjs, page 510
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 114
GCI_CHR_TO_OOP, page 134
GciCommit, page 147
GciNbCommit, page 301
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 114
GciBegin, page 127
GciExecuteStr, page 195
GciNbAbort, page 296
GciNbExecuteStr, page 308
GciNbClampedTrav

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

Syntax

```c
void GciNbClampedTrav(
    const OopType * theOops,
    int numOops,
    GciClampedTravArgsSType *travArgs);
```

Input Arguments

- `theOops`: An array of OOPs representing the objects to traverse.
- `numOops`: The number of elements in `theOops`.
- `travArgs`: Pointer to an instance of `GciClampedTravArgsSType`. See `GciClampedTrav` (page 135) for documentation on the fields in `travArgs`.

Result Arguments

- `travArgs`: Pointer to an instance of `GciClampedTravArgsSType` 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 135

298 VMware, Inc. June 2011
**GciNbClampedTraverseObjs**

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

**Syntax**

```c
void GciNbClampedTraverseObjs(
    OopType    clampSpec,
    const OopType theOops[],
    int        numOops,
    GciTravBufType * travBuff,
    int        level);
```

**Input Arguments**

- `clampSpec`: The OOP of the Smalltalk ClampSpecification to be used.
- `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`, an 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**

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`.

---

*June 2011 VMware, Inc.*

299
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 Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

GemBuilder clamped traversal functions are intended primarily for GemStone internal use.

See Also

`GciClampedTraverseObjs`, page 138
`GciNbTraverseObjs`, page 328
`GciTraverseObjs`, page 510
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 114
GCI_CHR_TO_OOP, page 134
GciCommit, page 147
GciNbAbort, page 296
**GciNbContinue**

Continue code execution in GemStone after an error (nonblocking).

**Syntax**

```c
void GciNbContinue(
    OopType    process);
```

**Input Arguments**

`process`  
The OOP of a GsProcess 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 145
- `GciContinue`, page 154
- `GciErr`, page 189
- `GciExecute`, page 191
- `GciNbExecute`, page 306
GciNbContinueWith

Continue code execution in GemStone after an error (nonblocking).

Syntax

```c
void GciNbContinueWith(
    OopType process,
    OopType replaceTopOfStack,
    int flags,
    GciErrSType *error);
```

Input Arguments

- **process**: The OOP of a GsProcess 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 154
GciContinueWith, page 156
GciErr, page 189
GciExecute, page 191
GciNbExecute, page 306
GciPerformNoDebug, page 373
**GciNbEnd**

Test the status of nonblocking call in progress for completion.

**Syntax**

```c
GciNbProgressEType GciNbEnd(
    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_RESULTREADY 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

```c
void nbEnd_example(void)
{
    void *resultPtr;
    GciNbExecuteStr("Globals size", OOP_NIL);
    do {
        // wait for non-blocking result
        GciHostMilliSleep(1);
    } while (GciNbEnd(&resultPtr) != GCI_RESULT_READY);

    OopType result = *(OopType*)resultPtr;
    BoolType conversionErr = FALSE;
    int gSize = GciOopToI32_(result, &conversionErr);
    if (conversionErr) {
        printf("error in execution\n");
    } else {
        printf("Globals size = %d \n", gSize);
    }
}
```

See Also

GciCallInProgress, page 131
GciNbExecute

Execute a Smalltalk expression contained in a String object (nonblocking).

Syntax

```c
void GciNbExecute(
    OopType source,
    OopType symbolList);

void GciNbExecute_(
    OopType source,
    OopType symbolList,
    ushort environmentId);
```

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`).
- **environmentId**
  The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 154
GciErr, page 189
GciExecute, page 191
GciExecuteFromContext, page 193
GciExecuteStr, page 195
GciExecuteStrFromContext, page 198
GciNbContinue, page 302
GciNbExecuteStr, page 308
GciNbExecuteStrFromContext, page 310
GciNbExecuteStr

Execute a Smalltalk expression contained in a C string (nonblocking).

Syntax

```c
void GciNbExecuteStr(
    const char source[],
    OopType symbolList);

void GciNbExecuteStr_(
    const char source[],
    OopType symbolList,
    ushort environmentId);
```

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`).
- **environmentId**: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 154
- `GciErr`, page 189
- `GciExecute`, page 191
- `GciExecuteFromContext`, page 193
- `GciExecuteStr`, page 195
- `GciExecuteStrFromContext`, page 198
- `GciNbContinue`, page 302
- `GciNbExecute`, page 306
- `GciNbExecuteStrFromContext`, page 310
GciNbExecuteStrFromContext

Execute a Smalltalk expression contained in a C string as if it were a message sent to an object (nonblocking).

Syntax

```c
void GciNbExecuteStrFromContext(
    const char *source[],
    OopType contextObject,
    OopType symbolList);

void GciNbExecuteStrFromContext_ ( 
    const char *source[],
    OopType contextObject,
    OopType symbolList,
    ushort environmentId);
```

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).
- **environmentId**
  The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 154
- `GciErr`, page 189
- `GciExecute`, page 191
- `GciExecuteFromContext`, page 193
- `GciExecuteStr`, page 195
- `GciExecuteStrFromContext`, page 198
- `GciNbContinue`, page 302
- `GciNbExecute`, page 306
- `GciNbExecuteStr`, page 308
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

```c
void GciNbExecuteStrTrav(
    const char source[ ],
    OopType contextObject,
    OopType symbolList,
    GciClampedTravArgsSType *travArgs);
```

```c
void GciNbExecuteStrTrav_(
    const char source[ ],
    OopType contextObject,
    OopType symbolList,
    GciClampedTravArgsSType *travArgs,
    ushort environmentId );
```

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 an instance of `GciClampedTravArgsSType`. See the `GciExecuteStrTrav` function (page 201) for field definitions.
- **environmentId**: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.
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 201
GciExecuteStr, page 195
GciMoreTraversal, page 293
GciPerformTraverse, page 379
GciNbMoreTraversal

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

Syntax

```c
void GciNbMoreTraversal(
    GciTravBufType *   travBuff);
```

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 Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

See Also

- `GCI_ALIGN`, page 119
- `GciFindObjRep`, page 255
- `GciMoreTraversal`, page 293
- `GciNbTraverseObjs`, page 328
- `GciObjRepSize_`, page 348
- `GciTraverseObjs`, page 510
**GciNbPerform**

Send a message to a GemStone object (nonblocking).

**Syntax**

```c
void GciNbPerform(
    OopType  receiver,
    const char  selector[],
    const OopType  args[],
    int  numArgs);

void GciNbPerform_(
    OopType  receiver,
    const char  selector[],
    const OopType  args[],
    int  numArgs,
    ushort  environmentId);
```

**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.
- `environmentId` The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

**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 154
GciErr, page 189
GciExecute, page 191
GciNbContinue, page 302
GciNbExecute, page 306
GciNbPerformNoDebug, page 317
GciPerform, page 371
GciPerformNoDebug, page 373
GciPerformSymDbg, page 375
GciNbPerformNoDebug

Send a message to a GemStone object, and temporarily disable debugging (nonblocking).

Syntax

```c
void GciNbPerformNoDebug(
    OopType receiver,
    const char selector[],
    const OopType args[],
    int numArgs,
    int flags);

void GciNbPerformNoDebug_(_
    OopType receiver,
    const char selector[],
    const OopType args[],
    int numArgs,
    int flags,
    ushort environmentId);
```

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.

- **environmentId**
  The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.
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 154
GciErr, page 189
GciExecute, page 191
GciNbContinue, page 302
GciNbExecute, page 306
GciNbPerform, page 315
GciPerform, page 371
GciPerformNoDebug, page 373
GciPerformSymDbg, page 375
GciNbPerformTrav

First send a message to a GemStone object, then traverse the result of the message (nonblocking).

Syntax

```
BoolType GciNbPerformTrav(
    OopType receiver,
    const char * selector,
    const OopType * args,
    int numArgs,
    GciClampedTravArgsSType *travArgs);
```

```
BoolType GciNbPerformTrav_(
    OopType receiver,
    const char * selector,
    const OopType * args,
    int numArgs,
    GciClampedTravArgsSType *travArgs,
    ushort environmentId);
```

Input Arguments

- **receiver**: The OOP of the receiver of the message.
- **selector**: A pointer to a character collection containing 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 an instance of `GciClampedTravArgsSType`. See the `GciClampedTrav` function (page 135) for documentation of the fields in `travArgs`.
- **environmentId**: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101
Result Arguments

The result of the `GciNbPerformTrav` 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 377
- `GciPerform`, page 371
- `GciClampedTrav`, page 135
GciNbStoreTrav

Store multiple traversal buffer values in objects (nonblocking).

Syntax

```c
void GciNbStoreTrav(
    GciTravBufType * travBuff,
    int behaviorFlag);
```

Input Arguments

- **travBuff**
  A buffer that contains the object reports to be stored. The first
  element in the buffer is an 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 Float and SmallFloat objects. (For
more about byte swizzling, see page 29.)

See Also

- GciMoreTraversal, page 293
- GciNbMoreTraversal, page 314
- GciNbTraverseObjs, page 328
- GciNewOopUsingObjRep, page 338
GciStoreTrav, page 478
GciTraverseObjs, page 510
GciNbStoreTravDo_

Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object (non-blocking).

NOTE
In previous GemStone/S 64 Bit releases, this function was named GciNbStoreTravDo (without the underscore).

Syntax

```c
void GciNbStoreTravDo_( 
    GciStoreTravDoArgsSType *stdArgs);
```

Input Arguments

- `stdArgs` An instance of `GciStoreTravDoArgsSType`. For details, refer to the discussion of `GciStoreTravDo_` on page 482.

Return Value

Unlike `GciStoreTravDo_`, the `GciNbStoreTravDo_` function 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 `GciStoreTravDo_` by using the argument to `GciNbEnd`.

Description

The `GciNbStoreTravDo_` function is equivalent in effect to `GciStoreTravDo_`. However, `GciNbStoreTravDo_` permits the application to proceed with non-GemStone tasks while the traversal is done, and `GciStoreTravDo_` does not.

See Also

- GciNbClampedTrav, page 298
- GciNbEnd, page 304
- GciNbStoreTrav, page 321
- GciNbStoreTravDoTrav_, page 324
- GciStoreTravDo_, page 482
GciNbStoreTravDoTrav_

Combine in a single function the calls to GciNbStoreTravDo_ and GciNbClampedTrav, to store multiple traversal buffer values in objects, execute the specified code, and traverse the result object (non-blocking).

NOTE

In previous GemStone/S 64 Bit releases, this function was named GciNbStoreTravDoTrav (without the underscore).

Syntax

```c
void GciNbStoreTravDoTrav_(
    GciStoreTravDoArgsSType *stdArgs,
    GciClampedTravArgsSType *ctArgs);
```

Input Arguments

- `stdArgs` An instance of GciStoreTravDoArgsSType. For details, refer to the discussion of GciStoreTravDo_ on page 482.
- `ctArgs` An instance of GciClampedTravArgsSType. For details, see the discussion of GciClampedTrav on page 135.

Return Value

The GciNbStoreTravDoTrav_ function, unlike GciStoreTravDoTrav_, 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 GciStoreTravDoTrav_ by using the argument to GciNbEnd.

Description

This function allows the client to execute behavior on the Gem and return the traversal of the result object in a single network round-trip.

The GciNbStoreTravDoTrav_ function is equivalent in effect to GciStoreTravDoTrav_. However, GciNbStoreTravDoTrav_ permits the application to proceed with non-GemStone tasks while the traversals are stored, and GciStoreTravDoTrav_ does not.
See Also

GciNbClampedTrav, page 298
GciNbEnd, page 304
GciNbStoreTrav, page 321
GciStoreTravDoTrav_, page 486
GciNbStoreTravDoTravRefs_

Combine in a single function modifications to session sets, traversal of objects to the server, optional Smalltalk execution, and traversal to the client of changed objects and (optionally) the result object (non blocking).

NOTE
In previous GemStone/S 64 Bit releases, this function was named Gci NbStoreTravDoTravRefs (without the underscore).

Syntax

```c
void GciNbStoreTravDoTravRefs_(`
    const OopType * oopsNoLongerReplicated,
    int numNotReplicated,
    const OopType * oopsGcedOnClient,
    int numGced,
    GciStoreTravDoArgsSType *stdArgs,
    GciClampedTravArgsSType *ctArgs);
```

Input Arguments

- `oopsNoLongerReplicated`: An Array of objects to be removed from the PureExportSet and added to the ReferencedSet.
- `numNotReplicated`: The number of elements in `oopsNoLongerReplicated`.
- `oopsGcedOnClient`: An Array of objects to be removed from both the PureExportSet and ReferencedSet.
- `numGced`: The number of elements in `oopsGcedOnClient`.
- `stdArgs`: An instance of `GciStoreTravDoArgsSType`. For details, refer to the discussion of `GciStoreTravDo_` on page 478.
- `ctArgs`: An instance of `GciClampedTravArgsSType`. For details, see the discussion of `GciClampedTrav` on page 135.

Return Value

The `GciNbStoreTravDoTravRefs_` function, unlike `GciStoreTravDoTravRefs_`, 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 `GciStoreTravDoTravRefs_` by using the argument to `Gci NbEnd`
Description

This function allows the client to modify the PureExportSet and ReferencedSet, modify or create any number of objects on the server, execute behavior on the Gem, and return the traversal of the result object, all in a single network round-trip.

The GciNbStoreTravDoTravRefs_ function is equivalent in effect to GciStoreTravDoTravRefs_. However, GciNbStoreTravDoTravRefs_ permits the application to proceed with non-GemStone tasks while the traversals are stored, and GciStoreTravDoTravRefs_ does not.

See Also

GciNbClampedTrav, page 298
GciNbEnd, page 304
GciStoreTravDoTrav_, page 486
GciStoreTravDoTravRefs_, page 488
GciNbTraverseObjs

Traverse an array of GemStone objects (nonblocking).

Syntax

```c
void GciNbTraverseObjs(
    const OopType    theOops[ ],
    int              numOops,
    GciTravBufType*  travBuff,
    int              level);
```

Input Arguments

- `theOops` An array of OOPs representing the objects to traverse.
- `numOops` The number of elements in theOops.
- `travBuffSize` The number of bytes allocated to the traversal buffer.
- `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 Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

See Also

GciFindObjRep, page 255
GciMoreTraversal, page 293
GciNbMoreTraversal, page 314
GciNbStoreTrav, page 321
GciNewOopUsingObjRep, page 338
GciObjRepSize_, page 348
GciStoreTrav, page 478
GciTraverseObjs, page 510
GciNewByteObj

Create and initialize a new byte object.

Syntax

```c
OopType GciNewByteObj(
    OopType   aClass,
    const ByteType * value,
    int64     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(
    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(
    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

```c
OopType GciNewOop(
    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. It may not be Symbol or DoubleByteSymbol. 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. It cannot be used to create instances of Symbol or DoubleByteSymbol.

Example

```c
OopType newOop_example(void)
{
    // create a new instance of String
    OopType result = GciNewOop(OOP_CLASS_STRING);
    return result;
}
```
See Also

GciNewOops, page 335
GciNewOopUsingObjRep, page 338
GciReleaseAllOops, page 399
GciReleaseGlobalOops, page 401
GciNewOops

Create multiple new GemStone objects.

Syntax

```c
void GciNewOops(
    int numOops,
    const OopType oclass[],
    const int64 idxSize[],
    OopType result[]);
```

Input Arguments

- **numOops**: The number of new objects to be created.
- **oclass**: For each new object, the OOP of its class. This should not be the OOP of Symbol or DoubleByteSymbol.
- **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. This function cannot be used to create instances of Symbol or DoubleByteSymbol. 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:

- \( idxSize \ < \ 0 \)
- \( (idxSize + \text{number\_of\_named\_instance\_variables}) > \text{maxSmallInt} \)

Example

```c
void newOops_example(void)
{
    enum { num_objs = 3 };  
    OopType classes[num_objs];
    classes[0] = OOP_CLASS_STRING;
    classes[1] = OOP_CLASS_IDENTITY_SET;
    classes[2] = OOP_CLASS_ARRAY;
    
    int64 sizes[num_objs];
    sizes[0] = 50;
    sizes[1] = 0;  /* ignored for NSCs anyway */
    sizes[2] = 3;
    
    OopType newObjs[num_objs];
    
    GciNewOops(num_objs, classes, sizes, newObjs);
    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        printf("error category "FMT_OID" number %d, %s\n",
            errInfo.category, errInfo.number, errInfo.message);
    } else {
        printf("objIds of new objects are "FMT_OID" "FMT_OID"
            "FMT_OID"
            "
            newObjs[0], newObjs[1], newObjs[2]);
    }
}
```
See Also

GciNewOop, page 333
GciNewOopUsingObjRep, page 338
GciReleaseAllOops, page 399
GciReleaseGlobalOops, page 401
GciStoreTrav, page 478
GciNewOopUsingObjRep

Create a new GemStone object from an existing object report.

Syntax

void GciNewOopUsingObjRep(
    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 ID of the object’s security policy (hdr.objectSecurityPolicyId), 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 94.

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 53.
GciNewOopUsingObjRep provides automatic byte swizzling for Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

**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 Float or SmallFloat, 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, SmallDouble, or UndefinedObject).
Example

```c
void newOopUsingObjRep_example(void)
{
    int arrSize = 100;
    size_t bodySize = sizeof(OopType) * arrSize;
    size_t rptSize = GCI_ALIGN(sizeof(GciObjRepSType) + bodySize);
    GciObjRepSType *rpt = (GciObjRepSType*) malloc(rptSize);
    if (rpt == NULL) {
        printf("malloc failure\n");
        return;
    }
    rpt->hdr.objId = OOP_NIL; // ignored by GciNewOopUsingObjRep
    rpt->hdr.oclass = OOP_CLASS_ARRAY;
    rpt->hdr.setObjImpl(GC_FORMAT_OOP);
    rpt->hdr.segmentId = WORLD_RW_SEGMENT_ID;
    rpt->hdr.firstOffset = 1;
    rpt->hdr.namedSize = 0; // ignored by GciNewOopUsingObjRep
    rpt->hdr.setIdxSize( arrSize );
    rpt->hdr.valueBuffSize = bodySize;

    OopType *body = rpt->valueBufferOops();
    for (int i = 0; i < arrSize; i += 1) {
        body[i] = GciI32ToOop(i);
    }
    GciNewOopUsingObjRep(rpt);

    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        printf("error category "FMT_OID" number %d, %s\n",
                        errInfo.category, errInfo.number, errInfo.message);
    }
}
```

See Also

GciNewOop, page 333
GciReleaseAllOops, page 399
GciReleaseGlobalOops, page 401
GciTraverseObjs, page 510
GciNewString

Create a new String object from a C character string.

Syntax

OopType GciNewString(
    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(
    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.
GciNscIncludesOop

Determines whether the given OOP is present in the specified unordered collection.

Syntax

```c
BoolType GciNscIncludesOop(
    OopType    theNsc,
    OopType    theOop);
```

Input Arguments

- `theNsc` The unordered collection in which to search.
- `theOop` The OOP to search for.

Return Value

True if the OOP was found; false otherwise.

Description

`GciNscIncludesOop` searches the specified unordered collection to determine if it includes the specified object. It is equivalent to the GemStone Smalltalk method `UnorderedCollection >> includesIdentical:`.

Example

```c
BoolType nscIncludesOop_example(OopType nscOop, OopType anOop)
{
    if (!GciIsKindOfClass(nscOop, OOP_CLASS_IDENTITY_BAG) ) {
        printf(“first argument is not an Nsc\n”);
        return FALSE; /* error: nscOop is not an NSC */
    }

    return GciNscIncludesOop(nscOop, anOop);
}
```
See Also

GciAddOopToNsc, page 115
GciAddOopsToNsc, page 117
GciRemoveOopFromNsc, page 406
GciRemoveOopsFromNsc, page 408
GciObjExists

Determine whether or not a GemStone object exists.

Syntax

_BoolType GciObjExists(
    _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(
    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 (such as a 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 GemStone Smalltalk method Object >> in:
GciObjIsCommitted

Determine whether or not an object is committed.

Syntax

```
BoolType GciObjIsCommitted(
    OopType     oop);
```

Input Arguments

`oop` The OOP of an object.

Return Value

`GciObjIsCommitted` returns TRUE if the object is committed, FALSE otherwise.

Description

The `GciObjIsCommitted` function determines if the given object is committed or not.

See Also

GciObjExists, page 345
GciObjRepSize

Find the number of bytes in an object report.

Syntax

```c
size_t GciObjRepSize_(anObjectReport)
const GciObjRepHdrSType *anObjectReport;
```

Input Arguments


Return Value

Returns the size of the specified object report.

Description

**NOTE**

In previous GemStone/S 64 Bit releases, this function was named `GciObjRepSize` (without the underscore). Customers must ensure that the variables that receive this function’s result are large enough to accommodate a 64-bit value.

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 53.
Example

```c
void objRepSize_example(void)
{
    // Assumes topaz code for GciFetchVaryingOops example has run

    OopType rootObj = GciResolveSymbol("AllComponents", OOP_NIL);
    GciTravBufType *buf = GciTravBufType::malloc(8000);

    GciTraverseObjs(&rootObj, 1, buf, 10);
    GciObjRepHdrSType *rpt = buf->firstReportHdr();
    GciObjRepHdrSType *limit = buf->readLimitHdr();
    if (rpt < limit) {
        size_t reportSize = GciObjRepSize_(rpt);
        printf("size of first report is %ld bytes\n", reportSize);
    } else {
        printf("error, GciTraverseObjs returned empty buffer\n");
    }
}
```

See Also

GciFindObjRep, page 255
GciMoreTraversal, page 293
GciTraverseObjs, page 510
GciOldOopToNewOop

Return a GemStone/S 64 Bit v2.0 OopType corresponding to a GemStone/S 64 Bit v1.1 OOP.

Syntax

    OopType GciOldOopToNewOop(
        unsigned int       oldOop);

Input Arguments

    oldOop            The GemStone/S 64 Bit v1.1.1 OOP.

Return Value

    Returns an OopType that corresponds to the GemStone/S 64 Bit v1.1 OOP. Returns
    OOP_ILLEGAL if the argument is not a valid GemStone/S 64 Bit v1.1 OopType.

Description

    This function converts a v1.1 OOP into the equivalent v2.0 OopType. If the result is not a
    special OOP, this function does not check for the existence of the object.
    This function returns OOP_ILLEGAL if the argument is not a legal special OOP or if the
    current session is not valid.
    This function does not convert LargeIntegers to new SmallIntegers.
**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_OOP_IS_SMALL_INT, page 352
- GCI_OOP_IS_SPECIAL, page 353
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_OOP_IS_BOOL, page 351
GCI_OOP_IS_SPECIAL, page 353
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_OOP_IS_BOOL, page 351
GCI_OOP_IS_SMALL_INT, page 352
**GciOopToBool**

Convert a Boolean object to a C Boolean value.

**Syntax**

```c
BoolType GciOopToBool(
    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

```c
BoolType oopToBoolExample(OopType anObj)
{
    BoolType aBool = GciOopToBool(anObj);

    GciErrSTYPE errInfo;
    if (GciErr(&errInfo)) {
        // argument was not a Boolean
        printf("error category "FMT_OID" number %d, %s\n",
            errInfo.category, errInfo.number, errInfo.message);
        return 0;
    }
    return aBool;
}
```

See Also

GCI_BOOL_TO_OOP, page 128
GCI_OOP_TO_BOOL

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

Syntax

GCI_OOP_TO_BOOL(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.

Provided for compatibility only. New code should use GciOopToBool (page 354). For the definition of GCI_OOP_TO_BOOL, see $GEMSTONE/include/gcicmn.ht

See Also

GCI_BOOL_TO_OOP, page 128
GciOopToChar16

Convert a Character object to a 16-bit C character value.

Syntax

```c
unsigned int GciOopToChar16(
    OopType           theObject );
```

Input Arguments

- `theObject` The OOP of the Character or JisCharacter object to be translated into a 16-bit C character value.

Return Value

Returns the 16-bit 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 16-bit C character value.

See Also

GciOopToChar32, page 358
GciOopToChr, page 359
**GciOopToChar32**

Convert a Character object to a 32-bit C character value.

**Syntax**

```c
unsigned int GciOopToChar32(
    OopType theObject );
```

**Input Arguments**

- `theObject` The OOP of the Character or JisCharacter object to be translated into a 32-bit C character value.

**Return Value**

Returns the 32-bit 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 32-bit C character value.

**See Also**

- GciOopToChar16, page 357
- GciOopToChr, page 359
GciOopToChr

Convert a Character object to a C character value.

Syntax

```c
char GciOopToChr(
    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

```c
char oopToChar_example(OopType arg)
{
    char aChar = GciOopToChr(arg);

    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        // argument was not a Character
        printf("error category "FMT_OID" number %d, %s\n",
            errInfo.category, errInfo.number, errInfo.message);
        return 0;
    }
    return aChar;
}
```

See Also

GCI_CHR_TO_OOP, page 134
GciOopToChar16, page 357
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

Provided for compatibility only. New code should use GciOopToChr or GciOopToChar16.

See Also

GciOopToChar16, page 357
GciOopToChr, page 359
GciOopToFlt

Convert a SmallDouble, Float, or SmallFloat object to a C double.

Syntax

double GciOopToFlt(  
    OopType theObject );

Input Arguments

theObject      The OOP of the SmallDouble, Float, or SmallFloat 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

- 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

double oopToFlt_example(OopType arg)
{
    double d = GciOopToFlt(arg);

    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        // argument was not a Float, SmallFloat or SmallDouble
        printf("error category "FMT_OID" number %d, %s\n",
            errInfo.category, errInfo.number, errInfo.message);
        return 0.0 ;
    }
    return d;
}

See Also

GciFltToOop, page 258
GciGetFreeOopsEncoded, page 264
GciOopToI32

Convert a GemStone object to a C 32-bit integer value.

Syntax

```c
int GciOopToI32(
    OopType       theObject);
```

Input Arguments

`theObject` The OOP of the object to be translated into a C 32-bit integer value.

Return Value

The `GciOopToI32` function returns the C 32-bit integer value that is equivalent to the value of `theObject`.

Description

The `GciOopToI32` function translates a GemStone object into the equivalent C 32-bit integer value. The GemStone object must be a SmallInteger within the range of C integers. Otherwise, `GciOopToI32` generates an error.

See Also

GciOopToI32_, page 365
GciOopToI64, page 366
GciOopToI64_, page 367
GciOopToI32_

Convert a GemStone object to a C 32-bit integer value, with error handling.

Syntax

```c
int GciOopToI32_(
    OopType       theObject,
    BoolType *    error);
```

Input Arguments

- **theObject**: The OOP of the object to be translated into a C 32-bit integer value.

Result Arguments

- **error**: TRUE if `theObject` does not fit in the result type or is not an Integer. Otherwise unchanged.

Return Value

The `GciOopToI32_` function returns the C 32-bit integer value that is equivalent to the value of `theObject`.

Description

The `GciOopToI32_` function translates a GemStone object into the equivalent C 32-bit integer value. The GemStone object must be a SmallInteger within the range of C integers. `GciOopToI32_` provides for error handling if `theObject` does not fit in the result type or is not an Integer. Compare with `GciOopToI32`, which does not provide for error handling.

See Also

- GciOopToI32, page 364
- GciOopToI64, page 366
- GciOopToI64_, page 367
**GciOopToI64**

Convert a GemStone object to a C 64-bit integer value.

**Syntax**

```c
int64 GciOopToI64(
    OopType theObject );
```

**Input Arguments**

- **theObject** The OOP of the object to be translated into a C 64-bit integer value.

**Return Value**

The **GciOopToI64** function returns the C int64_t value that is equivalent to the value of **theObject**.

**Description**

**NOTE**

In previous GemStone/S 64 Bit releases, this function was named **GciOopToInt64**. Customers must ensure that the variables that receive this function’s result are large enough to accommodate an int64 value.

The **GciOopToI64** function translates a GemStone object into the equivalent C 64-bit integer value.

The object identified by **theObject** must be a SmallInteger, a LargePositiveInteger, or a LargeNegativeInteger. If the object is not one of these kinds, **GciOopToI64** generates an error.

**See Also**

- GciOopToI32, page 364
- GciOopToI32_, page 365
- GciOopToI64_, page 367
**GciOopToI64**

Convert a GemStone object to a C 64-bit integer value, with error handling.

**Syntax**

```c
int64 GciOopToI64_(
    OopType             theObject,
    BoolType *          error);
```

**Input Arguments**

*theObject*  
The OOP of the object to be translated into a C 64-bit integer value.

**Result Arguments**

*error*  
TRUE if *theObject* does not fit in the result type or is not an Integer. Otherwise unchanged.

**Return Value**

The *GciOopToI64_* function returns the C int64_t value that is equivalent to the value of *theObject*.

**Description**

The *GciOopToI64_* function translates a GemStone object into the equivalent C 64-bit integer (int64_t) value. The GemStone object must be a SmallInteger, a LargePositiveInteger, or a LargeNegativeInteger. *GciOopToI64_* provides for error handling if *theObject* does not fit in the result type or is not an Integer. Compare with *GciOopToI64*, which does not provide for error handling.

**See Also**

*GciOopToI32*, page 364  
*GciOopToI32_*, page 365  
*GciOopToI64*, page 366
GciPathToStr

Convert a path representation from numeric to string.

This function is deprecated and may be removed from future releases.

Syntax

```c
BoolType GciPathToStr(
    OopType       aClass,
    const int    path[ ],
    int           pathSize,
    int64         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 237.

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 53.

Restrictions

Note that GciPathToStr can convert a numeric path only if:

• The instance variables of the specified Smalltalk class (aClass) are guaranteed to be the same valid for all instances along all paths.

• 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.

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 248.

```c
void pathToString_example(void)
{
    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr("AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);

    int ofs = 3;  // offset of cost instVar
    int pathSize = 1;
    char result[1024];
    GciPathToStr(GciFetchClass(aComponent), &ofs, pathSize, sizeof(result), result);

    printf("result = %s\n", result);
}
```

See Also

GciFetchPaths, page 237
GciStorePaths, page 471
GciStrToPath, page 497
GciPerform

Send a message to a GemStone object.

Syntax

```
OopType GciPerform(
    OopType receiver,
    const char selector[],
    const OopType args[],
    int numArgs);
```

```
OopType GciPerform_(
    OopType receiver,
    const char selector[],
    const OopType args[],
    int numArgs,
    ushort environmentId);
```

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.
- **environmentId**: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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 32.

See Also

GciContinue, page 154
GciErr, page 189
GciExecute, page 191
GciNbContinue, page 302
GciNbExecute, page 306
GciNbPerform, page 315
GciNbPerformNoDebug, page 317
GciPerformNoDebug, page 373
GciPerformSymDbg, page 375
**GciPerformNoDebug**

Send a message to a GemStone object, and temporarily disable debugging.

**Syntax**

```c
OopType GciPerformNoDebug(
    OopType    receiver,
    const char selector[ ],
    const OopType args[ ],
    int        numArgs,
    int        flags);

OopType GciPerformNoDebug_(
    OopType    receiver,
    const char selector[ ],
    const OopType args[ ],
    int        numArgs,
    int        flags,
    ushort     environmentId);
```

**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.

- **environmentId**
  The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.
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 may be 0 for default behavior, or can be given by using one or more of these 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.
- GCI_PERFORM_FLAG_SINGLE_STEP places a single-step breakpoint at the start of the method to be performed, and then executes to hit that breakpoint.

These flags can either be used alone or logically “or”ed together.

See Also

GciContinue, page 154
GciErr, page 189
GciExecute, page 191
GciNbContinue, page 302
GciNbExecute, page 306
GciNbPerform, page 315
GciNbPerformNoDebug, page 317
GciPerform, page 371
GciPerformSymDbg, page 375
GciPerformSymDbg

Send a message to a GemStone object, using a String object as a selector.

Syntax

```c
OopType GciPerformSymDbg(
    OopType receiver,
    OopType selector,
    const OopType args[],
    int numArgs,
    int flags);
```

```c
OopType GciPerformSymDbg_(
    OopType receiver,
    OopType selector,
    const OopType args[],
    int numArgs,
    int flags,
    ushort environmentId);
```

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.
- flags: Flags to disable or permit asynchronous events and debugging in Smalltalk.
- environmentId: The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.
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 GciPerformSymDbg takes an OOP as its selector instead of a C string.

See Also

GciContinue, page 154
GciErr, page 189
GciExecute, page 191
GciPerform, page 371
GciPerformNoDebug, page 373
**GciPerformTrav**

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

**Syntax**

```c
BoolType GciPerformTrav(
    OopType  receiver,
    const char * selector,
    const OopType * args,
    int       numArgs,
    GciClampedTravArgsSType *travArgs);

BoolType GciPerformTrav_((
    OopType  receiver,
    const char * selector,
    const OopType * args,
    int       numArgs,
    GciClampedTravArgsSType *travArgs,
    ushort   environmentId);
```

**Input Arguments**

- **receiver** The OOP of the receiver of the message.
- **selector** A pointer to a character collection 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 an instance of GciClampedTravArgsSType. See the GciClampedTrav function (page 135) for documentation of the fields in travArgs.
- **environmentId** The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.
Result Arguments

The result of the GciPerform is the first object in the resulting travBuff 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 371
GciClampedTrav, page 135
GciPerformTraverse

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

Syntax

```c
BoolType GciPerformTraverse(
    OopType receiver,
    const char selector[],
    const OopType args[],
    int numArgs,
    GciTravBufType * travBuff,
    int level);

BoolType GciPerformTraverse_(
    OopType receiver,
    const char selector[],
    const OopType args[],
    int numArgs,
    GciTravBufType * travBuff,
    int level,
    ushort environmentId);
```
Input Arguments

receiver  The OOP of the receiver of the message.
selector  A pointer to a character collection 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.
environmentId  The compilation environment for method lookup. Used with Ruby applications, but not with Smalltalk applications. For details, see “environmentId” on page 101.

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.
Consider the following function call:

```c
BoolType performTrav_1(void)
{
    OopType receiver = GciResolveSymbol("AllComponents", OOP_NIL);
    OopType arg = GciI32ToOop(1);
    GciTravBufType *buf = GciTravBufType::malloc(8000);

    BoolType atEnd = GciPerformTraverse(receiver, "at:", &arg, 1, buf, 10);
    return atEnd;
}
```

It is equivalent to the following code:

```c
BoolType performTrav_2(void)
{
    OopType receiver = GciResolveSymbol("AllComponents", OOP_NIL);
    OopType arg = GciI32ToOop(1);
    OopType obj = GciPerform(receiver, "at:", &arg, 1);
    GciTravBufType *buf = GciTravBufType::malloc(8000);
    BoolType atEnd = GciTraverseObjs(&obj, 1, buf, 10);
    return atEnd;
}
```

_GciPerformTraverse_ provides automatic byte swizzling for Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

**See Also**

- GciContinue, page 154
- GciErr, page 189
- GciExecute, page 191
- GciFindObjRep, page 255
- GciMoreTraversal, page 293
- GciNewOopUsingObjRep, page 338
- GciObjRepSize_, page 348
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciPerform</td>
<td>371</td>
</tr>
<tr>
<td>GciPerformNoDebug</td>
<td>373</td>
</tr>
<tr>
<td>GciPerformSymDbg</td>
<td>375</td>
</tr>
<tr>
<td>GciStoreTrav</td>
<td>478</td>
</tr>
<tr>
<td>GciTraverseObjs</td>
<td>510</td>
</tr>
</tbody>
</table>
GciPointerToByteArray

Given a C pointer, return a SmallInteger or ByteArray containing the value of the pointer.

Syntax

OopType GciPointerToByteArray(
       void *  
       pointer);

Input Arguments

pointer A C pointer.

Return Value

Returns a GemStone SmallInteger or ByteArray containing the value of the pointer.

If the argument is a 64-bit pointer aligned on an 8-byte boundary, or is a 32-bit pointer, the
result is a SmallInteger. Otherwise, the result is a ByteArray.

Description

The result has a machine-dependent byte order and is not intended to be committed.

See Also

GciByteArrayToPointer, page 130
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 GciLongJmp 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 185
GciErr, page 189
**GciPopErrJump**

Discard a previously saved error jump buffer.

**Syntax**

```c
void GciPopErrJump(
    GciJmpBufSType * jumpBuffer);
```

**Input Arguments**

- `jumpBuffer`: A pointer to a jump buffer specified in an earlier call to `GciPushErrJump`.

**Description**

This function discards one or more jump buffers that were saved with earlier calls to `GciPushErrJump`. 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`. See the following example.
Example

```c
void popErr_example(void) {
    GciJmpBufSType jumpBuff1, jumpBuff2, jumpBuff3, jumpBuff4;
    GciPushErrJump(&jumpBuff1);
    GciPushErrJump(&jumpBuff2);
    GciPushErrJump(&jumpBuff3);
    GciPushErrJump(&jumpBuff4);
    GciPopErrJump(&jumpBuff1); /* pops buffers 1-4 */
}
```

See Also

GciErr, page 189
GciPushErrJump, page 391
GciSetErrJump, page 427
GciLongJmp, page 292
**GciProcessDeferredUpdates_**

Process deferred updates to objects that do not allow direct structural update.

**Syntax**

```c
int64 GciProcessDeferredUpdates_( )
```

**Return Value**

Returns the number of objects that had deferred updates.

**Description**

**NOTE**

*In previous GemStone/S 64 Bit releases, this function was named GciProcessDeferredUpdates (without the underscore). Customers must ensure that the variables that receive this function’s result are large enough to accommodate an int64 value.*

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 one 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 478
GciProduct

Return an 8-bit unsigned integer that indicates the GemStone/S product.

Syntax

unsigned char GciProduct();

Return Value

Returns an 8-bit unsigned integer indicating the GemStone/S product to which the client library belongs. Currently-defined integers are:

- 1 — GemStone/S
- 2 — GemStone/S 2G
- 3 — GemStone/S 64 Bit

Description

GciProduct allows a GemBuilder client to determine which GemStone/S product it is talking to. Combined with GciVersion, it allows the client to adapt to differences between GemBuilder features across different products and versions.

Although GciProduct can be used by any GemBuilder client, it is specifically provided for the use of GemBuilder for Smalltalk.

Future products in the GemStone/S line will be assigned integers beginning with 4.

The integer zero is reserved, and will never be assigned to any product.

See Also

GciVersion, page 519
**GciPushErrJump**

Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.

**Syntax**

```c
void GciPushErrJump(
    GciJmpBufSType * jumpBuffer );
```

**Result Arguments**

- `jumpBuffer` A pointer to a jump buffer, as described below. The `jumpBuffer` must have been initialized by passing it as the argument to the macro `Gci_SETJMP`.

**Description**

Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.

This function allows your application program to take advantage of the `setjmp/longjmp` style of 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`.

Rather than using `setjmp` and `longjmp` directly, this style of error handling in GemBuilder requires you to use `Gci_SETJMP` and `GciLongJmp`.

When your program calls `Gci_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 implementation calls `GciLongJmp` using the buffer currently at the top of GemBuilder’s error jump stack, and pops that buffer from the stack.

For functions with local error recovery, your program can call `GciSetErrJump` to temporarily disable the `GciLongJmp` mechanism (and to re-enable it afterwards).
Whenever the jump stack is empty, the application must use \texttt{GciErr} to poll for any GemBuilder errors.

\textbf{Example}

For an example of how \texttt{GciPushErrJump} is used, see the \texttt{GciPopErrJump} function on page 386.

\textbf{See Also}

\begin{itemize}
  \item \texttt{GciErr}, page 189
  \item \texttt{GciPopErrJump}, page 386
  \item \texttt{GciSetErrJump}, page 427
\end{itemize}
GciRaiseException

Signal an error, synchronously, within a user action.

Syntax

```c
void GciRaiseException(  
    const 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 signaling mechanism, causing control to return to Smalltalk.

This function has no effect when executed outside of a user action.
GciReadSharedCounter

Lock and fetch the value of a shared counter.

Syntax

```c
BoolType GciReadSharedCounter(
    int counterIdx,
    int64_t * value);
```

Input Arguments

`counterIdx` The offset into the shared counters array of the value to fetch.

Result Arguments

`value` Pointer to a value that indicates the value at this shared counter.

Return Value

Returns a C Boolean value indicating whether the value was successfully read. Returns TRUE if successful, FALSE if an error occurred.

Description

Lock the shared counter indicated by `counterIdx`, and fetch its value. The contents of the `value` pointer will be set to the value of the shared counter.

Not supported for remote GCI interfaces.

See Also

GciFetchNumSharedCounters, page 225
GciDecSharedCounter, page 172
GciIncSharedCounter, page 271
GciSetSharedCounter, page 437
GciReadSharedCounterNoLock, page 395
GciFetchSharedCounterValuesNoLock, page 244
GciReadSharedCounterNoLock

Fetch the value of a shared counter without locking it.

Syntax

```
BoolType GciReadSharedCounterNoLock(
    int counterIdx,
    int64_t * value);
```

Input Arguments

- `counterIdx` The offset into the shared counters array of the value to fetch.

Result Arguments

- `value` Pointer to a value at this shared counter.

Return Value

Returns a C Boolean value indicating whether the value was successfully read. Returns TRUE if successful, FALSE if an error occurred.

Description

Fetch the value of the shared counter indicated by `counterIdx`. The contents of the `value` pointer will be set to the value of the shared counter. This function is faster than GciReadSharedCounter, but may be less accurate.

Not supported for remote GCI interfaces.

See Also

- GciFetchNumSharedCounters, page 225
- GciDecSharedCounter, page 172
- GciIncSharedCounter, page 271
- GciSetSharedCounter, page 437
GciReadSharedCounter, page 394
GciFetchSharedCounterValuesNoLock, page 244
GciRealloc

Reallocates memory.

Syntax

```c
void* GciRealloc(
    void * p,
    size_t length,
    int lineNumber,
    const char * fileName,
);
```

Description

Return NULL if the underlying realloc() fails.
GciReleaseAllGlobalOops

Remove all OOPS from the PureExportSet, making these objects eligible for garbage collection.

Syntax

void GciReleaseAllGlobalOops( )

Description

The GciReleaseAllGlobalOops function removes all OOPs from the PureExportSet, thus permitting GemStone to consider removing them as a result of garbage collection. Objects that are referenced from persistent objects are not removed during garbage collection, even if they are not in PureExportSet. If invoked from a user action, this function does not affect the user action’s export set.

GciReleaseAllGlobalOops is similar to GciReleaseAllOops, with the exception that OOPs are removed from the PureExportSet regardless of whether it is called from within a user action or not.

The GciSaveGlobalObjs or GciSaveGlobalObjs functions may be used to make objects ineligible for garbage collection. Note that results of the GciNew..., GciCreate..., GciPerform..., and GciExecute... functions are automatically added to the PureExportSet. You must release those objects explicitly if they are to be eligible for garbage collection.

CAUTION

Before releasing all objects, be sure that you do not need to retain any of them for any reason.

See Also

Garbage Collection, page 49
GciReleaseAllOops, page 399
GciReleaseGlobalOops, page 401
GciSaveGlobalObjs, page 421
GciSaveObjs, page 422
GciReleaseAllOops

Remove all OOPS from the PureExportSet, or if in a user action, from the user action's export set, making these objects eligible for garbage collection.

Syntax

void GciReleaseAllOops()

Description

The GciReleaseAllOops function removes all OOPs from the applicable export set, thus permitting GemStone to consider removing them as a result of garbage collection. If called from within a user action, GciReleaseAllOops releases only those objects that have been saved since the beginning of the user action and are therefore in the user action's export set. If not called from within a user action, GciReleaseAllOops removes all OOPs from the PureExportSet. To remove all objects from the PureExportSet, regardless of user action context, use GciReleaseAllGlobalOops.

Objects that are referenced by persistent objects are not removed during garbage collection, even if they are not in an export set. It is typical usage to call GciReleaseAllOops after successfully committing a transaction.

The GciSaveObjs or GciSaveGlobalObjs functions may be used to make objects ineligible for garbage collection. Note that results of the GciNew..., GciCreate..., GciPerform..., and GciExecute... functions are automatically ineligible. You must release those objects explicitly if they are to be eligible.

CAUTION

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 49
GciReleaseAllGlobalOops, page 398
GciReleaseGlobalOops, page 401
GciReleaseOops, page 402
GciSaveGlobalObjs, page 421
GciSaveObjs, page 422
GciReleaseAllTrackedOops

Clear the GciTrackedObjs set, making all tracked OOPs eligible for garbage collection.

Syntax

```c
void GciReleaseAllTrackedOops();
```

Description

The **GciReleaseAllTrackedOops** function removes all OOPs from the user session’s GciTrackedObjs set, thus making them eligible to be garbage collected. This function does not affect the export sets; objects that are also in an export set will remain protected from garbage collection.

**CAUTION**

Before releasing any of your objects, be sure that you do not need to retain them for any reason.

See Also

- GciHiddenSetIncludesOop, page 268
- GciReleaseAllGlobalOops, page 398
- GciReleaseAllOops, page 399
- GciReleaseTrackedOops, page 405
- GciSaveAndTrackObjs, page 419
GciReleaseGlobalOops

Remove an array of GemStone OOPs from the PureExportSet, making them eligible for garbage collection.

Syntax

```c
void GciReleaseGlobalOops(
    const OopType theOops[],
    int 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 `GciReleaseGlobalOops` function removes the specified OOPs from the PureExportSet, thus making them eligible to be garbage collected.

This function differs from `GciReleaseOops` in that it operates the same if invoked from within a user action or not.

The `GciSaveObjs` or `GciSaveGlobalObjs` functions may be used to make objects ineligible for garbage collection. Note that results of the `GciNew...`, `GciCreate...`, `GciPerform...`, and `GciExecute...` functions are automatically ineligible. You must release those objects explicitly if they are to be eligible.

**CAUTION**

Before releasing any of your objects, be sure that you do not need to retain them for any reason.

See Also

“Garbage Collection” on page 49
GciReleaseAllGlobalOops, page 398
GciReleaseOops, page 402
GciSaveGlobalObjs, page 421
GciReleaseOops

Remove an array of GemStone OOPs from the PureExportSet, or if in a user action, remove them from the user action’s export set, making them eligible for garbage collection.

Syntax

```c
void GciReleaseOops(
    const OopType theOops[],
    int 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 applicable export set, thus making them eligible to be garbage collected. If invoked from within a user action, the specified OOPs are removed from the user action’s export set, otherwise the OOPs are removed from the PureExportSet.

To remove OOPs from the PureExportSet, regardless of user action context, use **GciReleaseGlobalOops**.

The **GciSaveObjs** or **GciSaveGlobalObjs** functions may be used to make objects ineligible for garbage collection. Note that results of the **GciNew...**, **GciCreate...**, **GciPerform...**, and **GciExecute...** functions are automatically ineligible. You must release those objects explicitly if they are to be eligible.

**CAUTION**

Before releasing any of your objects, be sure that you do not need to retain them for any reason.
Example

```c
void releaseOops_example(void)
{
    // assumes topaz code for GciFetchVaryingOops example has run.
    OopType oClass = GciResolveSymbol("Component", OOP_NIL);

    OopType namedIvs[3];
    namedIvs[0] = GciI32ToOop(5699); // a SmallInteger, don't need to release
    namedIvs[2] = GciFltToOop(9.0e6); // a Float or SmallDouble

    OopType newComp = GciNewOop(oClass);
    GciStoreOops(newComp, 1, namedIvs, 3);

    OopType oColl = GciResolveSymbol("AllComponents", OOP_NIL);
    GciAddOopToNsc(oColl, newComp); // new objects now reachable from AllComponents

    // release newly created objects so that if aComp is removed from
    // AllComponents by other application code, these new objects can
    // be garbage collected.
    OopType releaseBuf[3];
    releaseBuf[0] = namedIvs[1]; // a String
    releaseBuf[1] = namedIvs[1]; // might be a Float
    releaseBuf[2] = newComp; // a Component
    GciReleaseOops(releaseBuf, 3);
}
```

See Also

"Garbage Collection" on page 49
GciReleaseAllGlobalOops, page 398
GciReleaseAllOops, page 399
GciReleaseGlobalOops, page 401
GciSaveGlobalObjs, page 421
GciSaveObjs, page 422
GciReleaseTrackedOops

Remove an array of OOPs from the GciTrackedObjs set, making them eligible for garbage collection.

Syntax

```c
void GciReleaseTrackedOops(
    const OopType theOops[],
    int 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 `GciReleaseTrackedOops` function removes the specified OOPs from the user session’s GciTrackedObjs set, thus making them eligible to be garbage collected. This function does not affect the export sets; objects that also appear in an export set will remain protected from garbage collection.

CAUTION

Before releasing any of your objects, be sure that you do not need to retain them for any reason.

See Also

- GciHiddenSetIncludesOop, page 268
- GciReleaseAllTrackedOops, page 400
- GciSaveAndTrackObjs, page 419
- GciTrackedObjsInit, page 509
GciRemoveOopFromNsc

Remove an OOP from an NSC.

Syntax

BoolType GciRemoveOopFromNsc(
    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

BOOLTYPE removeOop_example(void)
{
    // assumes topaz code for GciFetchVaryingOop has run
    OOPTYPE aComponent = GciExecuteStr(
        "AllComponents detect:[iji partNumber = 1234]", OOP_NIL);

    OOPTYPE aColl = GciResolveSymbol("AllComponents", OOP_NIL);
    BOOLTYPE wasPresent = GciRemoveOopFromNsc(aColl, aComponent);
    GciReleaseOops(&aComponent, 1); // release because it was a result
    // from an execute
    return wasPresent;
}

See Also

GciAddOopToNsc, page 115
GciAddOopsToNsc, page 117
GciNscIncludesOop, page 343
GciRemoveOopsFromNsc, page 408
**GciRemoveOopsFromNsc**

Remove one or more OOPs from an NSC.

**Syntax**

```c
BoolType GciRemoveOopsFromNsc(
    OopType theNsc,
    const OopType theOops[ ],
    int 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

```c
BoolType removeOops_example(void)
{
    // assumes topaz code for GciFetchVaryingOop has run
    OopType subColl = GciExecuteStr(
        "AllComponents select:[i|i partNumber > 1000 ]", OOP_NIL);

    OopType buf[10];
    int numRet = GciFetchVaryingOops(subColl, 1, buf, 10);
    // buf contains at most 10 components with partNumber > 1000.

    OopType aColl = GciResolveSymbol("AllComponents", OOP_NIL);
    BoolType allPresent = GciRemoveOopsFromNsc(aColl, buf, numRet);

    GciReleaseOops(&subColl, 1); // release because it was result of
    // an execute
    return allPresent;
}
```

See Also

- GciAddOopToNsc, page 115
- GciAddOopsToNsc, page 117
- GciNscIncludesOop, page 343
- GciRemoveOopFromNsc, page 406
GciReplaceOops

Replace all instance variables in a GemStone object.

Syntax

```c
void GciReplaceOops(
    OopType theObj,
    const OopType theOops[],
    int 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 412
GciStoreIdxOops, page 456
GciStoreNamedOops, page 462
GciStoreOops, page 468
GciReplaceVaryingOops

Replace all unnamed instance variables in an NSC object.

Syntax

```c
void GciReplaceVaryingOops(
    OopType theNsc,
    const OopType theOops[],
    int 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 410
- GciStoreIdxOops, page 456
- GciStoreNamedOops, page 462
- GciStoreOops, page 468
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(
    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 414
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(
    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 413
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 54 for more information.

See Also

  GciRtlLoad, page 416
  GciRtlUnload, page 418
GciRtlLoad

Load a GemBuilder library.

Syntax

```c
BoolType GciRtlLoad(
    BoolType useRpc,
    const char * path,
    char errBuf[],
    size_t 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 return 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 54 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 415
GciRtlUnload, page 418
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 54 for more information.

See Also

GciRtlLoad, page 416
GciRtlIsLoaded, page 415
GciSaveAndTrackObjs

Add objects to GemStone’s internal GciTrackedObjs set to prevent them from being garbage collected.

Syntax

```c
void GciSaveAndTrackObjs(
    const OopType theOops[ ],
    int numOops );
```

Input Arguments

- `theOops` An array of OOPs.
- `numOops` The number of elements in theOops.

Description

The `GciSaveAndTrackObjs` function adds the specified OOPS to GemStone’s GciTrackedObjs set. This prevents the GemStone garbage collector from causing the objects to disappear during a session if they become unreferenced, and enables changes to these objects to show up in the TrackedDirtyObjs set.

This function does not cause the objects to be referenced from a permanent object; there is no guarantee that they will be saved to disk at commit.

The results of `GciNew...`, `GciCreate...`, `GciSend...`, `GciPerform...`, and `GciExecute...` calls are automatically added to the export set, which also prevents them from being garbage collected.

This function may only be called after `GciTrackedObjsInit` has been executed.

You can use `GciReleaseTrackedOops` or `GciReleaseAllTrackedOops` calls to cancel the effect of a `GciSaveAndTrackObjs` call, thereby making objects eligible for garbage collection. Objects that have been added to the GciTrackedObjs set and have been modified can be retrieved using `GciTrackedDirtyObjs`, `GciDirtySaveObjs`, or `GciTrackedObjsFetchAllDirty`. 
See Also

GciHiddenSetIncludesOop, page 268
GciDirtySaveObjs, page 178
GciDirtyTrackedObjs, page 180
GciReleaseAllTrackedOops, page 400
GciReleaseTrackedOops, page 405
GciTrackedObjsInit, page 509
GciTrackedObjsFetchAllDirty, page 507
GciSaveGlobalObjs

Add an array of OOPs to the PureExportSet, making them ineligible for garbage collection.

Syntax

```c
void GciSaveGlobalObjs(
    const OopType    theOops[],
    int              numOops);
```

Input Arguments

- `theOops` - An array of OOPs.
- `numOops` - The number of elements in `theOops`.

Description

The `GciSaveGlobalObjs` function places the specified OOPs in the PureExportSet, thus preventing GemStone from removing them as a result of garbage collection. `GciSaveGlobalObjs` can add any OOP to the PureExportSet. It differs from `GciSaveObjs` in that OOPs are placed in the PureExportSet regardless of user action context.

The `GciSaveGlobalObjs` 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...`, `GciPerform...`, `GciExecute...`, and `GciResolve...` 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 49
- `GciReleaseAllGlobalOops`, page 398
- `GciReleaseAllOops`, page 399
- `GciReleaseGlobalOops`, page 401
- `GciReleaseOops`, page 402
- `GciSaveObjs`, page 422
**GciSaveObjs**

Add an array of OOPs to the PureExportSet, or if in a user action to the user action’s export set, making them ineligible for garbage collection.

**Syntax**

```c
void GciSaveObjs(
    const OopType theOops[],
    int 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 applicable export set, thus preventing GemStone from removing them as a result of garbage collection. If invoked from within a user action, the OOPs are added to the user action’s export set; otherwise the OOPs are added to the PureExportSet. To add OOPs to the PureExportSet, regardless of the user action context, use `GciSaveGlobalObjs`. `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...`, `GciPerform...`, `GciExecute...`, and `GciResolve...` 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 49
- `GciReleaseGlobalOops`, page 401
- `GciReleaseOops`, page 402
- `GciSaveGlobalObjs`, page 421
GciServerIsBigEndian

Determine whether or not the server process is big-endian.

Syntax

_BoolType GciServerIsBigEndian();

Return Value

Returns TRUE if the session is RPC and the server process is big-endian, or if the session is linked and this process is big-endian. Returns FALSE otherwise.

Description

This function determines whether the server process is big-endian. If the current session is invalid, this generates an error.
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.
GciSetCacheName_

Set the name that a linked application will be known by in the shared cache.

Syntax

```c
BoolType GciSetCacheName_(
    const char * name );
```

Input Arguments

- `name`: The processName reported by `System cacheStatistics`.

Return Value

Returns FALSE if called before GciInit and GciIsRemote returns FALSE.

Description

This function sets the name that a linked application will be known by in the shared cache. This function has no effect if GciIsRemote returns TRUE.
GciSetDynLib

Swap the byte order of an array of uint.

Syntax

```c
void GciSetDynLib(
    void * handle);
```

Description

Used by the topaz.c main program to save the result of dlopen() which loaded the GCI shared library.
GciSetErrJump

Enable or disable the current error handler.

Syntax

BoolType GciSetErrJump(
    BoolType aBoolean);

Input Arguments

aBoolean TRUE enables error jumps to the execution environment saved by the most recent GciPushErrJump; 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 189 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

void setErrJump_example(void)
{
    GciJmpBufSType jumpBuf1;
    GciPushErrJump(&jumpBuf1);
    if (Gci_SETJMP(&jumpBuf1)) {
        GciErrSType errInfo;
        if (GciErr(&errInfo)) {
            printf("LONGJMP, error category "FMT_OID" number %d, %s\n",
                    errInfo.category, errInfo.number, errInfo.message);
        } else {
            printf("GCI longjmp, but no error found\n"); // should not happen
        }
        GciPopErrJump(&jumpBuf1);
        return;
    }
    BoolType prevVal = GciSetErrJump(FALSE); // disable error jumps
    printf("error jumps previously %s\n", prevVal ? "enabled" : "disabled");
    OopType oRcvr = GciI32ToOop(3);
    GciPerform(oRcvr, "frob", NULL, 0); // expect does-not-understand error
    GciErrSType errInfo;
    if (GciErr(&errInfo)) {
        printf("error category "FMT_OID" number %d, %s\n",
                errInfo.category, errInfo.number, errInfo.message);
    } else {
        printf("expected error but found none\n");
    }
    GciSetErrJump(TRUE);
    GciPerform(oRcvr, "frob", NULL, 0); // expect a longjmp
    printf("GCI longjmp did not happen\n"); // should not reach here
}

VMware, Inc.
June 2011
See Also

GciErr, page 189
GciPopErrJump, page 386
GciPushErrJump, page 391
GciSetHaltOnError

Halt the current session when a specified error occurs.

Syntax

```c
int GciSetHaltOnError(   
    int               errNum );
```

Input Arguments

`errNum` When this error occurs, halt the current session.

Return Value

Returns the previous error number on which the session was to halt.

Description

The `GciSetHaltOnError` function causes the current session to halt for internal debugging when the specified GemBuilder error occurs. When `errNum` is zero, halt on error is disabled.

See Also

GciErr, page 189
Gci_SETJMP

(MACRO) Save a jump buffer in GemBuilder’s error jump stack.

Syntax

```c
void Gci_SETJMP(
    GciJmpBufSType *  jumpBuffer);
```

Input Arguments

`jumpBuffer` A pointer to a jump buffer.

Description

When your program calls this macro, the context of the C environment is saved in a jump buffer that you designate. GemBuilder maintains a stack of up to 20 error jump buffers. Except for the difference in argument type, the semantics of this function are the same as for setjmp() on Solaris and _setjmp() on HP-UX.

See Also

GciErr, page 189
GciLongJmp, page 292
GciPopErrJump, page 386
GciPushErrJump, page 391
GciSetErrJump, page 427
GciSetNet

Set network parameters for connecting the user to the Gem and Stone processes.

Syntax

```c
void GciSetNet(
    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 the System Administration Guide for GemStone/S 64 Bit for complete information on NRS Syntax and 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 “gs64stone” on node “lichen” could be described in a network resource string as:

```
!@lichen!gs64stone
```
A Stone of the same name that is running on the same machine as the application could be described in shortened form simply as:

gs64stone

*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 the Gem is a separate process specific to that login session. 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:

```
!@lichen!gemnetobject
```

*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

For an example of how GciSetNet is used, see the GciLogin function on page 289.

See Also

“GciLogin” on page 289
GciSetSessionId

Set an active session to be the current one.

Syntax

```c
void GciSetSessionId(
    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

```c
void setSession_example(void)
{
    // assume topaz code for GciFetchVaryingOop has run
    // see GciLogin for login_example()
    if (! login_example())
        return;
    GciSessionIdType sess1 = GciGetSessionId();

    if (! login_example())
        return;
    GciSessionIdType sess2 = GciGetSessionId();

    { OopType aColl = GciResolveSymbol("AllComponents", OOP_NIL);
        OopType aComponent = GciExecuteStr(
            "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
        GciRemoveOopFromNsc(aColl, aComponent);
        GciReleaseOops(&aComponent, 1);
        printf("session %d, size after removal "FMT_I64"\n",
            sess2, GciFetchVaryingSize_(aColl));
    }
    // other session will still see the original size before removal
    // because it has an independent transactional view of the
    // repository.
    GciSetSessionId(sess1);
    { OopType aColl = GciResolveSymbol("AllComponents", OOP_NIL);
        printf("session %d, current size "FMT_I64"\n",
            sess1, GciFetchVaryingSize_(aColl));
    }
}
```

See Also

GciGetSessionId, page 265
GciLogin, page 289
GciSetSharedCounter

Set the value of a shared counter.

Syntax

```c
BoolType GciSetSharedCounter(
    int    counterIdx,
    int64_t *    value);
```

Input Arguments

- `counterIdx`: The offset into the shared counters array of the value to modify.
- `value`: Pointer to a value that containing the new value for this shared counter.

Return Value

Returns a C Boolean value indicating whether the value was successfully changed. Returns TRUE if the modification succeeded, FALSE if it failed.

Description

Set the value of the shared counter indicated by `counterIdx`. The contents of the `value` pointer indicate the new value of the shared counter.

Not supported for remote GCI interfaces.

See Also

- GciFetchNumSharedCounters, page 225
- GciDecSharedCounter, page 172
- GciIncSharedCounter, page 271
- GciReadSharedCounter, page 394
- GciReadSharedCounterNoLock, page 395
- GciFetchSharedCounterValuesNoLock, page 244
GciSetTraversalBufSwizzling

Control swizzling of the traversal buffers.

Syntax

```c
BoolType GciSetTraversalBufSwizzling(
    BoolType enabled);
```

Input Arguments

(enabled) If TRUE, enable normal byte-order swizzling of traversal buffers for the current RPC session. This is the default state for a session created by successful GciLogin(). If FALSE, the application program (for example, GemBuilder for Smalltalk) is responsible for subsequent swizzling of traversal buffers if needed.

Return Value

Returns the previous value of swizzling of traversal buffers. When called on a linkable session, returns FALSE and has no effect. If the current session is invalid, generates an error and returns FALSE.

Description

GciSetTraversalBufSwizzling controls swizzling of the traversal buffers used by these calls in an RPC session:

- GciStoreTrav, GciNbStoreTrav
- GciStoreTravDo, GciNbStoreTravDo
- GciStoreTravDoTrav, GciNbStoreTravDoTrav
- GciClampedTrav, GciNbClampedTrav
- GciMoreTraversal, GciNbMoreTraversal
- GciPerformTrav, GciNbPerformTrav
- GciExecuteStrTrav, GciNbExecuteStrTrav
GciSetVaryingSize

Set the size of a collection.

Syntax

void GciSetVaryingSize(
  OopType collection,
  int64 size);

Input Arguments

  collection    The OOP of the collection whose size you are specifying.
  size          The desired number of elements in the collection.

Description

  GciSetVaryingSize changes the size of a collection, adding nils to grow it, or truncating it, as necessary. It is equivalent to the Smalltalk method Object >> size:. It does not change the number of any named instance variables.

Example

void setVaryingSize_example(void)
{
  OopType oArr = GciNewOop(OOP_CLASS_ARRAY); // create new Array of size 0
  GciSetVaryingSize(oArr, 1000000); // logical size now 1 million
  GciStoreOop(oArr, 500000, GciI32ToOop(5678));
}

See Also

  GciFetchVaryingSize_, page 253
**GciShutdown**

Logout from all sessions and deactivate GemBuilder.

**Syntax**

```c
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 certain small, single-user systems.
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 a signal handler 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 another thread.

If GemStone is executing when it receives the break, it replies with the error message RT_ERR_SOFT_BREAK. Otherwise, it ignores the break.

Example

#include "signal.h"

extern "C" {
    static void doSoftBreak(int sigNum, siginfo_t* info, void* ucArg)
    {
        GciSoftBreak();
    }
}
void softBreakExample(void)
{
    // save previous SIGINT handler and install ours
    struct sigaction oldHandler;
    struct sigaction newHandler;
    newHandler.sa_handler = SIG_DFL;
    newHandler.sa_sigaction = doSoftBreak;
    newHandler.sa_flags = SA_SIGINFO | SA_RESTART;
    sigaction(SIGINT, &newHandler, &oldHandler);

    // execute a loop that will take 120 seconds to execute and
    // return the SmallInteger with value 11.
    OopType result = GciExecuteStr("| a | a := 1. 10 timesRepeat:[ System sleep:10. a := a + 1]. ^
    a", OOP_NIL/*use default symbolList for execution*/);

    BoolType done = FALSE;
    int breakCount = 0;
    do {
        // assume the user may type ctl-C or issue kill -INT from
        // another shell process during the 120 seconds.
        GciErrSType errInfo;
        if (GciErr(&errInfo)) {
            if (errInfo.category == OOP_GEMSTONE_ERROR_CAT &&
                errInfo.number == RT_ERR_SOFT_BREAK) {
                // GciExecuteStr was interrupted by a GciSoftBreak.
                breakCount++;
                // now continue the execution to finish the computation
                result = GciContinue(errInfo.context);
            } else {
                // FMT_OID format string is defined in gci.ht
                printf("unexpected error category "FMT_OID" number %d, %s\n",
                        errInfo.category, errInfo.number, errInfo.message);
                // terminate the execution
                GciClearStack(errInfo.context);
                done = TRUE;
            }
        } else {
            // GciExecuteStr or GciContinue completed without error
        }
    }
}
done = TRUE;
BoolType conversionErr = FALSE;
int val = GciOopToI32_(result, &conversionErr);
if (conversionErr) {
    printf("Error converting result to C int\n");
} else {
    printf("Got %d interrupts, result = %d\n", breakCount, val);
}

// restore previous SIGINT handler
sigaction(SIGINT, &oldHandler, NULL);

See Also

GciClearStack, page 145
GciContinue, page 154
GciExecute, page 191
GciHardBreak, page 267
GciPerform, page 371
GciStep

Continue code execution in GemStone with specified single-step semantics.

Syntax

OopType GciStep(
    OopType process,
    int level);

Input Arguments

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

level One of the following values:
    0 — step-into semantics starting from top of stack
    1 — step-over semantics starting from top of stack
    > 1 — step-over semantics from specified level on stack

Return Value

Returns the OOP of the result of the Smalltalk execution. Returns OOP_ILLEGAL in case of error.

Description

The GciStep function continues code execution in GemStone using the specified single-step semantics. This function is intended for use by debuggers.

If you specify a level that is either less than zero or greater than the value represented by GciPerform(process, "stackDepth", NULL, 0), GciStep generates an error.

See Also

GciPerform, page 371
GciStoreByte

Store one byte in a byte object.

Syntax

```c
void GciStoreByte(
    OopType                     theObject,
    int64                       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 Float or SmallFloat. You must store all the bytes of a Float or SmallFloat if you store any.
Example

void storeByte_example(void)
{
    OopType oString = GciNewOop(OOP_CLASS_STRING);

    for (int j = 0; j < 200; j++) {
        ByteType val = j;
        GciStoreByte(oString, j + 1, val);
    }
}

See Also

GciFetchByte, page 204
GciFetchBytes_, page 206
GciStoreBytes, page 447
GciStoreBytes

(MACRO) Store multiple bytes in a byte object.

Syntax

```c
void GciStoreBytes(
    OopType       theObject,
    int64         startIndex,
    const ByteType theBytes[],
    int64         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` macro 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 Float or SmallFloat. Use `GciStoreBytesInstanceOf` instead for Float or SmallFloat objects.
Example

```c
void storeBytes_example(void)
{
    OopType oString = GciNewOop(OOP_CLASS_STRING);

    enum { buf_size = 2000 };  
    ByteType buf[buf_size];
    for (int j = 0; j < buf_size; j++) {
        buf[j] = (ByteType)j;
    }
    GciStoreBytes(oString, 1, buf, buf_size);
}
```

See Also

GciFetchByte, page 204
GciFetchBytes, page 206
GciStoreByte, page 445
GciStoreBytesInstanceOf, page 449
GciStoreChars, page 451
GciStoreBytesInstanceOf

Store multiple bytes in a byte object.

Syntax

```c
void GciStoreBytesInstanceOf(
    OopType    theClass,
    OopType    theObject,
    int64      startIndex,
    const ByteType    theBytes[ ],
    int64      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 Float or SmallFloat object.

`GciStoreBytesInstanceOf` provides automatic byte swizzling for Float and SmallFloat objects. (For more about byte swizzling, see page 29.) The presence of the argument `theClass` enables the swizzling to be implemented more efficiently. If `theObject` is a Float or SmallFloat, 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 Float or SmallFloat, then theClass is ignored. Hence, you must supply the correct class for theClass if theObject is a Float or SmallFloat, but you can use OOP_NIL otherwise.

Example

```c
void storeBytesInstanceOf_example(void)
{
  double pi = 3.1415926;
  OopType oFloat = GciNewOop(OOP_CLASS_FLOAT);
  GciStoreBytesInstanceOf(OOP_CLASS_FLOAT, oFloat, 1,
      (ByteType *)&pi, sizeof(pi));
}
```

See Also

GciFetchByte, page 204
GciFetchBytes, page 206
GciStoreByte, page 445
GciStoreBytes, page 447
GciStoreChars, page 451
**GciStoreChars**

Store multiple ASCII characters in a byte object.

**Syntax**

```c
void GciStoreChars(
    OopType       theObject,  
    int64         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 Float or SmallFloat. ASCII characters have no meaning as bytes in a Float or SmallFloat object.

**Example**

```c
void storeChars_example(void)
{
    OopType oString = GciNewOop(OOP_CLASS_STRING);

    GciStoreChars(oString, 1, "some string data");
}
```
See Also

GciFetchByte, page 204  
GciFetchBytes, page 206  
GciStoreByte, page 445  
GciStoreBytes, page 447
GciStoreDynamicIv

Create or change the value of an object’s dynamic instance variable.

Syntax

```c
void GciStoreDynamicIv(
    OopType theObject,
    OopType aSymbol,
    OopType value);
```

Input Arguments

- `theObject`: The OOP of the GemStone object.
- `aSymbol`: Specifies the dynamic instance variable of the object.
- `value`: The value to store in the dynamic instance variable.

Return Value

Creates or changes the value of the dynamic instance variable specified by `aSymbol` within `theObject`.

Description

This function stores a value into the dynamic instance variable specified by `aSymbol`.

Dynamic instance variables are not allowed in instances of ExecBlock, Behavior, GsNMethod, or special objects.

To delete a dynamic instance variable, pass OOP_REMOTE_NIL as the value.

See Also

- GciFetchDynamicIv, page 214
- GciFetchDynamicIvs, page 215
GciStoreIdxOop

Store one OOP in an indexable pointer object’s unnamed instance variable.

Syntax

```c
void GciStoreIdxOop(
    OopType theObject,
    int64 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 115.)
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 248.

```c
void storeIdxOop_example(void)
{
    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);

    OopType otherComp = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1333]", OOP_NIL);

    // store new value into 3rd element of aComponent’s parts list
    GciStoreIdxOop(aComponent, 3, otherComp);

    GciReleaseOops(&aComponent, 1);    // release results of execution
    GciReleaseOops(&otherComp, 1);
}
```

See Also

GciAddOopToNsc, page 115
GciFetchVaryingOop, page 248
GciFetchVaryingOops, page 251
GciStoreIdxOops, page 456
GciStoreIdxOops

Store one or more OOPs in an indexable pointer object’s unnamed instance variables.

Syntax

void GciStoreIdxOops(
  OopType theObject,
  int64 startIndex,
  const OopType theOops[],
  int 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.
umOops 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 117.)
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 248.

```c
void storeIdxOops_example(void)
{
    // retrieve a random instance of class Component
    OopType firstC = GciExecuteStr(
        "AllComponents detect:([i|i partNumber = 1234])", OOP_NIL);

    OopType secondC = GciExecuteStr(
        "AllComponents detect:([i|i partNumber = 1333])", OOP_NIL);

    // make first component’s parts list be identical to second
    component’s list
    enum { buf_size = 100 ];
    OopType buf[buf_size];
    int64 firstSize = GciFetchVaryingSize_(firstC);
    int64 idx = 1;
    while (idx <= firstSize) {
        int numRet = GciFetchVaryingOops(firstC, idx, buf, buf_size);
        GciStoreIdxOops(secondC, idx, buf, numRet);
        idx += numRet;
    }
    // truncate second component’s parts list if it was larger than first’s
    GciSetVaryingSize(secondC, firstSize);

    GciReleaseOops(&firstC, 1);  // release results of executions
    GciReleaseOops(&secondC, 1);
}
```

See Also

GciAddOopsToNsc, page 117
GciFetchVaryingOop, page 248
GciFetchVaryingOops, page 251
GciReplaceOops, page 410
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GciReplaceVaryingOops</td>
<td>412</td>
</tr>
<tr>
<td>GciStoreIdxOop</td>
<td>454</td>
</tr>
<tr>
<td>GciStoreIdxOops</td>
<td>456</td>
</tr>
<tr>
<td>GciStoreNamedOops</td>
<td>462</td>
</tr>
<tr>
<td>GciStoreOops</td>
<td>468</td>
</tr>
</tbody>
</table>
GciStoreNamedOop

Store one OOP into an object’s named instance variable.

Syntax

```c
void GciStoreNamedOop(
    OopType theObject,
    int64 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 248.

```c
void storeNamedOop_example(void)
{
  // C constants to match Smalltalk class definition
  enum { COMPONENT_OFF_PARTNUMBER = 1,
         COMPONENT_OFF_NAME       = 2,
         COMPONENT_OFF_COST       = 3 };

  // retrieve a random instance of class Component
  OopType aComponent = GciExecuteStr(
    "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
  if (aComponent == OOP_NIL) {
    // error during execution or detect found nothing
    return;
  }

  // assign a new value to the name instance variable of aComponent
  OopType newName = GciNewString("compressor blade");
  GciStoreNamedOop(aComponent, COMPONENT_OFF_NAME, newName);

  // alternate approach: assign a new value to a named instance
  //variable without knowing its offset at compile time
  GciStoreNamedOop(aComponent, GciIvNameToIdx(GciFetchClass(aComponent), "name"), newName);
  GciReleaseOops(&newName, 1);
  GciReleaseOops(&aComponent, 1);
}
```

See Also

GciFetchNamedOop, page 216
GciFetchNamedOops, page 219
GciStoreIdxOop, page 454
GciStoreNamedOops, page 462
**GciStoreNamedOops**

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

**Syntax**

```c
void GciStoreNamedOops(
    OopType theObject,
    int64 startIndex,
    const OopType theOops[],
    int 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 248.

```c
void storeNamedOops_example(void)
{
    // retrieve a random instance of class Component
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        // execution error, or detect: found nothing
        return;
    }

    // fetch name instance variables without knowing offset at C
    compile time
    int namedSize = GciFetchNamedSize(aComponent);
    if (namedSize == 0) {
        // error during fetch
        return;
    }
    OopType *oBuffer = (OopType*) malloc( sizeof(OopType) *
        namedSize );
    if (oBuffer == NULL) {
        printf("malloc failure\n");
        return;
    }
    int numRet = GciFetchNamedOops(aComponent, 1, oBuffer,
        namedSize);
    if (numRet != namedSize) {
        printf("error during fetch\n");
        return;
    }

    // alter one of the instVars and then store them all
    OopType newName = GciNewString("compressor blade");
    int ivOffset = GciIvNameToIdx(GciFetchClass(aComponent),
        "name");
    if (ivOffset <= 0) {
        printf("error during GciIvNameToIdx\n");
    }
}
```
return;
}
oBuffer[ivOffset - 1] = newName;
GciStoreNamedOops(aComponent, 1, oBuffer, namedSize);

GciReleaseOops(&newName, 1);
GciReleaseOops(&aComponent, 1);
}

See Also

GciFetchNamedOop, page 216
GciFetchNamedOops, page 219
GciReplaceOops, page 410
GciReplaceVaryingOops, page 412
GciStoreIdxOop, page 454
GciStoreIdxOops, page 456
GciStoreNamedOop, page 459
GciStoreNamedOops, page 462
GciStoreOops, page 468
GciStoreOop

Store one OOP into an object’s instance variable.

Syntax

```c
void GciStoreOop(
    OopType theObject,
    int64 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 115.
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 248.

```c
void storeOop_example(void) {
    /* C constants to match Smalltalk class definition */
    enum { COMPONENT_OFF_NAME = 2 };

    /* retrieve a random instance of class Component */
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);

    OopType newName = GciNewString("vane");

    /* Two ways to assign new value to name instance variable of aComponent */
    GciStoreOop(aComponent, COMPONENT_OFF_NAME, newName);
    GciStoreNamedOop(aComponent, COMPONENT_OFF_NAME, newName);

    OopType subPart = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1333]", OOP_NIL);

    /* 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) + 3, subPart);
    GciStoreIdxOop(aComponent, 3, subPart);
}
```

See Also

- `GciAddOopToNsc`, page 115
- `GciFetchVaryingOop`, page 248
- `GciFetchVaryingOops`, page 251
GcIFetchOops, page 234
GcIStoreOops, page 468
GciStoreOops

Store one or more OOPs into an object’s instance variables.

Syntax

```c
void GciStoreOops(
    OopType theObject,
    int64 startIndex,
    const OopType theOops[],
    int 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 117.
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 \texttt{GciFetchVaryingOop} function on page 248.

```c
void storeOops_example(void)
{
    /* retrieve a random instance of class Component */
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);

    int namedSize = GciFetchNamedSize(aComponent);
    int64 instSize = GciFetchSize_(aComponent);
    // allow space in buffer for storing into first varying instVar
    // appending a new varying instVar
    int64 bufVaryingSize = instSize - namedSize + 1;
    if (bufVaryingSize < 2)
        bufVaryingSize = 2;

    int64 bufSize = namedSize + bufVaryingSize;
    OopType *buf = (OopType*) malloc(sizeof(OopType) * bufSize);
    if (buf == NULL) {
        printf("malloc failure");
        return;
    }
    GciFetchOops(aComponent, 1, buf, instSize);

    OopType newName = GciNewString("strut");
    int nameOfs = GciIvNameToIdx(GciFetchClass(aComponent), "name");
    buf[nameOfs - 1] = newName;

    OopType firstSubPart = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1333]", OOP_NIL);

    OopType lastSubPart = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1555]", OOP_NIL);

    // assign first element of parts list
    buf[namedSize] = firstSubPart;
}
```
// append lastSubPart to aComponent’s parts list
int64 newSize = instSize + 1;
buf[newSize - 1] = lastSubPart;

// now store all the instVars back to the repository
GciStoreOops(aComponent, 1, buf, newSize);
}

See Also

GciAddOopsToNsc, page 117
GciFetchNamedOops, page 219
GciFetchOop, page 231
GciFetchOops, page 234
GciFetchVaryingOop, page 248
GciReplaceOops, page 410
GciReplaceVaryingOops, page 412
GciStoreIdxOops, page 456
GciStoreNamedOops, page 462
GciStoreOop, page 465
GciStoreOops, page 468
GciStorePaths

Store selected multiple OOPs into an object tree.

Syntax

```c
BoolType GciStorePaths(
    const OopType theOops[],
    int numOops,
    const int paths[],
    const int pathSizes[],
    int numPaths,
    const OopType newValues[],
    int * 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 an integer that indicates which element of the `newValues` array could not be successfully stored. If all values were successfully stored, `failCount` is 0. If the ith store failed, `failCount` is i. If any of the objects in `newValues` does not exist, or is not an OOP allocated to GemBuilder, `failCount` is 1.
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 53.

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 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:

\[\text{newValues[ ((i-1) * numPaths) + (j-1) ]}\]

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

```c
void storePath1(void)
{
    enum { path_size = 5 };  
    int   aPath[path_size]; /* the path itself */
    int   aSize = path_size; /* the size of the path */

    OopType anOop; // the OOP to use as the root of the path
    anOop = GciExecuteStr("AllComponents detect:[i\i partNumber = 1234]", OOP_NIL);
    if (anOop == OOP_NIL) {
        return; // error during resolve
    }

    OopType newValue = GciNewString("a new value");
    int    failCount;

    GciStorePaths(&anOop, 1, aPath, &aSize, 1, &newValue,
                  &failCount);
}
```
Example 2: Calling sequence for multiple objects with a single path

```c
void storePath2(void)
{
    OopType coll = GciResolveSymbol("AllComponents", OOP_NIL);
    if (coll == OOP_NIL) {
        return; // error during resolve
    }
    enum { num_roots = 3,
           path_size = 5};
    OopType oops[num_roots];
    int numRet = GciFetchVaryingOops(coll, 1, oops, num_roots);
    if (numRet != num_roots) {
        return; // error during fetch or collection too small
    }

    int aPath[path_size];
    int aSize = path_size;
    for (int j = 0; j < path_size; j++) {
        aPath[j] = 1; // arbitrary offsets
    }

    OopType newValues[num_roots];
    for (int j = 0; j < num_roots; j++) {
        newValues[j] = GciI32ToOop(1345600 + j);
    }
    int failCount;
    GciStorePaths(oops, num_roots, aPath, &aSize, 1, newValues, &failCount);
}
```
Example 3: Calling sequence for a single object with multiple paths

```c
void storePath3(void)
{
    OopType anOop; // the OOP to use as the root of the path
    anOop = GciExecuteStr("AllComponents detect:\[:i|i partNumber = 1234]", OOP_NIL);
    if (anOop == OOP_NIL) {
        return; // error during execution
    }

    enum { num_paths = 10,
        path_size = 5 }

    int pathSizes[num_paths];
    int paths[path_size * num_paths ];
    int idx = 0;
    for (int j = 0; j < num_paths; j++) {
        for (int k = 0; k < path_size; k++) {
            paths[idx++] = k + 1; // arbitrary offset
        }
    }
    OopType newValues[num_paths];
    for (int j = 0; j < num_paths; j++) {
        newValues[j] = GciI32ToOop(1345600 + j);
    }
    int     failCount;

    GciStorePaths(&anOop, 1, paths, pathSizes, num_paths, newValues, &failCount);
}
```
Example 4: Calling sequence for multiple objects with multiple paths

```c
void storePaths4(void)
{
    OopType coll = GciResolveSymbol("AllComponents", OOP_NIL);
    if (coll == OOP_NIL) {
        return ; // error during resolve
    }

    enum { num_roots = 10,
           num_paths = 3,
           path_size = 5,
           num_new_values = num_roots * num_paths
        };
    OopType oops[num_roots];
    int numRet = GciFetchVaryingOops(coll, 1, oops, num_roots);
    if (numRet != num_roots) {
        return; // error during fetch or collection too small
    }

    int pathSizes[num_paths];
    int paths[path_size * num_paths ];
    int idx = 0;
    for (int j = 0; j < num_paths; j++) {
        for (int k = 0; k < path_size; k++) {
            paths[idx++] = k + 1; // arbitrary offset
        }
    }

    OopType newValues[num_new_values];
    for (int j = 0; j < num_new_values; j++) {
        newValues[j] = GciI32ToOop(1345600 + j);
    }
    int failCount;
    GciStorePaths(oops, num_roots, paths, pathSizes, num_paths, 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 248.

```c
void storePaths5(void)
{
    // retrieve a random instance of class Component */
    OopType aComponent = GciExecuteStr(
        "AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);
    if (aComponent == OOP_NIL) {
        return; // error in execute, or detect: found nothing
    }

    // assign a new value to the name instVar of 5th element of
    // aComponent’s parts list
    enum { path_size = 2 };  
    int path[path_size];
    path[0] = -5;  // 5th varying instVar
    path[1] = GciIvNameToIdx(GciFetchClass(aComponent), "name");
    int pathSizes = path_size;

    OopType newValue = GciNewString("pump");
    int failCount;
    GciStorePaths(&aComponent, 1, path, &pathSizes, 1, &newValue, &failCount);
}
```

See Also

GciFetchPaths, page 237
GciStoreTrav

Store multiple traversal buffer values in objects.

Syntax

```c
void GciStoreTrav(
    GciTravBufType * travBuff,
    int 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 an 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 53.

The value of `behaviorFlag` should be given by using one or more of the following GemBuilder mnemonics: GCI_STORE_TRAV_DEFAULT, GCI_STORE_TRAV_NSREP, GCI_STORE_TRAV_CREATE, 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. 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.

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 Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

### Use of Object Reports

The **GciStoreTrav** function stores values in GemStone objects according to the object reports contained in `travBuff`. Each object report is an instance of the C++ class **GciObjRepSType** (described in “The Object Report Structure” on page 94). **GciStoreTrav** uses the fields in each object report as follows:

- `rpt->hdr.valueBuffSize`
  
  The size (in bytes) of the value buffer, where object data is stored. If `objId` is a Float or SmallFloat and `valueBuffSize` differs from the actual size for objects of `objId`’s class, then **GciStoreTrav** raises an error.

- `rpt->hdr.namedSize`
  
  Ignored by this function.
rpt->hdr.setIdxSize()

Only needs to be called 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.

rpt->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 nonindexable objects, GciStoreTrav raises an error if valueBuffSize and firstOffset imply a size that exceeds the actual size of the object. If objId is a Float or SmallFloat and firstOffset is not one, then GciStoreTrav raises an error.

rpt->hdr.objId

The OOP of the object to be stored.

rpt->hdr.oclass

Used only when creating a new object, to identify its intended class.

rpt->hdr.objectSecurityPolicyId

The ID of the object’s security policy.

rpt->hdr.clearBits()

Must be called before any of the following:

rpt->hdr.setObjImpl()

You must call rpt->hdr.setObjImpl to set this field to be consistent with the object’s implementation.

rpt->hdr.setInvariant()

Boolean value. Call rpt->hdr.setInvariant(TRUE) if you want this object to be made invariant after the store specified by rpt* is completed.

rpt->hdr.setIndexable()

Ignored by this function.

rpt->valueBufferBytes()

The value buffer of an object of byte format.
rpt->valueBufferOops()

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 293
GciNbMoreTraversal, page 314
GciNbStoreTrav, page 321
GciNbTraverseObjs, page 328
GciNewOopUsingObjRep, page 338
GciProcessDeferredUpdates_, page 388
GciStoreTravDo_, page 482
GciTraverseObjs, page 510
GciStoreTravDo_*

Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object.

NOTE
In previous GemStone/S 64 Bit releases, this function was named GciStoreTravDo (without the underscore).

Syntax

OopType GciStoreTravDo_((
    GciStoreTravDoArgsSType *args);

Input Arguments

args
An instance of GciStoreTravDoArgsSType (as described in $GEMSTONE/include/gcicmn.ht) containing the following fields:

GciTravBufType* storeTravBuff
The traversal buffer. For details, see “GciStoreTrav” on page 478.

int storeTravFlags
A flag that determines how the objects should be handled. For details, see “GciStoreTrav” on page 478.

int doPerform
If this field is 0, this function executes a string using args->u.executesr, with the semantics of “GciExecuteStrFromContext” on page 198. If this field is 1, then executes a perform using args->u.perform, with the semantics of “GciPerformNoDebug” on page 373. Other values of this field are only for use with GciStoreTravDoTravRefs_ or GciNbStoreTravDoTravRefs_

int doFlags
Flags to disable or permit asynchronous events
and debugging in Smalltalk, as described in “GciPerformNoDebug” on page 373. These flags apply whatever the value of \texttt{doPerform}.

\begin{verbatim}
union 
  u
  One of two structures containing appropriate
  input fields for the specified operation. The
  structure \texttt{u.perform} should be used when
  \texttt{doPerform} is set to 1, and \texttt{u.executestr} should be
  used when \texttt{doPerform} is set to 0. For more
  information on these structs and how to use
  them, see \texttt{gcicmn.h}.

OopType* alteredTheOops
  An array allocating memory for OOPs of objects
  that will be modified as a consequence of
  executing the specified code. For more
  information, see “GciAlteredObjs” on page 121.

int alteredNumOops
  The number of OOPs in the previous array. On
  input, the caller must set this to the maximum
  number of OOPs that will fit in \texttt{alteredTheOops}.
  Upon completion, this field indicates the number
  of OOPs actually written to \texttt{alteredTheOops}.

BoolType alteredCompleted
  Upon output, TRUE if the \texttt{alteredTheOops} contains
  the complete set of objects modified as a result of
  executing the specified code; false otherwise. If
  FALSE, call GciAlteredObjs for the rest of the
  modified objects.

const OopType* execBlock_args
  This field is ignored.

int execBlock_numArgs
  This field is ignored.
\end{verbatim}

Return Value

Returns the OOP of the result of executing the specified code. In case of error, this function returns OOP\_NIL.
Description

The GciStoreTravDo function works exactly as “GciStoreTrav” on page 478, and also executes the supplied code in the same network round-trip.

The description of “GciStoreTrav” on page 478 explains the first two arguments. If the value of the third argument is 1, see “GciPerformNoDebug” on page 373 for details of the next five arguments—flags to enable or disable asynchronous events, and the first nested structure.

If the value of the third argument is 2, see “GciExecuteStrFromContext” on page 198 for details on next set of arguments—flags to enable or disable asynchronous events, and the second nested structure of five arguments.

If the value of the third argument is 3, the arguments are similar to those for GciExecuteStrFromContext, but source must be a String that when compiled will return a Block. In this case, the last two arguments also are used, which provide the arguments, and the count of arguments, to be used to execute the compiled block.

The next five input arguments supply needed output after the function has completed. Read alteredTheOops to get the OOPs of the objects that were modified; read alteredSymbolBuf to get the pairs of symbols and symbol dictionaries for symbol canonicalization; finally, read alteredCompleted to determine if the array as originally allocated was large enough to hold all the modified objects. If the value is false, the array was too small and holds only some of the modified objects; in this case, call GciAlteredObjs for the rest.

Handling Error Conditions

If you get a run time error while executing GciStoreTravDo, we recommend that you abort the current transaction.

See Also

GciAlteredObjs, page 121
GciExecuteStrFromContext, page 198
GciMoreTraversal, page 293
GciNbMoreTraversal, page 314
GciNbStoreTrav, page 321
GciNbTraverseObjs, page 328
GciNewOopUsingObjRep, page 338
GciPerformNoDebug, page 373
GciProcessDeferredUpdates_, page 388
GciStoreTrav, page 478
GciTraverseObjs, page 510
**GciStoreTravDoTrav_**

Combine in a single function the calls to GciStoreTravDo_ and GciClampedTrav, to store multiple traversal buffer values in objects, execute the specified code, and traverse the result object.

NOTE

In previous GemStone/S 64 Bit releases, this function was named GciStoreTravDoTrav (without the underscore).

**Syntax**

```c
BoolType GciStoreTravDoTrav_
    (GciStoreTravDoArgsSType *stdArgs,
     GciClampedTravArgsSType *ctArgs);
```

**Input Arguments**

- `stdArgs`: An instance of GciStoreTravDoArgsSType. For details, refer to the discussion of GciStoreTravDo_ on page 482.
- `ctArgs`: An instance of GciClampedTravArgsSType. For details, see the discussion of GciClampedTrav on page 135.

**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**

This function allows the client to execute behavior on the Gem and return the traversal of the result object in a single network round-trip.
See Also

GciClampedTrav, page 135
GciStoreTrav, page 478
GciStoreTravDo_, page 482
**GciStoreTravDoTravRefs**

Combine in a single function modifications to session sets, traversal of objects to the server, optional Smalltalk execution, and traversal to the client of changed objects and (optionally) the result object.

**NOTE**

In previous GemStone/S 64 Bit releases, this function was named

**GciStoreTravDoTravRefs** (without the underscore).

**Syntax**

```c
int GciStoreTravDoTravRefs_(
    const OopType * oopsNoLongerReplicated,
    int numNotReplicated,
    const OopType * oopsGcedOnClient,
    int numGced,
    GciStoreTravDoArgsSType *stdArgs,
    GciClampedTravArgsSType *ctArgs);
```

**Input Arguments**

- `oopsNoLongerReplicated`: An Array of objects to be removed from the PureExportSet and added to the ReferencedSet.
- `numNotReplicated`: The number of elements in `oopsNoLongerReplicated`.
- `oopsGcedOnClient`: An Array of objects to be removed from both the PureExportSet and ReferencedSet.
- `numGced`: The number of elements in `oopsGcedOnClient`.
- `stdArgs`: An instance of `GciStoreTravDoArgsSType` (as described in `$GEMSTONE/include/gcicmn.ht`) containing the following fields:
  - `GciTravBufType* storeTravBuff`: The traversal buffer. For details, see “GciStoreTrav” on page 478.
  - `int storeTravFlags`: A flag that determines how the objects should be handled. For details, see “GciStoreTrav” on page 478.
int doPerform
If this field is 0, this function executes a string using args->u.executestr, with the semantics of “GciExecuteStrFromContext” on page 198. If this field is 1, then executes a perform using args->u.perform, with the semantics of “GciPerformNoDebug” on page 373. If this field is 2, execute a string that is the source code for a Smalltalk block using stdArgs->u.executestr, passing the block arguments in execBlock_args. If this field is 3, perform no server Smalltalk execution, but traverse the object specified in stdArgs->u.perform.receiver as if it was the results of execution. If this field is 4, resume execution of a suspended Smalltalk Process using stdArgs->u.continueArgs, with the semantics of “GciContinueWith” on page 156.

int doFlags
Flags to disable or permit asynchronous events and debugging in Smalltalk, as described in “GciPerformNoDebug” on page 373. These flags apply whatever the value of doPerform.

union u
One of three structures containing appropriate input fields for the specified operation. The structure u.perform should be used when doPerform is set to 1 or 3, u.executestr should be used when doPerform is set to 0 or 2, and u.continueArgs should be used when doPerform is set to 4. For more information on these structs and how to use them, see gcicmn.ht.

OopType* alteredTheOops
This field is ignored.

int alteredNumOops
This field is ignored.

BoolType alteredCompleted
This field is not used.

const OopType* execBlock_args
An array of the arguments to the block to be
executed. Only applies if `doPerform` is 2, ignored otherwise.

```c
int execBlock_numArgs
```

The number of the arguments provided in `execBlock_args`. This must match the declared number of arguments in the block source string. Only applies if `doPerform` is 2, ignored otherwise.

c`tArgs`

An instance of `GciClampedTravArgsSType`. For details, see the discussion of `GciClampedTrav` on page 135, with one exception. The valid `retrievalFlags` are limited to:

- `GCI_RETRIEVE_DEFAULT`
- `GCI_TRAV_REFS_EXCLUDE_RESULT_OBJ` will suppress traversal of the result object. The altered objects will still be traversed to the specified level.

No other `retrievalFlags` values should be used with this function.

**Return Value**

Returns an int with the following meaning:

- 0 — traversal of both altered objects and execution result completed.
- 1 — traversal buffer became full. You must call `GciMoreTraversal` to finish traversal of the altered and result objects.

**Description**

This function allows the client to modify the PureExportSet and ReferencedSet, modify or create any number of objects on the server, execute behavior on the Gem, and return the traversal of the changed objects and the result object, all in a single network round-trip.

The elements in `oopsGcedOnClient` are removed from both PureExportSet and ReferencedSet, and the elements in `oopsNoLongerReplicated` are removed from the PureExportSet and added to the ReferencedSet.

Objects in the ReferencedSet are protected from garbage collection, but may be faulted out of memory. Dirty tracking is not done on objects in the ReferencedSet.

Then per the `stdArgs`, a `GciStoreTrav` is done, which may modify or create any number of objects on the server. Newly created objects are added to the PureExportSet.
Then, if specified, Smalltalk execution is performed as in `GciPerformNoDebug`, `GciExecuteStrFromContext`, or executing the block code with the given arguments.

Finally, this function does a special `GciClampedTrav` starting with altered objects, followed by the execution result from the previous step. If no execution was specified, the specified object is traversed as if it was an execution result. Altered objects are those that would be returned from a `GciAlteredObjs` after the code execution step. This traversal both relies on the contents of the PureExportSet and ReferencedSet does not, and also modifies those sets in ways that `GciClampedTrav` does not. For details, see the comments in `gci.hf`.

`GciStoreTravDoTravRefs_` is not intended for use within a user action.

**See Also**

- `GciClampedTrav`, page 135
- `GciStoreTrav`, page 478
- `GciStoreTravDo_`, page 482
- `GciStoreTravDoTrav_`, page 486
GciStringToInteger

Convert a C string to a GemStone SmallInteger, LargePositiveInteger or LargeNegativeInteger object.

Syntax

OopType GciStringToInteger(
    const char* string,
    int64 stringSize );

Input Arguments

string The C string to be translated into a GemStone SmallInteger, LargePositiveInteger or LargeNegativeInteger object.

stringSize The length of string.

Return Value

Returns the OOP of the GemStone SmallInteger, LargePositiveInteger or LargeNegativeInteger object. If string has an invalid format, this function returns OOP_NIL without an error.

Description

The GciStringToInteger function translates a C string to a GemStone SmallInteger, LargePositiveInteger or LargeNegativeInteger object that has the same value.

Leading blanks are ignored. Trailing non-digits are ignored.

See Also
**GciStrKeyValueDictAt**

Find the value in a symbol KeyValuex dictionary at the corresponding string key.

**Syntax**

```c
void GciStrKeyValueDictAt(
    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 KeyValuex 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.

**See Also**

- `GciStrKeyValueDictAtObj`, page 494
- `GciStrKeyValueDictAtObjPut`, page 495
- `GciStrKeyValueDictAtPut`, page 496
GciStrKeyValDictAtObj

Find the value in a symbol KeyValue dictionary at the corresponding object key.

Syntax

```c
void GciStrKeyValDictAtObj(
    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 GemStone Smalltalk expression:

```
^ { theDict at: keyObj }
```

See Also

- GciStrKeyValDictAt, page 493
- GciStrKeyValDictAtObjPut, page 495
- GciStrKeyValDictAtPut, page 496
GciStrKeyValueDictAtObjPut

Store a value into a symbol KeyValue dictionary at the corresponding object key.

Syntax

```c
void GciStrKeyValueDictAtObjPut(
    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:

```smalltalk
theDict at: keyObj put: theValue
```

See Also

- GciStrKeyValueDictAt, page 493
- GciStrKeyValueDictAtObj, page 494
- GciStrKeyValueDictAtPut, page 496
GciStrKeyValueDictAtPut

Store a value into a symbol KeyValue dictionary at the corresponding string key.

Syntax

```c
void GciStrKeyValueDictAtPut(
    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.

See Also

- GciStrKeyValueDictAt, page 493
- GciStrKeyValueDictAtObj, page 494
- GciStrKeyValueDictAtObjPut, page 495
GciStrToPath

Convert a path representation from string to numeric.

This function is deprecated and may be removed from future releases.

Syntax

```c
BoolType GciStrToPath(
    OopType aClass,
    const char pathString[],
    int64 maxPathSize,
    int * resultPathSize,
    int 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 53.

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 237.

Restrictions

Note that **GciStrToPath** can convert a numeric path only if the instance variables of the specified Smalltalk class (**aClass**) are guaranteed to have the same valid path for all instances.

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 248.

```c
void strToPath_example(void)
{
  // retrieve a random instance of class Component */
  OopType aComponent = GciExecuteStr("AllComponents detect:[i|i partNumber = 1234]", OOP_NIL);

  // fetch name instVar of the first 10 elements of aComponent’s part list */
  OopType oClass = GciFetchClass(aComponent);
  enum { buf_size = 10 }; 
  OopType subParts[buf_size];
  int64 numSubParts = GciFetchVaryingOops(aComponent, 1, subParts, buf_size);

  enum { path_array_size = 3 }; 
  int path[path_array_size];
  int pathSize; // actual number of terms in path , expect 1
  GciStrToPath(oClass, "name", path_array_size, &pathSize, path);

  int numPaths = 1;
  OopType nameIvs[buf_size];
  GciFetchPaths(subParts, numSubParts, path, &pathSize, numPaths, nameIvs);
}
```

GciFetchPaths, page 237
GciPathToStr, page 368
GciStorePaths, page 471
GciSwapBytesUint

Swap the byte order of an array of uint.

Syntax

```c
void GciSwapBytesUint(
    uint * buf,
    intptr_t numChars);
```

Input Arguments

- `buf` An array of uint.
- `numChars` The size of the array.

Description

Swaps the byte order of the specified array of uint.

See Also

GciSwapBytesUshort, page 501
GciSwapBytesUshort

Swap the byte order of an array of ushort.

Syntax

```c
void GciSwapBytesUshort( 
    ushort *  buf, 
    intptr_t   numChars );
```

Input Arguments

- `buf` An array of ushort.
- `numChars` The size of the array.

Description

Swaps the byte order of the specified array of ushort.

See Also

GciSwapBytesUint, page 500
GciSymDictAt

Find the value in a symbol dictionary at the corresponding string key.

Syntax

```c
void GciSymDictAt(
    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`, 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.
See Also

GciSymDictAtObj, page 504
GciSymDictAtObjPut, page 505
GciSymDictAtPut, page 506
GciSymDictAtObj

Find the value in a symbol dictionary corresponding to the key object.

Syntax

```c
void GciSymDictAtObj(
    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

Fetches 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 GemStone Smalltalk expression:

```smalltalk
^ { theDict at: keyObj . theDict associationAt: keyObj }
```

See Also

- GciSymDictAt, page 502
- GciSymDictAtObjPut, page 505
- GciSymDictAtPut, page 506
GciSymDictAtObjPut

Store a value into a symbol dictionary at the corresponding object key.

**Syntax**

```c
void GciSymDictAtObjPut(
    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
```

**See Also**

- GciSymDictAt, page 502
- GciSymDictAtObj, page 504
- GciSymDictAtPut, page 506
GciSymDictAtPut

Store a value into a symbol dictionary at the corresponding string key.

Syntax

void GciSymDictAtPut(
    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.

See Also

GciSymDictAt, page 502
GciSymDictAtObj, page 504
GciSymDictAtObjPut, page 505
GciTrackedObjsFetchAllDirty

Find all exported or tracked objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.

Syntax

```c
void GciTrackedObjsFetchAllDirty(
    OopType                  exportedDirty,
    int64 *                  numExportedDirty,
    OopType                  trackedDirty,
    int64 *                  numTrackedDirty);
```

Input Arguments

- `exportedDirty` OOP of the collection (an instance of either IdentitySet or IdentityBag) that will contain the objects in the ExportedDirtyObjs set.
- `trackedDirty` OOP of the collection (an instance of either IdentitySet or IdentityBag) that will contain the objects in the TrackedDirtyObjs set.

Result Arguments

- `numExportedDirty` Pointer to an integer that returns the number of objects in the `exportedDirty` collection.
- `numTrackedDirty` Pointer to an integer that returns the number of objects in the `trackedDirty` collection.

Description

GciTrackedObjsFetchAllDirty fetches all dirty objects and sorts them into two categories:

- Objects in the ExportedDirtyObjs set - objects in the PureExportSet that have been changed since the ExportedDirtyObjs set was initialized or cleared.
- Objects in the TrackedDirtyObjs set - objects in the GciTrackedObjs set that have been changed since the TrackedDirtyObjs set was initialized or cleared.
The ExportedDirtyObjs set is initialized by `GciDirtyObjsInit`; it is cleared by calls to `GciDirtyAlteredObjs`, `GciDirtyExportedObjs`, `GciDirtySaveObjs`, or `GciTrackedObjsFetchAllDirty` (this function). The TrackedDirtyObjs set is initialized by `GciTrackedObjsInit` and cleared by calls to `GciDirtyAlteredObjs`, `GciDirtySaveObjs`, `GciDirtyTrackedObjs`, or `GciTrackedObjsFetchAllDirty` (this function).

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

- The object was changed by Smalltalk execution.
- 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).

You must call both `GciDirtyObjsInit` and `GciTrackedObjsInit` once after `GciLogin` before calling `GciTrackedObjsFetchAllDirty`.

Note that the ExportedDirtyObjs and TrackedDirtyObjs sets are cleared when this function is executed.

See Also

- “Garbage Collection” on page 49
- “GciDirtyExportedObjs” on page 174
- “GciDirtyObjsInit” on page 176
- “GciDirtySaveObjs” on page 178
- “GciDirtyTrackedObjs” on page 180
- “GciTrackedObjsInit” on page 509
**GciTrackedObjsInit**

Reinitialize the set of tracked objects maintained by GemStone.

**Syntax**

```c
void GciTrackedObjsInit();
```

**Description**

The `GciTrackedObjsInit` function permits an application to request GemStone to maintain a set of tracked objects. `GciTrackedObjsInit` must be called once after `GciLogin` before other tracked objects functions in order for those functions to operate properly, because they depend upon GemStone’s set of tracked objects.

**See Also**

- `GciDirtySaveObjs`, page 178
- `GciDirtyTrackedObj`, page 180
- `GciHiddenSetIncludesOop`, page 268
- `GciTrackedObjFetchAllDirty`, page 507
**GciTraverseObjs**

Traverse an array of GemStone objects.

**Syntax**

```c
BoolType GciTraverseObjs(
    const OopType theOops[ ],
    int numOops,
    GciTravBufType * travBuff,
    int level);
```

**Input Arguments**

- `theOops` An array of OOPs representing the objects to traverse.
- `numOops` The number of elements in `theOops`.
- `travBuffSize` 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.
- `level` The number of bytes in `travBuff`.

**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 53.

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 Float and SmallFloat objects. (For more about byte swizzling, see page 29.)

Organization of the Traversal Buffer

The first element placed in a traversal buffer is an 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 described on page 479.

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:

\[
\text{GciTravBufType* travBufAllocation_example(void)}
\]

\[
\begin{align*}
\text{int numObjs} & = 100; \\
\text{int bodyBytesPerObj} & = 1000; \\
\text{size_t allocationSize} & = \\
\text{numObjs} \times \text{GCI_ALIGN}(&\text{sizeof(GciObjRepHdrSType)} + \text{bodyBytesPerObj}); \\
\text{GciTravBufType* buf = GciTravBufType::malloc(allocationSize);} \\
\text{return buf;} \\
\end{align*}
\]

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 7 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 `rpt->valueBufferBytes()` is an array of type `ByteType`; for pointer objects and NSCs, the buffer `rpt->valueBufferOops()` is an array of type `OopType`. The size of the report’s value buffer (`rpt->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.

**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.

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.objSecurityPolicyId` = 0
- `hdr.objImpl` = `GC_FORMAT_SPECIAL`
- `hdr.isInvariant` = 0

Incomplete Object Reports

It is possible for an object report to not contain all the instance variables of an object, due to traversal specifications or buffer size limitations. The value buffer is incomplete when `hdr.isPartial()` returns non-zero.

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 `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**

For an example of how `GciTraverseObjs` is used, see the `GciMoreTraversal` function on page 293.

**See Also**

- `GciFindObjRep`, page 255
- `GciMoreTraversal`, page 293
- `GciNbMoreTraversal`, page 314
- `GciNbStoreTrav`, page 321
- `GciNbTraverseObjs`, page 328
- `GciNewOopUsingObjRep`, page 338
- `GciObjRepSize_`, page 348
- `GciStoreTrav`, page 478
**GciUncompress**

Uncompress the supplied data, assumed to have been compressed with **GciCompress**.

**Syntax**

```c
int GciUncompress(
    char * dest,
    uint * destLen,
    const char * source,
    uint sourceLen );
```

**Input Arguments**

- **dest** Pointer to the buffer intended to hold the resulting uncompressed data.
- **destLen** Length, in bytes, of the buffer intended to hold the uncompressed data.
- **source** Pointer to the source data to uncompress.
- **sourceLen** Length, in bytes, of the source data.

**Result Arguments**

- **dest** The resulting uncompressed data.

**Return Value**

**GciUncompress** returns `Z_OK` (equal to 0) if the decompression succeeded, or various error values if it failed; see the documentation for the `uncompress` function in the GNU zlib library at [http://www.gzip.org](http://www.gzip.org).

**Description**

**GciUncompress** passes the supplied inputs unchanged to the `uncompress` function in the GNU zlib library Version 1.2.3, and returns the result exactly as the GNU `uncompress` function returns it.
See Also

GciCompress, page 151
GciUserActionInit

Declare user actions for GemStone.

Syntax

void GciUserActionInit();

Description

GciUserActionInit is implemented by the application developer, but it is called by GciInit. It enables Smalltalk to find the entry points for the application’s user actions, so that they can be executed from the database.
**GciUserActionShutdown**

Enable user-defined clean-up for user actions.

**Syntax**

```c
void GciUserActionShutdown()
```

**Description**

*GciUserActionShutdown* is implemented by the application developer, and is called when a session user action library is unloaded. It enables user-defined clean-up for the application’s user actions.
GciVersion

Return a string that describes the GemBuilder version.

Syntax

const char* GciVersion();

Description

GciVersion returns a string terminated by 0, containing fields that describe the specific release of GemBuilder. Fields in the string are delimited by a period (.).

For more version information, use the methods in class System in the Version Management category.

See Also

GciProduct, page 390
Appendix

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
- 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
- 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_CLUSTERS_BUCKETS
This appendix describes the GemStone C Statistics Interface (GCSI), a library of functions that allow your C application to collect GemStone statistics directly from the shared page cache without starting a database session.

B.1 Developing a GCSI Application

The command lines in this appendix assume that you have set the GEMSTONE environment variable to your GemStone installation directory.

Required Header Files

Your GCSI program must include the following header files:

- $GEMSTONE/include/shrpcstats.ht — Defines all cache statistics. (For a list of cache statistics, refer to the “Monitoring GemStone” chapter of the System Administration Guide for GemStone/S.)
- $GEMSTONE/include/gcsi.hf — Prototypes for all GCSI functions.
- $GEMSTONE/include/gcsierr.ht — GCSI error numbers.

Your program must define a main() function somewhere.
The GCSI Shared Library

GemStone provides a shared library, $GEMSTONE/lib/libgcsi30.so, that your program will load at runtime.

- Make sure that $GEMSTONE/lib is included in your LD_LIBRARY_PATH environment variable, so that the runtime loader can find the GCSI library. For example:

  export LD_LIBRARY_PATH=$GEMSTONE/lib:$LD_LIBRARY_PATH

- $GEMSTONE/lib/libgcsi30.so is a multi-threaded library, so your program must also be compiled and linked as a multi-threaded program.

Compiling and Linking

The $GEMSTONE/examples directory includes the sample GCSI program gsstat.cc, along with a set of sample makefiles that show how to compile the sample GCSI program, using the compilers that are used to build the GemStone product.

NOTE
It may still be possible to build your program with another compiler, as long as you specify the appropriate flags to enable multi-threading.

Whenever you upgrade to a new GemStone version, you must re-compile and re-link all your GCSI programs. This is because the internal structure of the shared cache may change from version to version. Assuming you’ve created a makefile, all you should need to do is change $GEMSTONE and rebuild.

Connecting to the Shared Page Cache

The GCSI library allows your program to connect to a single GemStone shared page cache. Once the connection is made, a thread is started to monitor the cache and disconnect from it if the cache monitor process dies. This thread is needed to prevent your program from “holding on” to the shared cache after all other processes have detached from it. In this way, your program can safely sleep for a long time without preventing the operating system from freeing and recycling shared memory should the Stone be unexpectedly shut down.
The Sample Program

The sample program `gsstat.cc` (in `$GEMSTONE/examples`) monitors a running GemStone repository by printing out a set of statistics at a regular interval that you specify. The program prints the following statistics:

- **Sess** — `TotalSessionsCount`; the total number of sessions currently logged in to the system.
- **CR** — `CommitRecordCount`; the number of outstanding commit records that are currently being maintained by the system.
- **PNR** — `PagesNeedReclaimSize`; the amount of reclamation work that is pending, that is, the backlog waiting for the GcGem reclaim task.
- **PD** — `PossibleDeadSize`; the number of objects previously marked as dereferenced in the repository, but for which sessions currently in a transaction might have created a reference in their object space.
- **DNR** — `DeadNotReclaimedSize`; the number of objects that have been determined to be dead (current sessions have indicated they do not have a reference to these objects) but have not yet been reclaimed.
- **FP** — The number of free pages in the Stone.
- **OCS** — `OldestCrSession`; the session ID of the session referencing the oldest commit record. Prints 0 if the oldest commit record is not referenced by any session, or if there is only one commit record.
- **FF** — `FreeFrameCount`; the number of unused page frames in the shared page cache.

To invoke `gsstat`, supply the name of a running Stone (or shared page cache, if running on a Gem server) and a time interval in seconds. For example:

```
% gsstat myStone 2
```

To stop the `gsstat` program and detach from the cache, issue a CTRL-C.

B.2 GCSI Data Types

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

**ShrPcMonStatSType**

Shared page cache monitor statistics.
ShrPcStnStatSType
Stone statistics.

ShrPcPgsvrStatSType
Page server statistics.

ShrPcGemStatSType
Gem session statistics.

ShrPcStatUnion
The union of all four statistics structured types: shared page cache monitor, page server, Stone, and Gem.

ShrPcCommonStatSType
Common statistics collected for all processes attached to the shared cache.

The Structure for Representing the GCSI Function Result

The structured type GcsiResultSType provides a C representation of the result of executing a GCSI function. This structure contains the following fields:

```c
typedef struct {
    signed int              processId;
    signed int              sessionId;
    ShrPcCommonStatSType cmn;
    union ShrPcStatUnion u;
}  ShrPcStatsSType;

class GcsiResultSType {
    public:
    char              vsdName[SHRPC_PROC_NAME_SIZE + 1];
    unsigned int      statType;
    ShrPcStatsSType   stats;
};
```
In addition, a set of C mnemonics support representation of the count of each process-specific structured type.

```c
#define COMMON_STAT_COUNT
(sizeof(ShrPcCommonStatSType)/sizeof(int))

#define SHRPC_STAT_COUNT
(sizeof(ShrPcMonStatSType)/sizeof(int) + \n   COMMON_STAT_COUNT)

#define GEM_STAT_COUNT
(sizeof(ShrPcGemStatSType)/sizeof(int) + \n   COMMON_STAT_COUNT)

#define PGSVR_STAT_COUNT
(sizeof(ShrPcPgsvrStatSType)/sizeof(int) + \n   COMMON_STAT_COUNT)

#define STN_STAT_COUNT
(sizeof(ShrPcStnStatSType)/sizeof(int) + \n   COMMON_STAT_COUNT)
```
GcsiAllStatsForMask

Get all cache statistics for a specified set of processes.

Syntax

```c
int GcsiAllStatsForMask(mask, result, resultSize);
```

- `unsigned int mask;
- `GcsiResultSType * result;
- `int * resultSize;

Input Arguments

- `mask` Indicates what types of processes to collect statistics for.
- `result` Address of an array of kind GcsiResultSType where statistics will be stored.
- `resultSize` Pointer to an integer that indicates the size of the result in elements (not bytes). On return, indicates the number of that were stored into `result`. Indicates the maximum number of processes for which statistics can be returned.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.h.

Example

Mask bits should be set by a bitwise OR of the desired process types. For example, to get statistics for the stone and Shared Page Cache Monitor:

```c
unsigned int mask = SHRPC_MONITOR | SHRPC_STONE;
```
GcsiAttachSharedCache

Attach to the specified shared page cache.

Syntax

```c
int GcsiAttachSharedCache(const char * fullCacheName, char * errBuf, size_t errBufSize);
```

Input Arguments

- `fullCacheName` Full name of the shared page cache, in the format `stoneName@stoneHostIpAddress`. To determine the full name of the shared cache, use the gslist -x utility.
- `errBuf` A buffer that will contain a string describing an error.
- `errBufSize` Size (in bytes) of `errBuf`.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

See Also

- GcsiAttachSharedCacheForStone, page 530
- GcsiDetachSharedCache, page 531
GcsiAttachSharedCacheForStone

Attaches this process to the specified shared page cache.

Syntax

```c
int GcsiAttachSharedCacheForStone( stoneName, errBuf, errBufSize);
```

- `stoneName`: const char *
- `errBuf`: char *
- `errBufSize`: size_t

**Input Arguments**

- **stoneName**: Name of the Stone process.
- **errBuf**: A buffer that will contain a string describing an error.
- **errBufSize**: Size (in bytes) of `errBuf`.

**Return Value**

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

**Description**

This function assumes that the cache name is `<stoneName>@<thisIpAddress>` where `thisIpAddress` is the IP address of the local machine. This function may fail if the host is multi-homed (has more than one network interface). In that case, use `GcsiAttachSharedCache` (page 529) to specify the full name of the shared cache.

**See Also**

- GcsiAttachSharedCache, page 529
- GcsiDetachSharedCache, page 531
GcsiDetachSharedCache

Detach from the shared page cache.

Syntax

int GcsiDetachSharedCache (void);

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

See Also

GcsiAttachSharedCache, page 529
GcsiAttachSharedCacheForStone, page 530
GcsiFetchMaxProcessesInCache

Return the maximum number of processes that can be attached to this shared cache at any instant. The result may be used to allocate memory for a calls to the GcsiFetchStatsForAll* family of functions.

Syntax

```c
int GcsiFetchMaxProcessesInCache(int *maxProcesses);
```

Input Arguments

- `maxProcesses` The maximum number of processes that can be attached to this shared cache at any instant.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.
GcsilInit

Initialize the library. This function must be called before all other GCSI functions.

Syntax

GcsilInit(void);
**GcsiShrPcMonStatAtOffset**

Get the SPC monitor cache statistic at the given byte offset within the ShrPcMonStatSType structure type.

**Syntax**

```c
int GcsiShrPcMonStatAtOffset(byteOffset, stat);
size_t byteOffset;
unsigned int * stat;
```

**Input Arguments**

- `byteOffset` Offset (in bytes) of the desired statistic in the ShrPcStatUnion type.
- `stat` Value of the requested statistic.

**Return Value**

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

**See Also**

GcsiStnStatAtOffset, page 535
GcsiStnStatAtOffset

Get the Stone cache statistic at the given byte offset within the ShrPcStnStatSType structure type.

Syntax

```c
int GcsiStnStatAtOffset(byteOffset, stat);
size_t byteOffset;
unsigned int * stat;
```

Input Arguments

- `byteOffset` Offset (in bytes) of the desired statistic in the ShrPcStatUnion type.
- `stat` Value of the requested statistic.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

See Also

GcsiInit, page 533
GcsiStatsForGemSessionId

Get the cache statistics for the given Gem session id.

Syntax

```c
int GcsiStatsForGemSessionId(sessionId, result);
```

```c
int sessionId;
GcsiResultSType * result;
```

Input Arguments

- `sessionId`: Session ID of the Gem for which statistics are requested.
- `result`: Pointer to a GcsiResultSType structure.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

See Also

- GcsiStatsForGemSessionWithName, page 537
- GcsiStatsForPgsvrSessionId, page 538
- GcsiStatsForProcessId, page 539
- GcsiStatsForShrPcMon, page 540
- GcsiStatsForStone, page 541
GcsiStatsForGemSessionWithName

Get the cache statistics for the first Gem in the cache with the given cache name.

Syntax

```c
int GcsiStatsForGemSessionWithName(gemName, result);
char * gemName;
GcsiResultSType * result;
```

Input Arguments

- **gemName**: The case-sensitive name of the Gem for which statistics are requested.
- **result**: Pointer to a GcsiResultSType structure.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

See Also

- GcsiStnStatAtOffset, page 535
- GcsiStatsForPgsvrSessionId, page 538
- GcsiStatsForProcessId, page 539
- GcsiStatsForShrPcMon, page 540
- GcsiStatsForStone, page 541
GcsiStatsForPgsvrSessionId

Get the cache statistics for the given page server with the given session id. Remote Gems have page servers on the Stone’s cache that assume the same session ID as the remote Gem.

Syntax

```c
int GcsiStatsForPgsvrSessionId(sessionId, result);
int sessionId;
GcsiResultSType * result;
```

Input Arguments

- `sessionId`: Session ID of the page server for which statistics are requested.
- `result`: Pointer to a GcsiResultSType structure.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.h.

See Also

- GcsiStatsForGemSessionId, page 536
- GcsiStatsForGemSessionWithName, page 537
- GcsiStatsForProcessId, page 539
- GcsiStatsForShrPcMon, page 540
- GcsiStatsForStone, page 541
GcsiStatsForProcessId

Get the cache statistics for the given process ID.

Syntax

```c
int GcsiStatsForProcessId(pid, result);
int pid;
GcsiResultSType * result;
```

Input Arguments

- `pid` : Process ID for which statistics are requested.
- `result` : Pointer to a GcsiResultSType structure.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.h.

See Also

- GcsiStatsForGemSessionId, page 536
- GcsiStatsForGemSessionWithName, page 537
- GcsiStatsForPgsvrSessionId, page 538
- GcsiStatsForShrPcMon, page 540
- GcsiStatsForStone, page 541
**GcsiStatsForShrPcMon**

Get the cache statistics for the shared page cache monitor process for this shared page cache.

**Syntax**

```c
int GcsiStatsForShrPcMon(result);
GcsiResultSType * result;
```

**Input Arguments**

`result` Pointer to a GcsiResultSType structure.

**Return Value**

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

**See Also**

GcsiInit, page 533  
GcsiStatsForGemSessionId, page 536  
GcsiStatsForGemSessionWithName, page 537  
GcsiStatsForPgsvrSessionId, page 538  
GcsiStatsForProcessId, page 539  
GcsiStatsForStone, page 541
**GcsiStatsForStone**

Get the cache statistics for the Stone if there is a Stone attached to this shared page cache.

**Syntax**

```c
int GcsiStatsForStone(result);
GcsiResultSType * result;
```

**Input Arguments**

`result` Pointer to a GcsiResultSType structure.

**Return Value**

Returns 0 if successful; otherwise returns an error code, as defined in gcsierr.ht.

**See Also**

GcsiStnStatAtOffset, page 535  
GcsiStatsForGemSessionId, page 536  
GcsiStatsForGemSessionWithName, page 537  
GcsiStatsForPgsvrSessionId, page 538  
GcsiStatsForProcessId, page 539  
GcsiStatsForStone, page 541
## GCSI Errors

The following errors are defined for the GemStone C Statistics Interface.

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCSI_SUCCESS</td>
<td>The requested operation was successful.</td>
</tr>
<tr>
<td>GCSI_ERR_NO_INIT</td>
<td>GcsiInit() must be called before any other Gcsi functions.</td>
</tr>
<tr>
<td>GCSI_ERR_CACHE_ALREADY_ATTACHED</td>
<td>The requested shared cache is already attached.</td>
</tr>
<tr>
<td>GCSI_ERR_NOT_FOUND</td>
<td>The requested session or process was not found.</td>
</tr>
<tr>
<td>GCSI_ERR_BAD_ARG</td>
<td>An invalid argument was passed to a Gcsi function.</td>
</tr>
<tr>
<td>GCSI_ERR_CACHE_CONNECTION_SEVERED</td>
<td>The connection to the shared cache was lost.</td>
</tr>
<tr>
<td>GCSI_ERR_NO_STONE</td>
<td>Stone statistics were requested on a cache with no stone process.</td>
</tr>
<tr>
<td>GCSI_ERR_CACHE_NOT_ATTACHED</td>
<td>No shared page cache is currently attached.</td>
</tr>
<tr>
<td>GCSI_ERR_NO_MORE_HANDLES</td>
<td>The maximum number of shared caches are attached.</td>
</tr>
<tr>
<td>GCSI_ERR_CACHE.Attach_FAILED</td>
<td>The attempt to attach the shared cache has failed.</td>
</tr>
<tr>
<td>GCSI_ERR_WATCHER_THREAD_FAILED</td>
<td>The cache watcher thread could not be started.</td>
</tr>
<tr>
<td>GCSI_ERR_CACHE_WRONG_VERSION</td>
<td>The shared cache version does not match that of the libgcsixx.so library.</td>
</tr>
</tbody>
</table>
A
aborting transactions 25
adding
  OOPs to an indexable object (Collection) 38, 465, 468
  OOPs to an NSC 39, 115, 117
alignment of traversal buffer 119
allocating
  multiple OOPs 262, 264
  OOPs 260
altered objects
  finding 121
appending
  to a byte object 124, 125
  to a collection 126
application
  binding 55, 81
  compiling 78
  improving performance 41, 44, 237, 293, 314, 321, 323, 328, 398, 399, 400, 401, 405, 471, 478, 482, 486, 510
  linking 20
  application user action 67
  application user actions 58
authorization
  traversal 513
  violation, what to do 25

B
beginning
  a transaction 127
binding to GemBuilder 55, 81
boolean
  converting to an object 128
  represented as a special object 27
byte array
  converting to a C pointer 130
byte object
  creating 330
  fetching bytes from 35, 204, 206
  fetching characters from 209
  fetching the size 222, 246
implementation type 35, 228
initializing 330
storing bytes in 36, 445, 447, 449
bytes
appending 124

cluster bucket
mnemonic for category 522
committing transactions 25, 147, 301
compiling
applications 78
C code 76
class methods 22, 141, 149
instance methods 22, 277
methods 22, 277
user actions 78
compressing objects 151, 515
concurrency conflict 147, 301
what to do 25
configuration files 274, 275
constraint violation, what to do 25
context
call stack 145
error handling 49
of GemStone system 154, 156, 302, 303
continuable error 154, 156, 302, 303
continuing
after an error 48, 154, 156, 302, 303
traversal 294, 513
controlling
sessions 147, 289, 291, 301, 432
transactions 114, 147, 296, 301
converting between
byte arrays and pointers 130, 383
objects and booleans 128, 354, 356
objects and characters 134, 357, 358, 359,
361
objects and floating point numbers 182,
258, 362
objects and integers 270, 364, 365, 366, 367
special objects and C values 27
strings and integers 492
v1.1.1 oops and OopTypes 350
creating
byte object 330
character object 331
class methods 141, 149
database objects 21
DateTime object 332

C

C mnemonic
sizes and offsets into objects 88
C types defined for GemBuilder functions 88, 89
call
determining if in progress 131
call stack
clearing 49, 145
calling
the virtual machine 154, 191, 195, 198,
302, 306, 308, 310, 315, 371, 373,
375, 377, 379
user actions
from GemStone 66
changed object, and re-reading 132
changing class definitions 33
character
converting to an object 134
instance defined in GemStone 522
represented as a special object 27
character object
creating 331
initializing 331
characters
converting to objects 134
checking for GemBuilder errors 48, 189
clamped object traversal 135, 138, 299, 324,
486, 488
structured type 135, 202
class
compiling methods 22, 141, 149
fetching an object’s 35, 211
modifying 33
object report 255
clearing the call stack 49

character object
creating 331
initializing 331
characters
converting to objects 134
checking for GemBuilder errors 48, 189
clamped object traversal 135, 138, 299, 324,
486, 488
structured type 135, 202
class
compiling methods 22, 141, 149
fetching an object’s 35, 211
modifying 33
object report 255
clearing the call stack 49

VMware, Inc.
June 2011
GemStone sessions 24, 289, 432
instances of a GemStone class 34
objects 40, 333, 335, 338
OOPs 333, 335

current session, defined 24

D
date, time
   structured type 90

DateTime object
   creating 332
   fetching contents of 213
   initializing 332

debugging 430
   function, enabling 164, 166, 167
   information, finding 391, 423, 425, 427
   use GciRpc 54, 81
   user action 67

decoding
   an OOP array 170

decrementing shared counter 172
default
   directory, host file access 47
   login parameter value 273, 432
defining
   new methods 22
deprecated functions
   GciPathToStr 368
   GciStrToPath 497
developing a user action 59
dirty objects 121, 174, 176, 178, 180, 507, 509
disabling
   error handlers 423, 425, 427
dynamic instance variable
   fetching contents of 214, 215, 453

E
enabling
   debugging functions 164, 166, 167
   error handlers 423, 425, 427
   full compression 184
   run-length encoding 183
   signaled errors 185
encoding
   an OOP array 187
   free OOPs 183
   enumerating named instance variables 143, 285
environmentId 101
error
   checking 48, 189, 430
   continuing execution after 154, 156, 302, 303
   dictionary 27, 522
   handling 386, 391, 423, 425, 427
   jump buffer 48, 386, 391, 423, 425, 427
   mnemonics 88
   polling 48, 189, 384
error report
   structured type 90
errors 292, 430, 431
   signaled 185
executing code in
   GemBuilder, advantages over GemStone 21
   advantages over GemBuilder 22
   host file access method 47
executing user action 67
execution environment
   and Ruby 101
export set 121, 174, 176, 178, 180, 419, 421, 422, 507, 509
exporting objects to GemStone 21, 34

F
false, GemStone special object 27, 28, 128, 521
fetching
   bytes from a byte object 35, 204, 206
   characters from a byte object 209
   class 35, 211
   DateTime 213
dynamic instance variables 214, 215
number of shared counters 225
object implementation format 35, 228
object information 226, 229
object size 222, 246, 253
objects by using paths 44, 237
OOPs from a pointer object 37, 216, 219, 248, 251
OOPs from an indexable object
(Collection) 38, 231, 234
OOPs from an NSC 39, 231, 234
shared counter value 244, 394, 395
finding debugging information 391, 423, 425, 427
object reports in a traversal buffer 255
objects in a traversal buffer 43
float kind 257
floating point number as a byte object 36
converting to an object 258
format of an object fetching 228
format of an object, fetching 35
free OOPs run-length encoding 183
full compression enabling 184

garbage collection 398, 399, 400, 401, 405, 419, 421, 422
saving and releasing objects 49
GciAbort 114
GciAddOopsToNsc 39, 117
GciAddOopToNsc 39, 115
GCL_ALIGN 119
GciAllocTravBuf 120
GciAlteredObjs 121
GciAppendBytes 30, 124
GciAppendChars 125
GciAppendOops 126
GciBegin 127

GCL_BOOL_TO_OOP 128
GciByteArrayToPointer 130
GciCallInProgress 46
GciCheckAuth 132
GCI_CHR_TO_OOP 134
GciClampedTrav 135
GciClampedTravArgsSType 135, 202
GciClampedTraverseObjs 138
GciClassMethodFromClass 141
GciClassNamedSize 143
GciClearStack 145
GciCommit 147
GciCompileMethod 149
GciCompress 151
GciContinue 154
GciContinueWith 156
GciCreateByteObj 158
GciCreateOopObj 160
GciCTimeToDateTime 162
GciDateTimeSType 90
GciDateTimeToCTime 163
GciDbgEstablish 164
GciDbgEstablishToFile 166
GciDbgLogString 167
GciDeclareAction 61, 168
GciDecodeOopArray 170
GciDecSharedCounter 172
GciDirtyExportedObjs 174
GciDirtyObjsInit 176
GciDirtySaveObjs 178
GciDirtyTrackedObjs 180
Gci_doubleToSmallDouble 182
GciEnableFreeOopEncoding 183
GciEnableFullCompression 184
GciEnableSignaledErrors 185
GciEncodeOopArray 187
GciErr 189
GciErrSType 90
GciExecute 31, 191
GciExecuteFromClass 31, 193
GciExecuteStr 31, 195
GciExecuteStrFromClass 198
GciExecuteStrTrav 201
GciFetchByte 204
GciFetchBytes_ 206
GciFetchChars_ 209
GciFetchClass 211
GciFetchDateTime 213
GciFetchDynamicIv 214
GciFetchDynamicIvs 215
GciFetchNamedOop 37, 216
GciFetchNamedOps 37, 219
GciFetchNamedSize 222
GciFetchNameOfClass 223
GciFetchNumEncodedOops 224
GciFetchNumSharedCounters 225
GciFetchObjectInfo 226
GciFetchObjImpl 228
GciFetchObjInfo 229
GciFetchObjInfoArgsSType 226
GciFetchOop 38, 39, 231
GciFetchOops 38, 39, 234
GciFetchPaths 237
GciFetchSharedCounterValuesNoLock 244
GciFetchSize_ 246
GciFetchVaryingOop 248
GciFetchVaryingOops 251
GciFetchVaryingSize_ 253
GciFindObjRep 255
GciFloatKind 257
GciFloatKindEType 257
GciFltToOop 258
GciGetFreeOop 260
GciGetFreeOops 262
GciGetFreeOopsEncoded 264
GciGetSessionId 265
GciHardBreak 46, 267
GciHiddenSetIncludesOop 268
GCI_I64_IS_SMALL_INT 269
Gci64ToOop 270
GciIncSharedCounter 271
GciInit 46, 273
GciInstallUserAction 276
GciInstMethodForClass 277
GciInUserAction 279
GciIsKindOfClass 280
GciIsRemote 282
GciIsSubclassOf 283
GciIsSubclassOfClass 284
GciIvNameToIdx 285
GciLnk
  configuration file 274
  GciIsRemote 282
  Loading 415, 416, 418
  object traversal function 54
  path access function 54
  use only with debugged applications 81
  use to enhance performance 54
  user action 70
GciLoadUserActionLibrary 287
GciLogin 289
GciLogout 291
GciLongImp 48, 391
GciMoreTraversal 293
GciNbAbort 296
GciNbBegin 297
GciNbClampedTrav 298
GciNbClampedTraverseObjs 299
GciNbCommit 301
GciNbContinue 302
GciNbContinueWith 303
GciNbEnd 304
GciNbExecute 306
GciNbExecuteStr 308
GciNbExecuteStrFromContext 310
GciNbExecuteStrTrav 312
GciNbMoreTraversal 314
Gci NbPerform 315
GciNbPerformNoDebug 317
Gci NbPerformTrav 319
GciNbStoreTrav 321
GciNbStoreTravDoTrav 324
GciNbStoreTravDoTravRefs 326
GciNbTraverseObjs 328
GciNewByteObj 330
GciNewCharObj 331
GciNewDateTime 332
GciNewOop 333
GemStone/S 64 Bit 3.0 GemBuilder for C

VMware, Inc. June 2011
GciStoreBytesInstanceOf 449
GciStoreChars 30, 451
GciStoreDynamicIv 453
GciStoreIdxOop 454
GciStoreIdxOops 456
GciStoreNamedOop 37, 459
GciStoreNamedOops 37, 462
GciStoreOop 38, 455
GciStoreOops 38, 455
GciStorePaths 471
GciStoreTrav 478
GciStoreTravDo_ 482
GciStoreTravDoTrav_ 486
GciStoreTravDoTravRefs 488
GciStringToInteger 492
GciStrKeyValueDictAt 493
GciStrKeyValueDictAtObj 494
GciStrKeyValueDictAtObjPut 495
GciStrKeyValudDictAtPut 496
GciStrToPath 497
GciSwapBytesUint 500
GciSwapBytesUshort 501
GciSymDictAt 502
GciSymDictAtObj 504
GciSymDictAtObjPut 505
GciSymDictAtPut 506
GciTrackedObjs and garbage collection 400, 405
GciTrackedObjsFetchAllDirty 507
GciTrackedObjsInit 509
GciTraverseBufType 100
allocating 120
GciTraverseObjs 510
GciUncompress 515
GciUserActionInit 61
GCI_USER_ACTION_INIT_DEF 61
GciUserActionShutdown 61, 62
GCI_USER_ACTION_SHUTDOWN_DEF 62
GciUserActionSType 99
GciVersion 519
GCSI
compiling and linking 524
connecting to shared page cache 524

data types 525
errors 542
function library 523
sample program
explained 525
introduced 524
shared library 524

GcsiAllStatsForMask 528
GcsiAttachSharedCache 529
GcsiDetachSharedCache 531
GcsiFetchMaxProcessesInCache 532
GcsiInit 533
GcsiResultSType 526
GcsiResultSType (structured type) 526
GcsiShrPcMonStatAtOffset 534
GcsiStatsForGemSessionId 536
GcsiStatsForGemSessionWithName 537
GcsiStatsForPgsvrSessionId 538
GcsiStatsForProcId 539
GcsiStatsForShrPcMon 540
GcsiStatsForStone 541
GcsiStnStatAtOffset 535

GemBuilder
initializing 273
libraries 54
library file
gcirpc50.* 55
libgcklnk.* 55
loading 415, 416, 418
run-time binding 55, 81
starting 273
stopping 440

GemBuilder errors 292, 430, 431
GemRpc, user action 71
GemStone C Statistics Interface, see GCSI 523
GemStone service name 432
GemStone-defined object, making available to applications 27
gssstat.cc, sample GCSI program 524
**H**
handling errors 386, 391, 423, 425, 427
hard break 114, 267, 296
defined 33
hidden set 268
host
  file access, default directory 47
  password 432
  username 432
host-specific C definition 88

**I**
implementation of an object
  fetching 35, 228
  object report 138, 255, 299, 324, 379
implementing a user action 59
importing objects
  from GemStone 21, 34
improving application performance 41, 44,
  237, 293, 314, 321, 323, 328, 398, 399,
  400, 401, 405, 471, 478, 482, 486, 510
include file (GCI)
  gci.ht 88
  gcioop.ht 521
include file (GemBuilder)
  flag.ht 88
  gci.ht 87
  gci.ht 26, 89
  gcicmn.ht 88
  gcier.ht 88
  gcifloat.hf 88
  gcioe.hf 88
  gcioop.ht 27, 28, 88
  gcirtl.hf 87, 88
  gcirtl.hf 88
  gcirtm.hf 88
  gciscend.hf 88
  gcia.hf 87
  gciauser.hf 88
  version.hf 88
incomplete
  object report 513

incrementing shared counter 271
indexable instance variable, fetching the value
  of an object’s 251
indexable object (Collection)
  adding OOPs to 38, 465, 468
  fetching OOPs from 38, 231, 234
initializing
  byte object 330
  character object 331
  DateTime object 332
  objects 333, 335
  OOPs 333, 335
initializing GemBuilder 23, 273
initiating a GemStone session 289
installing a user action 168, 276
instance
  GemStone-defined 521
  method, compiling 22, 277
  variable 36
    enumerating for a class 143, 285
interrupt
  GemStone (hard break) 33
  handling 32, 46
  issuing 33, 114, 267, 296, 441
  virtual machine (soft break) 32, 154, 302,
    441
interrupts 131

**J**
jump buffer, error handling in GemBuilder 48,
  386, 391, 423, 425, 427

**K**
kernel class 333, 335
  mnemonics 27, 522
kind
  of a class 280, 281
level traversal 42, 293, 314, 328, 510
library
  GemBuilder 54
  run-time loading 63
  search 56
  user action 60
linkable GemBuilder (GciLnk)
  GciIsRemote 282
  use to enhance performance 54
linking
  applications 20
  applications and user actions 69, 70
loading
  user action 63
loading GemBuilder 415, 416, 418
logging in to GemStone 24, 289, 432
logging out from GemStone 24, 291
logical access to objects 22, 30
login parameter 289
longjmp, setjmp
  equivalent functionality 48, 292, 391, 423, 425, 427, 431
macros defined 88
manual, organization of 4
memory
  reallocating 397
message
  GemBuilder function 31
  sending 30, 315, 371, 373, 375, 377, 379
method
  calling C functions from 168, 276
  compiling 22, 141, 149, 277
mnemonic
  GemStone error 48, 88
modifying
  objects directly in C 21, 34
  caution 34
  multiple GemStone sessions 70

GciRpc 54
  switching among 265, 435, 438
multiple objects
  defining 338
  exporting 255, 293, 314, 321, 323, 324, 328, 471, 478, 482, 486, 488, 510
  importing 237, 255, 293, 314, 321, 323, 324, 328, 478, 482, 486, 488, 510

named instance variable
  fetching 216, 219
  number of 143, 222, 285
  pointer object 36
network 432
  minimizing traffic 41, 44, 237, 293, 314, 321, 323, 328, 471, 478, 482, 486, 510
  node 432
  parameter 289, 432
  traffic, minimizing 44
nil, GemStone special object 27, 28, 521
node name, network 432
nonblocking functions 44
non-sequenceable collection
  searching 343
non-sequenceable collection (NSC) 38
  adding OOPs to 39, 115, 117
  fetching OOPs from 39, 231, 234
  fetching the size 222, 246
  implementation type 35, 38, 228
  removing OOPs from 39, 406, 408
number of an object’s instance variables
  object report 138, 255, 299, 324, 379
number of named instance variables in a class 143
number, converting to an object 258
numeric representation of a path 237, 471
object byte implementation type 35
control function 398, 399, 400, 401, 405, 419, 421, 422
converting to
  boolean 354, 356
  character 357, 358, 359, 361
  floating-point number 362
  integer 270, 364, 365, 366, 367
creating 40, 338
identity 26
importing or exporting multiple 41
mnemonic 27
NSC implementation type 38
pointer implementation type 36
releasing 49, 291, 398, 399, 400, 401, 405
  finding in a traversal buffer 255
incomplete 513
size 348
special objects 512
structure summary 43
traversal buffer 42
word alignment 119
representation in C 21, 34
saving 49
  sending messages 315, 371, 373, 375, 377, 379
object information
  fetching 226, 229
  structured type 92, 226
object report
  structured type 94, 202
object report header
  structured type 95
object traversal 377
objects
  creating 333, 335
  initializing 333, 335
  saving from garbage collection 419, 421,

422
objectSecurityPolicyId, in GciObjRepHdrSType 95
OOP (object-oriented pointer)
  adding to an indexable object (Collection)
    38, 465, 468
  adding to an NSC 39, 115, 117
  defined 26
  fetching from an indexable object
    (Collection) 38, 231, 234
  fetching from an NSC 39, 231, 234, 248, 251
  removing from an NSC 39, 406, 408
  searching an NSC for 343
  searching for 268
OOP array
  decoding 170
  encoding 187
  obtaining the size of 224
OOPs
  allocating 260, 262, 264
  appending 126
  creating 333, 335
  initializing 333, 335
operating system considerations 46

P
password
  GemStone 289, 432
  host 432
path access
  function, GciLnk 54
  function, GciRpc 54
  to objects 44, 237, 471
pause message 154, 156, 302, 303
performance, improving application 41, 44, 237, 293, 314, 321, 323, 328, 398, 399, 400, 401, 405, 471, 478, 482, 486, 510
persistance and user action results 59
pointer
  converting to a byte array 383
pointer object
  fetching OOPs from 37, 216, 219, 248, 251
fetching the size 222, 246
implementation type 35, 36, 228
storing OOPs in 37, 454, 456, 459, 462,
  465, 468
polling for GemBuilder errors 48, 189
polling for signal errors 384
prerequisites 3
primitive, user-defined 168, 276
private method, compilation restrictions 141,
  149, 277
ProfMonitor
  and signals 102

R
reallocating
  memory 397
reclaiming storage 398, 399, 400, 401, 405
ReferencedSet 490
releasing objects 49, 291, 398, 399, 400, 401,
  405
remote procedure call GemBuilder (GciRpc)
  282
  use in debugging your application 54
removing OOPs from an NSC 39, 406, 408
report, of an object 138, 255, 293, 299, 314, 321,
  323, 324, 328, 338, 348, 379, 478, 482,
  510
re-reading objects from the database 25
reserved OOP 27
resolving symbols 31, 141, 149, 191, 195, 198,
  201, 277, 306, 308, 310, 312
Ruby
  and GCI execution environment 101
run-length encoding 170, 187, 224
enabling 183
run-time binding
  GemBuilder 55, 81
run-time loading 63

S
saving objects 49
  export set 50
schema 20
security policy
  GciCheckAuth function 132
security policy ID in Object Information
  Structure 92
security policy, in data structure 95
sending messages to GemStone objects 22, 30,
  315, 371, 373, 375, 377, 379
service name, GemStone 432
session
  control 23, 289, 291, 432
  creating (logging in) 24, 432
  current 24
  defined 23
  finding the current ID number 265
  halting on error 430
  setting the current ID number 435, 438
  switching among multiple 265, 435, 438
  terminating (logging out) 24, 291
session user action 67
session user actions 58
setjmp, longjmp
  equivalent functionality 48, 292, 391, 423,
    425, 427, 431
shared counter
  decrementing 172
  fetching value 244, 394, 395
  finding how many 225
  incrementing 271
  setting value 437
shared libraries
  GemBuilder 54
  user action 57
SIGIO 46
signal
  handling 46
signal (system function) 46
signal errors 384
signal handling
  UNIX 102
signaled errors 185
single-step execution 444
size of an object
  fetching 222, 246
  object report 255
size of an object report, calculating 348
size of an OOP array 224
SmallInteger
  represented as a special object 27
soft break 154, 302, 441
defined 32
special object
  implementation type 35, 228
  object report 512
  traversal of 512
stack
  clearing the call 145
starting GemBuilder 23, 273
statistics, collecting directly from shared page cache 523
stopping GemBuilder 23, 440
storing
  bytes in a byte object 36, 445, 447, 449
dynamic instance variables 453
  objects by using paths 44, 471
OOPs in a pointer object 37, 454, 456, 459, 462, 465, 468
string
  appending 125
  as a byte object 35
  converting to an integer 492
  fetching 36, 204, 206, 209
  storing 36, 445, 447, 451
function 101
calculation when using 101, 111
structured types
  GciClampedTravArgsSType 135, 202
  GciDateTimeSType 90
  GciErrSType 90
  GciFetchObjInfoArgsSType 226
  GciObjInfoSType 92
  GciObjRepHdrSType 95
  GciObjRepSType 94, 202
  GciTravBufType 100
  GciUserActionSType 99
subclass
determining 283, 284
switching among multiple GemStone sessions 265, 435, 438
symbol
  as a byte object 35
  creating 342
  resolution 31, 141, 149, 191, 193, 195, 198, 201, 277, 306, 308, 310, 312
T
  terminating GemStone sessions 291
testing an application
  use GciRpc 81
tracing a GemBuilder call while debugging 164, 166, 167
tracked objects 507, 509
transaction
  aborting 25, 114, 296
  beginning a 127
  committing 25, 147, 301
  control 114, 147, 296, 301
  management 132
  workspace, creating 289, 432
  workspace, terminating 291
  buffer 42, 293, 314, 321, 323, 324, 328, 478, 482, 486, 488, 510
  finding object reports 43, 255
  word alignment 119
function
  GciLnk 54
  GciRpc 54
  inability to continue 294, 513
  level 42, 293, 314, 328, 510
special object 512
structured type 135, 202
threshold 138, 293, 299, 314, 321, 323, 324, 328, 377, 478, 482, 486, 488, 510
word alignment 119
traversal buffer
allocating 120
structured type 100
true, GemStone special object 27, 28, 128, 521

U
uncommitted object, releasing 49, 291, 398, 399, 400, 401, 405
uncompressing objects 515
underscore character, private method 141, 149, 277
UNIX signal handling 102
unnamed instance variable, fetching 248, 251
updating the C representation of database objects 120, 121, 132, 134
user action 279
application 58, 67
calling from GemStone 66
compiling 78
configurations 67
debugging 67
defined 57
executing 67
implementing 59
include file 88, 89
installation macro defined 88, 89
installation verified 65
installing 168, 276
library 60
loading 287
linked application 70
loading 63
making results persistent 59
RPC application 69
run-time loading 63
session 58, 67
structured type 99
userAction instance method 66
user action libraries 57
user actions
GciUserActionInit 517
GciUserActionShutdown 518
user name
GemStone 289, 432
host 432
user profile, searching the symbol list in 141, 149, 191, 195, 198, 277, 306, 308, 310
user session
creating 289, 432
terminating 291
user, searching the symbol list for 31

V
value buffer
object report 255, 293, 314, 321, 323, 324, 328, 338, 478, 482, 510
word alignment 119
value of an instance variable, object report 138, 255, 299, 324, 379
version
GemBuilder 519
virtual machine
call stack 49
clearing 49, 145

W
word alignment 119