GemBuilder™ for C for GemStone/S 64 Bit™

Version 3.5

June 2019

GEMTALK™ SYSTEMS

GemTalk Systems LLC
15220 NW Greenbrier Parkway
Suite 240
Beaverton, OR 97006

Preface

About This Manual

This manual describes GemBuilder™ for C — a library 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 program with complete access to a GemStone database of objects, and to a virtual machine on which to execute GemStone Smalltalk code.

This manual describes the classic GemBuilder for C interface, which is not thread-safe. A newer, thread-safe interface is provided by gcits.hf. This is not described in this manual; see the C header file, `$GEMSTONE/include/gcits.hf`, for usage details.

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

You should have the GemStone system installed as described in the *GemStone/S 64 Bit Installation Guide* for your platform.

Terminology Conventions

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

Technical Support

Support Website

gemtalksystems.com

GemTalk's website provides a variety of resources to help you use GemTalk products:

- **Documentation** for the current and for previous released versions of all GemTalk products, in PDF form.
- **Product download** for the current and selected recent versions of GemTalk software.
- **Bugnotes**, identifying performance issues or error conditions that you may encounter when using a GemTalk product.
- **Supplemental Documentation** and **TechTips**, providing information and instructions that are not in the regular documentation.
- **Compatibility matrices**, listing supported platforms for GemTalk product versions.

We recommend checking this site on a regular basis for the latest updates.

Help Requests

GemTalk Technical Support is limited to customers with current support contracts. Requests for technical assistance may be submitted online (including by email), or by telephone. We recommend you use telephone contact only for urgent requests that require immediate evaluation, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: techsupport.gemtalksystems.com

Email: techsupport@gemtalksystems.com

Telephone: (800) 243-4772 or (503) 766-4702

Please include the following, in addition to a description of the issue:

- The versions of GemStone/S 64 Bit and of all related GemTalk products, and of any other related products, such as client Smalltalk products, and the operating system and version you are using.
- Exact error message received, if any, including log files and statmonitor data if appropriate.

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

24x7 Emergency Technical Support

GemTalk 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 GemTalk Support Renewals.

Table of Contents

Chapter 1 Introduction

GemBuilder™ for C is a library of C functions that provide access to the GemStone/S 64 Bit™ repository and the GemStone Smalltalk programming language from C applications, or from other environments that can invoke C functions.

The GemBuilder for C functions are provided by shared libraries that are distributed with the GemStone/S 64 Bit product. These shared libraries are used by the GemStone/S 64 Bit topaz interface, and by the GemBuilder for Smalltalk and GemBuilder for Java products. These libraries can be used from any environment that allows you to invoke C libraries, including Smalltalk implementations such as Dolphin and Pharo, and other languages such as Ruby and Python.

The GemStone object server contains your schema (class definitions) and objects (instances of those classes), while your GCI client program provides the user interface. Logic may be implemented on the client, server, or both; the GemBuilder functions allow your GCI application to create, read, and write objects in the GemStone repository either by invoking Smalltalk code or by using structural access and basic C data types.

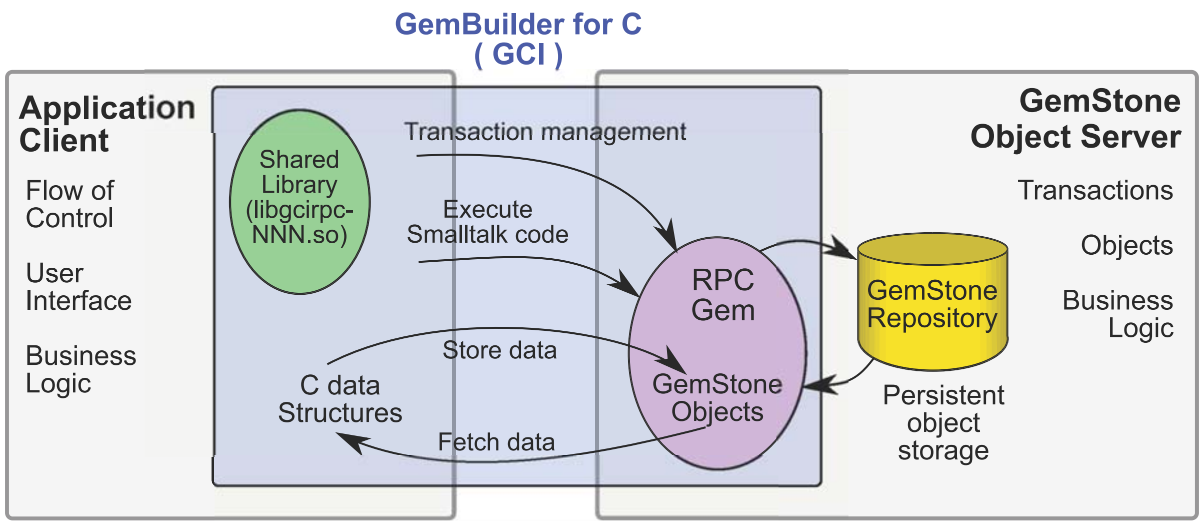
The following interfaces allow your GemStone application to invoke C functions:

- GCI Application — a standalone application in C, C++, or another language, in which the C API is invoked according to the rules for that environment, and the final application is compiled or packaged for use according to the rules for that language. This manual describes using C or C++ to invoke GemBuilder functions, and provides compile and link information for C/C++.
- User Actions — user actions provide a way for GemStone Smalltalk to invoke C/C++ code that you have compiled into shared libraries. This is similar to the concept of "user-defined primitives" offered by other object-oriented systems. The functions themselves are written as for a standalone GCI application, although there are some differences in handling objects. This manual provides information on how to define the API and compile and link User Action libraries.
- Foreign Function Interface (FFI) — the GemStone FFI interface allows you to interface to C/C++ libraries. This interface is designed to invoke third-party libraries, although it can be used to invoke the GemBuilder C functions. The FFI interface is described in the *Programming Guide*, and is not covered in this manual.

1.1 GemBuilder Application Overview

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

Figure 1.1 The Role of GemBuilder in Application Development



We recommend the following steps for developing your hybrid application:

Step 1. Define the application’s external interface.

GemBuilder-based applications must manage the user interface using language-specific modules. GemStone and GemBuilder for C have no graphical user interface.

Step 2. Decide where to perform the work.

Applications that are a hybrid of client-side code and GemStone 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 C data types and perform the work on the client, or to send a message which invokes a Smalltalk method on the server?

Step 3. Implement and debug the application.

After you’ve developed a satisfactory design, you can implement and test the client functions using language-specific techniques and tools.

Step 4. Compile and link the application.

For instructions about compiling and linking your application, please see Chapter 5, “Compiling and Linking” For full details, see your C compiler user documentation.

1.2 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 on the client and how much in Smalltalk methods on the server. 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 "Structural Access to Objects" on page 30 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 “Performing work in GemStone” on page 28 defines GemBuilder functions that allow applications to send Smalltalk messages to objects and execute Smalltalk code.

Environments

The GemStone server includes multiple execution environments, although normally Smalltalk applications exclusively use the primary environment, environment 0.

The GCI functions that support an environmentId argument include GciExecute* and GciPerform*, and non-blocking variants of these, that have names with a trailing underscore. These functions allow you to access other environments. These further environments were primarily intended to support Ruby applications, and are not described in this manual. Examine the header file `$GEMSTONE/include/gci.hf` for details.

1.3 Objects in GemStone and C data

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 64-bit object-oriented pointer, or OOP. Whenever your client application 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 the client.

Your client application can only use predefined OOPs, or OOPs that it has received from the GemStone server; it cannot create new OOPs directly.

Since the client application accesses objects only via the OOP, the terms "OOP" and "Object" are used interchangeably in this manual, in function names, and in header comments. This differs from GemStone Smalltalk code and other GemTalk manuals, in which "OOP" is generally used to distinguish the numeric pointer value associated with an object from the object itself.

1.4 Garbage Collection

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

Garbage Collection includes

- in-memory garage collection of non-persistent temporary objects in each client process
- on the server, periodic sweeps of the entire repository to detect persistent objects that are no longer referenced.

In-memory garbage collection

In-memory garbage collection occurs regularly in the Gem process, to ensure memory is available for ongoing needs. 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. While you may have a reference the OOP of a temporary objects from data structures in your GCI application, this does not by itself prevent the actual object in the Gem process from being garbage collected. In-memory GarbageCollection can occur at any time, particularly if the GemStone session is in active use.

To avoid this problem, GemStone uses several internal sets in the Gem process, including the *PureExportSet* and the *GciTrackedObjs* set, that hold objects that need to be preserved from garbage collection. Before removing any objects, the GemStone in-memory garbage collector checks the *PureExportSet* and the *GciTrackedObjs* sets in the Gem's workspace. Any object in these sets is considered to be referenced. User actions maintain their own export set, and objects in that set are also protected.

The garbage collector does not remove objects that are in these sets, and it also does not remove objects that are referenced by objects in these sets or by persistent object. Garbage Collection does not remove objects for which there is a reference path from an object that cannot itself be garbage collected.

When you execute a GemBuilder function that creates an object, the newly created object is automatically added to the PureExportSet, or in a user action, to the user action's export set, to avoid any risk of it being garbage collected before it can be used.

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 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 export set.
- When the function **GciErr**(GciErrSType *errorReport) returns TRUE, values of type OopType in the *errorReport are added to the 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 set. These are already referenced from the object with the instance variables. However, 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.

GemBuilder does not automatically add other objects to the export sets. If you have temporary objects that should not disappear, the application must be careful to explicitly call **GciSaveObjs** or **GciSaveGlobalObjs** function when it needs to be sure to retain an object that is not already in an export set.

While code that is invoked from a GCI application automatically puts objects in the PureExportSet, code that is executed from within a user action has somewhat different requirements, and use the user action export set, rather than the PureExportSet, to preserve objects from garbage collection. 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 holding on to memory; for example if the user action exits with an unexpected error. User actions can call **GciSaveObjs** to explicitly add objects to their export set, or **GciSaveGlobalObjs** to add specifically add them to the PureExportSet.

Tracked Objects

Another way to preserve objects from garbage collection is to add them to the tracked objects set. Use of this set requires calling **GciTrackedObjsInit** to initialize, and objects are explicitly added by passing them to **GciSaveAndTrackOops**.

Unreferenced persistent object garbage collection

In-memory garbage collection cleans up objects in session memory, and newly created objects or modified objects are put in the export set (or tracked objects set) to ensure that they are not garbage collected before the changes are committed.

When an existing persistent object (that is, an object that has references from other persistent objects, such as a collection) is no longer referenced, they also are subject to garbage collection. markForCollection and epoch garbage collection scan the repository and will dispose of persistent objects that have no reference chain from the repository root.

If the session commits changes that remove a persistent reference to an object, then that object may be at risk of persistent object garbage collection. If the session will be doing further work with that object, it should be added to the exported or tracked objects set, which protect objects from persistent as well as in-memory garbage collection.

Once a persistent object is disposed, its OOP may be reused for another objects. If a session did not add the object to the export set, in a busy application there is a chance that the next GemBuilder call specifying that object will either encounter an object does not exist error, or the object will be an entirely different one than expected.

Releasing Objects

Preserving objects from garbage collection is critical, however, these objects use memory and too many of them will impact performance.

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

Chapter

2

Interacting with GemBuilder

This chapter describes how GemBuilder C functions are used to interact between a client application using a C API and the GemStone server's Smalltalk environment.

2.1 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.

Prepare for Login

Before you can login, you need a running GemStone repository Stone and, usually, a NetLDI process. Setting up the repository and NetLDI, the options for RPC vs. linked logins, and the login parameters to login a Gem session process for either kind of login, is described in the *System Administration Guide for GemStone/S 64 Bit*.

While linked logins may have performance advantages under some circumstances, RPC logins have advantages, and are required when the application is on a different node than the Stone, and RPC logins avoid the risk of invalid memory references causing repository corruption. While performing development and debugging you should log in RPC, to avoid the risk of repository corruption. This discussion will primarily describe the process for RPC logins.

As described in the *System Administration Guide for GemStone/S 64 Bit*, you will need to determine the following:

- Network Resource Syntax (NRS) for the Stone. By default, this is gs64stone.
- NRS for the Gem. This is not set for a linked login, and may be 'gemnetobject' for a local RPC login.
- Authentication information for the Stone (GemStone userId and password).
- Authentication information for the NetLDI (UNIX username and password). This is not required if the NetLDI is running in guest mode.

Logging In and Out

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 GemBuilder functions other than this until it calls **GciInit**.

The **GciSetNet** or **GciSetNetEx** functions set the values for the Stone and Gem NRS, and NetLDI authentication information if required.

Then, to do any useful repository work, you must create a session with the GemStone system by calling **GciLogin**, **GciLoginEx**, or **GciNbLoginEx**; these functions have arguments for providing the GemStone username and password.

The ***Ex** variants support encrypting passwords before passing these from the GCI application to the Gem, which is useful if you are operating over an unsecure network. Encryption is done using the **GciEncrypt** function.

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's classic API, 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. The GemBuilder for C functions operate on the current session, and if you login multiple sessions, 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 committing a series of sequentialized 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. If other sessions have made changes, your attempt to commit may conflict, and the commit will therefore fail. If an attempt to commit fails, your application must call **GciAbort** to discard the transaction.

As mentioned earlier, if your client 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, which is the default, 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 manualBegin. For example:

```
// set aSession in manual transaction mode
GciExecuteStr("System transactionMode: #manualBegin", OOP_NIL);

// set aSession in automatic transaction mode
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 explicitly set the mode to manualBegin.

For a list of functions that are useful for managing sessions, see Table 7.1 on page 81.

2.2 Representing Objects in C

The GemStone data model provides each object with an object-oriented pointer (OOP), a 64-bit integer. This OOP preserves an object's identity distinct from its state.

In your client application, the OOP is represented in variables of type **OopType**. Whenever the client attempts to access or modify the state of a GemStone object, it uses the OOP to identify that object. Both the OOP and a representation of the object's state may be imported into the client application.

The GemBuilder include file `gci.ht` defines type **OopType**, along with other types used by GemBuilder functions. For more information, see Chapter 6, "GemBuilder for C Files and Data Structures", starting on page 65.

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.

The state of the GemStone object—the data in its instance variables—can be imported into the C application in a number of ways, depending on the specific type of data.

GemStone-Defined Object Macros

The GemBuilder include file `gcioop.ht` defines macros for a number of frequently-used objects, such as the GemStone objects *nil*, *true*, and *false*, and kernel classes in the GemStone repository.

In addition, it defines *OOP_ILLEGAL*. This 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.

GemStone Object Implementation Types

Objects in GemStone can be divided up into four different types, based on the internal implementation. These are handled and structure internally differently, and handled differently by some GemBuilder functions. These types are Special, Byte, Pointer, and NSC.

More detail is provided in the *GemStone Programming Guide*.

Special

Special objects are those in which the OOP itself encodes the value. These objects are fast, canonical (each is identical to another of the same value), and do not use repository space. Specials include:

- SmallInteger objects (for example, the number 5)
- SmallDouble objects (for example, the number 0.5)
- SmallFraction objects (for example, the number 1/3)
- Character (for example, the letter ‘b’)
- Boolean values (true and false)
- The instance of class UndefinedObject (nil)

You cannot create new Specials, nor update instances of specials. Conversion between a special and a C data type does not require making a call to the GemStone server.

Byte

Byte classes are collection classes (classes with indexed, unnamed instance variables) that contain objects that use one byte of storage space. The most obvious is String. Strings that require more than one byte of storage or individual characters are byte objects, but two or more bytes are logically required for elements stored in the collection. Byte objects do not have named instance variables.

Pointer

Pointer classes are collection classes that contain objects (such as Array) that have an internal ordering, and ordinary classes with instance variables. Pointer objects use 8 bytes of space per element.

NSC

NSC - Non-sequential collection, instances of the Smalltalk class UnorderedCollection - are stored internally in specialized leaf structures, are unordered, and may also have instance variables.

GemStone Data Types

When moving data between GemStone and your C application, the GemStone object representation must be converted to structures consisting of C data types.

For basic data types such as integers, floating points, and strings, you must convert between the OOP representation in GemStone and the basic C data type.

GemBuilder provides a number of functions that allow you to test for the type of data and perform conversion. These are listed in Table 7.7, "Functions and Macros for Converting Objects and Values" on page 86.

In Example 2.1, assume that you have defined a Smalltalk class called Address that represents a mailing address. If the class has five instance variables, the OOPs of one instance of Address can be imported into a C array called *address*. Assume that the fifth instance variable represents the zip code of the address.

The fifth element of *address* is the OOP of the SmallInteger object that represents the zip code, not the zip code itself. Example 2.1 imports the value of the zip code object to the C variable *zip*.

Example 2.1

```
int64 example_zipcode(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;
}
```

Integers

Integer types in Smalltalk are either SmallIntegers, which have the range -2^{60} to 2^{60} -1, or LargeIntegers. SmallIntegers are specials, but LargeIntegers have OOPs and must be resolved using a GemStone call.

For convenience, `gcioop.ht` defines a number of SmallInteger mnemonics, including OOP_Zero, OOP_One, OOP_Two, and OOP_MinusOne. See `gcioop.ht` for all mnemonics.

Floating Points

Floating point types in Smalltalk are either SmallDouble or Floats. SmallDoubles are specials and can represent C doubles that have value zero or that have exponent bits in range 0x381 to 0x3ff, which corresponds to about 5.0e-39 to 6.0e38; approximately the range of C 4-byte floats. While SmallDouble are specials, Float have OOPs and must be resolved using a GemStone call.

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.

Binary floats other than SmallDouble are represented in GemStone as 8-byte instances of Float or 4-byte instances of SmallFloat. The programmer must supply all the bytes, or provide a C double, to create or store a Float or SmallFloat.

Creating Floats or SmallDoubles using **GciFltToOop** or **Gci_doubleToSmallDouble** and fetched using **GciOopToFlt** avoids byte order problems.

Most GemBuilder functions provide automatic byte swizzling for instances of Float and SmallFloat, or raise an error. **GciFetchBytes_** does not raise an error, but it also does not provide automatic byte swizzling. It is intended primarily for use with other kinds of byte objects, such as strings.

Other Numeric Types

Smalltalk also includes a number of other non-integer numerical types, including SmallFraction and Fraction, DecimalFloats, ScaledDecimals, and others. These require additional effort to represent in C, similar to how application objects are handled.

Character

Character types in Smalltalk are specials, with the full Unicode range. Conversions to C are to int types to allow the full range of Unicode characters.

Boolean

Booleans—true and false—are specials in Smalltalk. GemStone uses BoolType to represent these in C. true and false defined by the macros OOP_TRUE and OOP_FALSE.

Nil/null

The Smalltalk special nil is an instance of the class UndefinedObject, and is equivalent to NULL. This is defined by the macro OOP_NIL.

Strings

GemStone provides a number of classes to support working with collections of characters, particularly to support UTF-8 encoding.

- Traditional strings include String (Characters requiring 8 bits), DoubleByteString (16 bits), and QuadByteString (32 bits).
- Unicode Strings rely on the ICU libraries, and include Unicode7 (7 bits), Unicode16 (16 bits) and Unicode32 (32 bits).
- Traditional and Unicode strings can be encoded to UTF-8 and put into instances of Utf8, a specialized subclass of ByteArray.
- Symbols are canonical CharacterCollections, and include Symbol, DoubleByteSymbol and QuadByteSymbol.

GemBuilder provides the following functions to create a variety of String object from a C string, and to fetch the contents. See **Functions for Creating and Initializing Objects** (page 85) and **Structural Access Functions and Macros** (page 89).

Object Creation

The following GemBuilder functions allow your C program to create one or more new instances of one or more Smalltalk classes:

GciNewOop (page 268)

GciNewOops (page 269)

GciNewOopUsingObjRep (page 271)

Once your application has created a new object, it can export the object to the repository by making sure some committed object in the repository references the new object. For example, add the object to a collection, or set the value of an instance variable in an existing object. This can be done with a call to one of the **GciStore...** functions.

Your C application may also create a new object by executing Smalltalk code, and create the reference from a persistent object in Smalltalk code.

Once the new object is created and the reference is committed, the object has been exported to the repository.

2.3 Performing work in GemStone

GemBuilder provides functions that allow C applications to execute Smalltalk code in the repository, and to send messages directly to GemStone objects.

You can also compile Smalltalk methods using **GciCompileMethod**, **GciClassMethodForClass** or **GciInstMethodForClass**; and the function **GciStoreTravDoTravRefs_** includes a way to perform code execution.

Functions to perform Smalltalk code execution include:

Table 2.1 Summary of functions to execute in Smalltalk

GciExecute* and **GciNbExecute***	Execute Smalltalk code that is contained in a GemStone string object.
GciExecuteStr* and **GciNbExecuteStr***	Execute Smalltalk code that is contained in a C string.
GciPerform* and **GciNbPerform***	Send a message to a GemStone Object

Executing Smalltalk Code in GemStone

Your C application can perform Smalltalk code execution using the GemBuilder functions since as **GciExecute***and **GciExecuteStr***.

The Smalltalk code may be a message expression, a statement, or a series of statements; any self-contained section of code that you could, for example, execute within a Topaz **run** command. The code can be in a GemStone string object and executed using a **GciExecute** function, or a C string and performed using a **GciExecuteStr** function.

These functions expect a SymboList, which is used 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 2.2 demonstrates the use of **GciExecuteStr**.

Example 2.2

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

   OopType result = GciExecuteStr("Globals keys size", OOP_NIL);
   return result;
}
```

Your Smalltalk code has the same format as a method, and may include temporaries. You may include a caret in the smalltalk code to indicate an explicit return, but this is not required. GemStone returns the result of the last Smalltalk statement executed.

There are a number of variants of **GciExecute**, including ones that return specific types of results, and nonblocking variants. See Table 7.4, "Functions for Compiling and Executing Smalltalk Code in the Database" on page 84

Sending Messages to GemStone Objects

To send a message to a GemStone object, use **GciPerform** or one of its variants. **GciPerform** takes a object, a selector, and arguments; when GemStone receives this, it invokes and executes the method associated with that message.

Code execution occurs in the repository server, not in the application.

The following example shows two statements, one using **GsExecuteStr** and the other using **GsPerform**.

Example 2.3

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

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

  // both the following have the same effect

   GciExecuteStr("UserGlobals at: #myNumber put: 55", OOP_NIL);

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

GciPerform is useful when your C code already has GemStone objects that it is manipulating, and when you have specific selectors and arguments to send.

Interrupting GemStone Execution

You can interrupt GemBuilder execution in two different ways:

- **GciSoftBreak** sends a *soft break*, which interrupts the Smalltalk virtual machine only. The GemBuilder functions that can recognize a soft break are **GciPerform**, **GciContinue**, **GciExecute**, **GciExecuteFromContext**, **GciExecuteStr**, and **GciExecuteStrFromContext**.
- **GciHardBreak** sends a *hard break*, which interrupts the Gem process itself, and is not trappable through Smalltalk exception handlers.

Issuing a soft break may be desirable if, for example, your application is executing code in Smalltalk, and 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.

2.4 Structural Access to Objects

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 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-sequence able collection), or special, specified as GC_FORMAT_OOP, GC_FORMAT_BYTE, GC_FORMAT_NSC, and GC_FORMAT_SPECIAL. Data types are described under "GemStone Object Implementation Types" on page 24. 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. **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 (see page 25), 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 (page 101)

GciAppendChars (page 102)

GciFetchByte (page 163)

GciFetchBytes_ (page 164)

GciFetchChars_ (page 166)

GciStoreByte (page 356)

GciStoreBytes (page 357)

GciStoreChars (page 361)

Although instances of Float are implemented within GemStone as byte objects, you should not used the above functions to fetch and store values; use the functions **GciOopToFlt** and **GciFltToOop** to convert between floating point objects and their equivalent C values.

Example 2.4 demonstrates importing a String to a C variable. This code assumes that the C variable *suppId* contains an OOP representing an object of class String, and imports it into *suppName*.

Example 2.4

```
void example_fetchBytes(OopType aGsString)
{
   char supplierName[1025];
   int64 size = GciFetchBytes_(aGsString, 1,
       (ByteType*)supplierName,
      sizeof(supplierName) - 1);
   supplierName[size] = '\0';

   // supplierName now contains the bytes of the GemStone object
   // aGsString, or the first 1024 bytes.
}
```

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 fetch one or more values by the index of the instance variable, using **GciFetchNamedOop*** or **GciFetchOop***, and store them using **GciStoreNamedOop*** or **GciStoreOop***.

Indexable Objects

Any indexable object not implemented as a byte object is represented as an array of OOPs. You can fetch one or more unnamed (indexable) instance variable values using **GciFetchVaryingOop***, and store them using **GciStoreIdxOop***.

If the indexable object also has instance variables, then you can access both named and unnamed (varying or indexable) instance variables using **GciFetchOop*** and **GciStoreOop***. Using these functions, pointers to the named instance variables precede pointers to the indexable instance variables.

For example, assume that the C variable *currSup* contains an OOP representing an object of class Supplier (which defines seven named instance variables). Example 2.5 imports the state of the Supplier object (that is, the OOPs of its component instance variables) into the C variable *instVar*.

Example 2.5

```
void example_fetchOops(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)

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.

To fetch the contents of an NSC, you may use **GciFetchOops**. While this provides an ordered array, GemStone preserves the position of objects in an NSC only until the NSC is modified, or until the session is terminated (whichever comes first).

While you can access named instance variables of NSC objects as you would any other object, and retrieve the contents, you must use special functions to add or remove elements from the collection:

GciAddOopToNsc (page 94)

GciAddOopsToNsc (page 95)

GciRemoveOopFromNsc (page 323)

GciRemoveOopsFromNsc (page 324)

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.

Suppose, for example, that you had created a GemStone Employee class like the one in Example 2.6.

Example 2.6

```
Object subclass: 'Employee'
   instVarNames: #( 'name' 'empNum' 'jobTitle'
                    'department' 'address' 'favoriteTune')
   classVars: #()
   classInstVars: #()
   poolDictionaries: #()
   inDictionary: UserGlobals
```

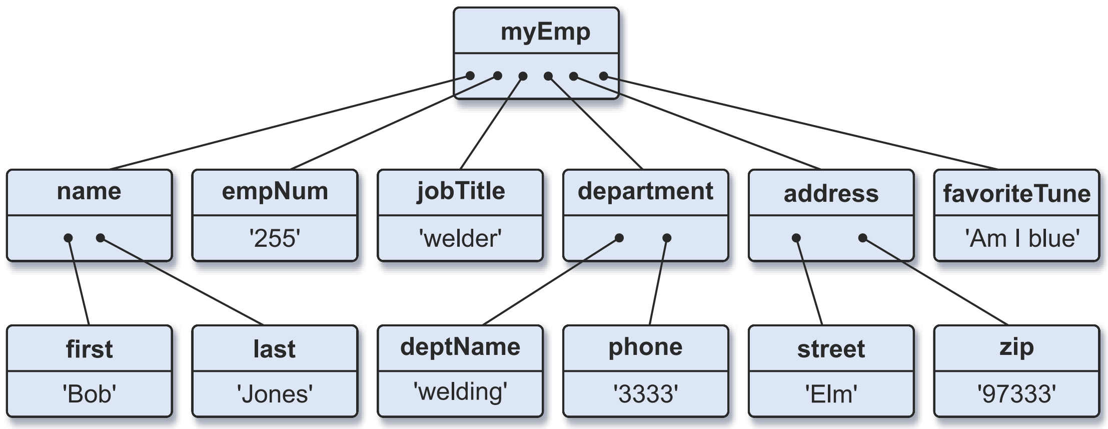
Imagine that you needed to write C code to make a two-column display of job titles and favorite tunes. By using GemBuilder's "object traversal" functions, you can minimize the number of network fetches and avoid running the Smalltalk virtual machine.

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

Figure 2.1 Object Traversal and Paths



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

Thus, to make your list of job titles and favorite tunes with the smallest possible amount of network traffic per employee processed, you could ask GemStone to traverse each employee to two levels (the first level is the Employee object itself and the second level is that object's instance variables). You could then pick out the object reports describing *jobTitle* and *favoriteTune*, and extract the values stored by those reports (*welder* and *Am I Blue* respectively).

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

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

The Object Traversal Functions

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

- *Traversal buffers* are instances of the C++ class GciTravBufType, which is defined in `$GEMSTONE/include/gcicmn.ht`. (For details about GciTravBufType, see "Traversal Buffer - GciTravBufType" on page 78.)
- *Object reports* within the traversal buffer are described by the C++ classes GciObjRepSType and GciObjRepHdrSType, which are defined in `$GEMSTONE/include/gci.ht`. (For details about these classes, see "Object Report Structure - GciObjRepSType" on page 71.)

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.

Functions such as **GciStoreTravDo_** and **GciStoreTravDoTrav_** are 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. These functions takes a structure as an 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 2.1. 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.

2.5 Nonblocking Functions

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

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

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

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

GciCallInProgress
GciErr
GciGetSessionId
GciHardBreak
GciNbEnd
GciSetSessionId
GciShutdown
GciSoftBreak

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

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

Nonblocking functions are not truly nonblocking if they are called from a linkable GemBuilder session, because the Gem and GemBuilder are part of the same process. However, those functions can still be used in linkable sessions. If they are, **GciNbEnd** must still be called at least once per nonblocking call, and it always indicates that the operation is complete.

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

2.6 Operating System Considerations for C Applications

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. *Do not, under any circumstances, turn off SIGIO.*

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.

SIGVTALRM is used by the ProfMonitor class. Similar chaining is required for SIGVTALRM if you intend to use ProfMonitor.

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 any of the **GciPerform*** or **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 if you are running linked or RPC GemBuilder.

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*.

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/gci.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. These macros are recommended:

```
OopType - FMT_OID

int64 - FMT_I64

uint64 - FMT_U64

intptr_t - FMT_PTR_T

uintptr_t - FMT_UPTR_T
```

- 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 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.

2.7 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, run-time (virtual machine), aborting, fatal, and event.

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

Polling for Errors

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

If you are in a UserAction and want to pass the error back to the Smalltalk client, you must execute **GciRaiseException**.

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 GciLnk, relink your application with GciRpc and run it again with an uncorrupted copy of your repository. Your GemStone system administrator can refer to the repository backup and recovery procedures in the *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 GciLnk again.

Chapter 3

Building GCI Applications

This chapter explains how to use GemBuilder for C to build your C application. Your application will provide the main program and API. When linking your compiled code, you will link the GemBuilder for C libraries that allow you to make GemBuilder function calls.

3.1 Environment

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.

There are a number of other GemStone configuration parameters and environment variables that affect your session. In particular, GemStone allows many ways to configure processes to be distributed over multiple hosts, which require specific details in the login parameters. Also, there are a number of configuration parameters that affect your session, such as the amount of memory it can use.

Defaults are provided, so in the most simple case, with all processes running on a single machine, you do not need any further configuration.

Refer to the *System Administration Guide for GemStone/S 64 Bit* for details on both how to use Network Resource String syntax in login parameters, and how to setup configuration files for use by Gems.

3.2 RPC and Linked applications

The GemBuilder for C interface provide two versions: Remote Procedure Call (RPC) and linked.

- In an RPC application, your application exists in a process separate from the Gem. The two processes communicate through remote procedure calls. This interprocess communication requires a small overhead associated with each GemBuilder call, independent of whatever object access is performed or Smalltalk code is executed.
- In a linked application, your application and Gem (the GemStone session) exist as a single process. Your application is expected to provide the main entry point.

  A linked application can also run RPC Gems, in addition to its linked Gem. An RPC application cannot have a linked Gem.

The function **GciIsRemote** reports whether your application was linked with the RPC version or Linked version of the GemBuilder interface.

RPC for Debugging

When debugging a new application, you must use the RPC interface. You should use GciLnk *only* after your application has been completely debugged and thoroughly tested, to avoid the risk of errors corrupting your repository.

When using an RPC Gem, you may achieve better performance by using functions such as **GciTraverseObjs**, **GciStoreTrav**, and **GciFetchPaths**. Those functions are designed to reduce the number of network round-trips by combining functions that are commonly used in sequence.

Linked for Performance

You can use a linked, single-Gem configuration to enhance performance. In a linked application, a GemBuilder function call is a machine-instruction procedure call 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 59.

In a linked application, you may achieve better performance by using the simple **GciFetch**... and **GciStore**... functions instead of the complex object traversal functions. This has the advantage of making the application easier to write, depending on your requirements. However, if your application will login to GemStone multiple times, any sessions after the first will be RPC gems and you should design your application accordingly.

The GemBuilder for C 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`.*, and other required files, are provided in the GemStone distribution under `$GEMSTONE/lib`.

3.3 GemBuilder Applications that load Shared Libraries

Most applications will load shared libraries 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 loads GemBuilder shared libraries:

1. Include `gci.hf` and `gcirtl.hf` in the C source code.

   ```
   #include "gci.hf"
   #include "gcirtl.hf"
   ```

   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** (page 331) to load the RPC GemBuilder.

   To load the RPC version, pass **true** as the first argument; to load the linked version, pass in **false**.

   Call **GciRtlLoad** before any other GemBuilder calls. To unload the GemBuilder libraries, call **GciRtlUnload**. The **GciRtlIsLoaded** function can be used to determine if an interface is loaded or not.

3. When linking, include `gcirtlobj.o,` not one of the GemBuilder libraries `(libgcirpc.*` and `libgcilnk.*).`

   Chapter 5, "Compiling and Linking" tells how to compile and link your application, and provides specific details for what should be included in the compile and link operations.

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 in the argument to **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 the environment variable LD_LIBRARY_PATH.
2. Any path recorded in the executable when it was built.
3. The global directory `/usr/lib`.

3.4 Building Statically Linked Applications

Normally, applications will load shared libraries at runtime. However, it is also possible to statically link GemBuilder applications, which in some cases may improve performance.

In a statically linked application, the executable is linked to the GemBuilder library archive file when the executable is built. When the executable runs, it automatically loads the library. Unlike when loading a shared library, your code cannot programmatically determine which library to load. For example, with run-time loading, your code can determine which version of the GemBuilder libraries to load, but this must be determined ahead of time for a statically linked application.

Static linking is only possible with the RPC shared libraries.

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

1. Include `gci.hf` in the C source code.
2. When linking, include the RPC GemBuilder library (`libgcirpc.*`).

GemBuilder finds build-time-bound shared libraries by using the normal operating system searches described in “Searching for the Library” on page 43. If a library cannot be found, the operating system returns an error.

Chapter

4

Writing User Actions

Within a GemStone Smalltalk-based application, you may choose to write a C function for certain operations, rather than to perform the work in GemStone. For example, operations that are computationally intensive 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”.

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.

Some of the functionality provided by User Action libraries can instead be supported using the Foreign Function Interface (FFI) in GemStone Smalltalk, which allows you to invoke functions in C libraries directly from Smalltalk, without compiling C code. The FFI is described in *Programming Guide*.

4.1 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.

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 Gem session process. The libraries themselves are the same for both, it is the operation that is used to load the library determines which type it is.

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.

4.2 How User Actions Work

User Actions are invoked from GemStone Smalltalk. This can be done from GemStone Smalltalk application code, topaz execution, etc, or using GemBuilder for C by calling a function such as **GciExecuteStr** or **GciPerform**.

Before it can be used, the user action library must be loaded and initialized; see “Loading User Actions” on page 50.

1. GemStone Smalltalk code invokes a user action function (written in C) by sending a message of the form:

   `System userAction:` *aSymbol* `with:` *args*

   The *args* arguments are passed to the C user action function named *aSymbol*.

2. 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).

   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 execute the code.

3. 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.*

4.3 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, and 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.

1. Write the User Action Functions

Writing a C function to install as a user action called from Smalltalk is much the same as writing other C functions. However, since 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—user actions do not solely control the way objects are preserved from garbage collection.

For more on the issue of retaining references to newly created objects and risk of premature garbage collection, see the discussion under“Garbage Collection” on page 18.

To enable objects to be retained, user actions maintain a separate export set that is specific to the user action, in addition to the PureExportSet. When objects are automatically added to an export set or added using **GciSaveObjs**, they are put into the user action's export set rather than into the PureExportSet.

When the user action function completes, its export set is cleared. Don't save the OOPs in static C variables for use by a subsequent invocation of the user action or by another C function.

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 persistent collection that the application looked up in the GemStone repository, or that was provided as an argument to the user action function.
- 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.

To create a reference that prevents garbage collection but is not shared between sessions, you may use GemStone Smalltalk code to add them to the user-definable portion of `System sessionState` using `System >> sessionStateAt:put:`. Objects referenced in this way will be preserved until the session terminates, at which point they will be subject to garbage collection if there are no other references.

2. 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 the appropriate options, as described in Chapter 5.
4. Link with `gciualib.o` and the appropriate options, as described in Chapter 5.
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 4.1 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 4.1

```
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 4.2 shows a shutdown definition that does nothing but report that it has been called.

Example 4.2

```
#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 5, “Compiling and Linking” for instructions.

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

Using Existing User Actions in a User Action Library

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

Using Third-party C Code with a User Action Library

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

3. 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
- 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; it is recommended to use the base name. 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. For example, the base name `custreport` would be expanded to `libcustreport.so` on many UNIX platforms.

The code for loading searches for the file in the following places in the specified order:

- The current directory of the application or Gem.
- The directory the executable is in, if it can be determined.
- The `$GEMSTONE/ualib` directory.
- The normal operating system search, as described in “Searching for the Library” on page 43.

Creating User Actions in Your C Application

Loading user action libraries at run time is the preferred behavior for GemBuilder applications. For application user actions, however, you have the option to create the user actions directly in your C application, not as part of a library. When you implement user actions this way, include `gcirtl.hf` or `gci.hf` in your C source code, instead of `gciua.hf`. (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:

```
System hasUserAction: aSymbol
```

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

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

```
System userActionReport
```

4. 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.
```

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; no more than 2 user actions may be active at any one time on the current Smalltalk stack. If the limit is exceeded, GemStone raises an error.

5. 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 59.

Use the instructions for user actions in Chapter 5, "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.

4.4 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.

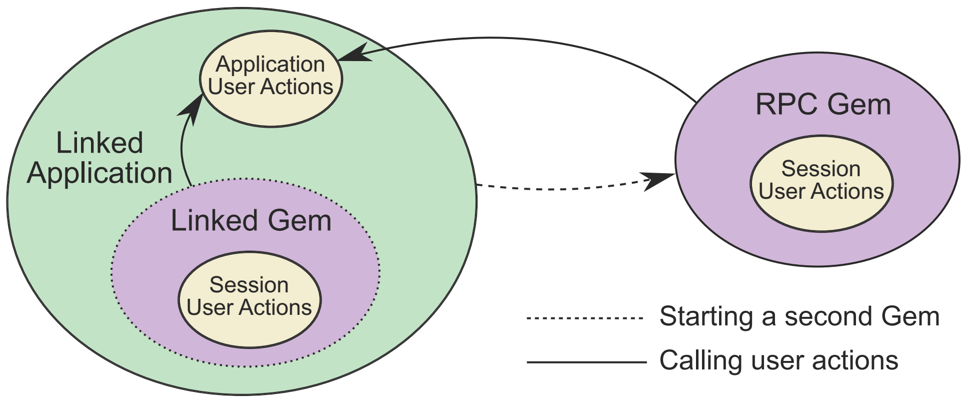
Applications With Multiple Gems

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 4.1. 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 4.1 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

In an RPC 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

In a linked 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. 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 59.*

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.

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 59.*

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.

Chapter 5 Compiling and Linking

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 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.

5.1 Development Environment and Standard Libraries

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 example command lines.

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.

5.2 Compiling and Linking 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.

Compiling

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. This chapter assumes that a compatible compiler has been installed and is in your path.

The C++ compilers listed in section 5.3 were used to produce the GemStone product, and have been tested for producing C/C++ applications and user action libraries. Other compilers, such as ANSI C++ compilers, are assumed to work, but have not been tested.

Compilation Options

Compiling user actions and GCI applications use the same 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 must specify the GemStone `include` directory, but it is likely you will have other include file directories.

For simplicity in compiling code for user actions, the .c 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.

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

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

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

Linking

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

Link options

Linking user actions and GCI applications require different link options.

There is no difference in the link options for applications that will login linked or RPC. The 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.

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 documentation.

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

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.

5.3 Platform Specific versions and example options

The compile command lines for each platform illustrate 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.

The link command lines illustrate

- how to link a user action object file named *userCode*.o with GemStone libraries to produce a user action library named lib*userAct*.so or lib*userAct*.dylib, depending on your platform.
- how to link a simple application program with one application object file, *userCode*.o. Its result is one executable file, *userAppl*.

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

Linux Compile and Link Information

Complier version

Red Hat 6.x: gcc/g++ 4.4.7

Red Hat 7.x: gcc/g++ 4.8.5

Ubuntu Linux 16.04: gcc/g++ 5.4.0

Ubuntu Linux 18.04: gcc/g++ 7.3.0

SUSE Linux 12: gcc/g++ 4.8.5

Listing the Compiler Version

```
% g++ -v
```

Debugger version

Red Hat 6.x: GNU gdb 7.2-92.el6

Red Hat 7.1: GNU gdb 8.2.1

Ubuntu Linux 16.04: GNU gdb 7.11.1

Ubuntu Linux 18.04: GNU gdb 8.1.0.20180409-git

SUSE Linux 12: GNU gdb 7.9.1

Compiling a user action or GCI application

```
g++ -fmessage-length=0 -fcheck-new -O3 -ggdb -m64 -pipe
   -D_REENTRANT -D_GNU_SOURCE -pthread -fPIC -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 -Wtrigraphs
   -Wno-aggregate-return -Wswitch -Wshadow -Wunused-value
   -Wunused-variable -Wunused-label -Wno-unused-function
   -Wchar-subscripts -Wmissing-braces -Wmissing-declarations
   -Wmultichar -Wparentheses -Wsign-compare -Wsign-promo
   -Wwrite-strings -Wreturn-type -Wuninitialized
```

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.

Linking a user action library

```
g++ -shared -Wl,-Bdynamic,-hlibuserAct.so userCode.o
   $GEMSTONE/lib/gciualib.o -o libuserAct.so -m64
   -Wl,--no-as-needed -lpthread -Wl,--as-needed -lcrypt -ldl -lc
   -lm -lrt
```

Linking a GCI application

```
g++ userCode.o $GEMSTONE/lib/gcirtlobj.o -Wl,-traditional
   -Wl,-z,lazy -m64 -Wl,--no-as-needed -lpthread -Wl,--as-needed
   -lcrypt -ldl -lc -lm -lrt -o userAppl
```

Solaris on x86 Compile and Link Information

Complier version

CC: Studio 12.5 Sun C++ 5.14 SunOS_i386 2016/05/31

Listing the Compiler Version

```
% CC -V
```

Debugger version

Sun DBX Debugger 7.7 SunOS_i386 2009/06/03

Compiling a user action or GCI application

```
CC -xO4 -m64 -xarch=generic -Kpic -mt -D_REENTRANT
   -D_POSIX_PTHREAD_SEMANTICS -I$GEMSTONE/include
   -features=no%except -c userCode.c -o userCode.o
```

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

Linking a user action library

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

Linking a GCI application

```
CC -xildoff -m64 -xarch=generic -i userCode.o
   $GEMSTONE/lib/gcirtlobj.o -z nodefs -Bdynamic -lc -lpthread
   -lsendfile -ldl -lrt -lsocket -lnsl -lm -lpam -lCrun -o userAppl
```

AIX Compile and Link Information

Complier version

AIX 6.1, and 7.1 on POWER7: IBM XL C/C++ for AIX, V11.1

AIX 7.1 on POWER8: IBM XL C/C++ for AIX, V13.1.2

Listing the Compiler Version

```
% /usr/vacpp/bin/xlC_r -qversion
```

Debugger version

dbx

Compiling a user action or GCI application

```
xlC_r -O3 -qstrict -qalias=noansi -q64 -+ -qpic
   -qthreaded -qarch=pwr6 -qtune=balanced -D_LARGEFILE64_SOURCE
   -DFLG_AIX_VERSION=version -D_REENTRANT -D_THREAD_SAFE
   -qminimaltoc -qlist=offset -qmaxmem=-1 -qsuppress=1500-010:1500
   -029:1540-1103:1540-2907:1540-0804:1540-1281:1540-1090 -qnoeh
   -I$GEMSTONE/include -c userCode.c -o userCode.o
```

Depending on your version of AIX, you need to include -DFLG_AIX_VERSION=61, -DFLG_AIX_VERSION=71, or -DFLG_AIX_VERSION=72.

Also 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`.

Linking a user action library

```
xlC_r -G -Wl,-bdatapsize:64K -Wl,-btextpsize:64K
   -Wl,-bstackpsize:64K -q64 userCode.o $GEMSTONE/lib/gciualib.o
   -o libuserAct.so -e GciUserActionLibraryMain -L/usr/vacpp/lib
   -lpthreads -lc_r -lC_r -lm -ldl -lbsd -lpam -Wl,-berok
```

Linking a GCI application

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

DARWIN Compile and Link Information

Complier version

Apple LLVM version 6.0 (clang-600.0.56)

Listing the Compiler Version

```
% g++ -v
```

Debugger version

lldb-1000.0.38.2, Swift-4.2

Compiling a user action or GCI application

```
g++ -fmessage-length=0 -O3 -ggdb -m64 -pipe -fPIC
   -fno-strict-aliasing -D_LARGEFILE64_SOURCE -D_XOPEN_SOURCE
   -D_REENTRANT -D_GNU_SOURCE -I$GEMSTONE/include -x c++
   -c userCode.c -o userCode.o
```

The following warn flags are recommended for compilation:

```
   -Wno-format -Wtrigraphs -Wcomment -Wsystem-headers -Wtrigraphs
   -Wno-aggregate-return -Wswitch -Wshadow -Wunused-value
   -Wunused-variable -Wunused-label -Wno-unused-function
   -Wchar-subscripts -Wno-conversion -Wmissing-braces -Wmultichar
   -Wparentheses -Wsign-compare -Wsign-promo -Wwrite-strings
   -Wreturn-type -Wno-nullability-completeness
   -Wno-expansion-to-defined
```

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

Linking a user action library

```
g++ -dynamiclib userCode.o $GEMSTONE/lib/gciualib.o
   -o libuserAct.dylib -m64 -lpthread -ldl -lc -lm -lpam -undefined
   dynamic_lookup
```

Linking a GCI application

```
g++ userCode.o $GEMSTONE/lib/gcirtlobj.o -undefined dynamic_lookup
   -m64 -lpthread -ldl -lc -lm -lpam -o userAppl
```

Windows Compile and Link Information

GemStone/S 64 Bit supports Windows only as a client platform. You may compile and link GCI applications, but not user actions, on Windows.

Complier/Debugger version

Microsoft Visual Studio 2010 Version 10.0.30319.1 RTMRel

Microsoft Visual C++ 2010 01021-532-2002102-70611

Compiling a GCI application

```
cl /W3 /Zi /MD /O2 /Oy- -DNDEBUG /TP /nologo /D_LP64 /D_AMD64_
   /D_CONSOLE /D_DLL /DWIN32_LEAN_AND_MEAN /
   D_CRT_SECURE_NO_WARNINGS /EHsc
   /DNATIVE /I 'VisualStudioInstallPath\atlmfc\include'
   /I 'VisualStudioInstallPath\VC\include'
   /I 'C:\Program Files (x86)\Microsoft
   SDKs\Windows\v7.0A\Include'
   /I '%GEMSTONE%\include' -c userCode.c -FouserCode.obj
```

Linking a GCI application

```
link /LIBPATH:"VisualStudioInstallPath\VC\lib\amd64"
   /LIBPATH:"C:\Program Files (x86)\Microsoft
   SDKs\Windows\v7.0A\Lib\x64" -RELEASE
   /OPT:REF /INCREMENTAL:NO /MAP /nologo /MANIFEST
   /MANIFESTFILE:userAppl.exe.manifest
   /MANIFESTUAC:"level='asInvoker'" userCode.obj
   %GEMSTONE%\lib\gcirpc.lib ws2_32.lib netapi32.lib advapi32.lib
   comdlg32.lib user32.lib gdi32.lib kernel32.lib winspool.lib
   Secur32.lib /out:userAppl.exe
```

Chapter 6

GemBuilder for C Files and Data Structures

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

6.1 GemBuilder for C Include Files

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

Your C source code should include one of these include files:

`gci.hf`	Forward references to the GemBuilder functions. Use when creating GCI applications.
`gciua.hf`	Forward references to the GemBuilder functions that includes definitions to support user actions. Include this when creating user actions.

Using the GemStone C Statistics Interface (GCSI), as described in Chapter 8, “GemStone C Statistics Interface”, starting on page 405, include the following:

`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 64 Bit.)
`gcsi.hf`	Prototypes for all GCSI functions.
`gcsierr.ht`	GCSI error numbers.

The following include files are included by `gciua.hf` and/or `gci.hf`; you do not need to explicitly include any other files in order to use the GCI functions:

`gcirtl.hf`	Forward references to the GemBuilder functions that perform run time loading.
`flag.ht`	Contains host-specific C definitions for compilation.
`gcicmn.ht`	Defines common C types and macros.
`gcierr.ht`	Defines mnemonics for all GemStone errors.
`gcifloat.ht`	Macros, constants and functions for accessing the bodies of instances of GemStone classes Float and SmallFloat.
`gci.ht`	Defines C types for use by GemBuilder functions.
`gcilegacy.ht`	Definitions that are used by gci.hf; they are in a distinct file since they are not used by the thread-safe GCI.
`gcioc.ht`	Defines C mnemonics for sizes and offsets into objects.
`gcioop.ht`	Defines C mnemonics for predefined GemStone objects.
`gciuser.ht`	Defines a macro to be used to install user actions; loaded by `gciua.hf` , not by `gci.hf` .
`version.ht`	Defines C mnemonics for version-dependent strings.

6.2 Reserved Oops

The GemStone/S 64 Bit distribution includes the file `gcioop.ht`, which defines C macros that define the OOPs of certain commonly used GemStone objects. Your code can compare these with any value of type `OopType`.

Note that the actual definition of these macros is subject to change without notice in future software releases. Your C application should refer to the OOPs of predefined GemStone objects *by defined macro name only*.

The following are some of the names for predefined GemStone objects Refer to `$GEMSTONE/include/gcioop.ht` for a complete list.

- 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 to test whether or not an OOP is valid.

  OOP_ILLEGAL

- Boolean and nil objects

  OOP_NIL (*nil*)
  OOP_FALSE (*FALSE*)
  OOP_TRUE (*true*)

- Instances of SmallInteger

  OOP_MinusOne
  OOP_MinusTwo
  OOP_Zero
  OOP_One
  OOP_Two
  OOP_Three
  OOP_Four
  MAX_SMALL_INT
  MIN_SMALL_INT

- Instances of Character

  OOP_ASCII_NUL represents the first ASCII character OOP. Other Characters have specific OOPs, but no macros.

- The GemStone Smalltalk kernel classes

  OOP_CLASS_*className*, where *className* is the name of a class (which may be uppercase or camelcase) Refer to `gcioop.ht` for specific names.

  OOP_LAST_KERNEL_OOP (which has the same value as the last class)

6.3 GemBuilder Data Types

The following C types/C++ classes are used by GemBuilder functions. The file `$GEMSTONE/include/gci.ht` defines each of the GemBuilder types.

BoolType	An integer.
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	A GemStone error report; see “Error Report Structure - GciErrSType” on page 69.
GciJumpBufSType	Error jump buffer.
GciObjInfoSType	A C++ class for a GemStone object information report; see “Object Information Structure - GciObjInfoSType” on page 70.
GciObjRepHdrSType	A C++ class for an object report header; see “Object Report Header - GciObjRepHdrSType” on page 71.

GciObjRepSType	A C++ class for an object report; see “Object Report Structure - GciObjRepSType” on page 71.
GciSessionIdType	A signed 32-bit integer.
GciStoreTravDoArgsSType	A C++ class for store traversal arguments.
GciTravBufType	A traversal buffer; see “Traversal Buffer - GciTravBufType” on page 78.
GciUserActionSType	A structure for describing a user action (see “User Action Information Structure - GciUserActionSType” on page 79.

Date/Time Structure- GciDateTimeSType

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

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

```
typedef struct {
      int year;
      int dayOfYear;
      int milliseconds;
      OopType timeZone;
} GciDateTimeSType;
```

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

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

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

Error Report Structure - GciErrSType

An error report is a C++ class named GciErrSType. This includes 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 Utf8 containing the error message. In this release, GCI_ERR_STR_SIZE is defined to be 1024.
char	*reason*[GCI_ERR_reasonSize + 1]	The null-terminated Utf8 reason. GCI_ERR_reasonSize is defined to be the same as GCI_ERR_STR_SIZE.

The arguments (*args*) are specific to the error encountered. See the `gcierr.ht` file for a full list of errors and their arguments.

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).

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.

Object Information Structure - GciObjInfoSType

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 181).

OopType	*objId*	OOP of the object.
OopType	*objClass*	Class of the object.
int64	*objSize*	Object's total size in bytes or OOPs; see **GciFetchSize_** (page 190).
int	*namedSize*	Number of named instance variables in the object.
unsigned short	*objectSecurityPolicyId*	The ID of the object’s security policy.
unsigned short	*_bits*	Bits indicating the implementation format and mask. See gci.hf for the bit definitions.

Functions

The object information class GciObjInfoSType provides the following functions:

`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. The mnemonics for these values are:

GC_FORMAT_OOP
GC_FORMAT_BYTE
GC_FORMAT_NSC
GC_FORMAT_SPECIAL

For bit functions, see gci.hf.

Object Report Structure - GciObjRepSType

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):

```
class GciObjRepSType {
 public:
  GciObjRepHdrSType hdr;         /* object report header */
  union {
    ByteType        bytes[1];   /* Byte objects   */
    OopType         oops[1];    /* Pointer objects */
  } u;
```

Functions

The object report class **GciObjRepSType** provides these functions:

`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.

Object Report Header - GciObjRepHdrSType

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.
int64	*firstOffset*	Offset of first value to fetch or store; 1 means first instVar is the first named instVar.
unit64	*_idxSizeBits*	40 bits size, 8 bits dynamic inst var size, 6 bits unused, 2 bits swizzle, 8 bits implementation format and mask. See gcicmn.hf for the bit definitions.

Functions

The object report header class GciObjRepHdrSType provides the following functions:

`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 73.

`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. You may use the mnemonics GC_FORMAT_OOP, GC_FORMAT_BYTE, GC_FORMAT_NSC, and GC_FORMAT_SPECIAL.

`int64 objSize( );`
Returns the total number of instance variables in the object (both indexable and named).

`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 isOverlayed( );`
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 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 end of the report so that the next report in the buffer begins on an address that is a multiple of 8.

`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 `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.

Object implementation

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.

Table 6.1 Object Implementation Restrictions on Instance Variables

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

For more information about object implementation types, see “Structural Access to Objects” on page 30.

Store Traversal Arguments - GciStoreTravDoArgsSType

GciStoreTravDoArgsSType holds traversal arguments, and is described in `$GEMSTONE/include/gcicmn.ht`. It contains the following fields:

GciTravBufType*	*storeTravBuff*	The traversal buffer. For details, see “Traversal Buffer - GciTravBufType” on page 78.
int	*storeTravFlags*	Flag bits that determines how the objects should be handled. GCI_STORE_TRAV_DEFAULT = 0 Default behavior: use "add" semantics for varying instvars of Nsc's; if object to be stored into does not exist, give error. GCI_STORE_TRAV_NSC_REP = 0x1, Use REPLACE semantics for varying instvars of NSCs. GCI_STORE_TRAV_CREATE = 0x2 If an object to be stored into does not exist, create the new object and add it to the PureExportSet . GCI_STORE_TRAV_FINISH_UPDATES = 0x8 After processing the last object report in the traversal buffer, execute GciProcessDeferredUpdates. GCI_STORE_TRAV_SWIZZLE_IN_PLACE = 0x10 if swizzling by the client is being done for transmit to a server with opposite byte order, this flag says it is ok to swizzle the caller's buffer in place before transmitting. Reduces client memory access footprint of a large store traversal
int	*doPerform*	‣ If this field is 0, this function executes a string using *args->u.executestr*, with the semantics of “GciExecute” on page 151. ‣ If this field is 1, then executes a perform using *args->u.perform*, with the semantics of “GciPerform” on page 296. ‣ 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 123.

int	*doFlags*	Flags to disable or permit asynchronous events and debugging in Smalltalk. These flags apply whatever the value of *doPerform*. ‣ 0 (default) disables the debugger during execution. ‣ GCI_PERFORM_FLAG_ENABLE_DEBUG = 1 allows debugging, making this function behave like GciPerform. ‣ GCI_PERFORM_FLAG_DISABLE_ASYNC_EVENTS = 2 disables asynchronous events. ‣ GCI_PERFORM_FLAG_SINGLE_STEP = 3 places a single-step breakpoint at the start of the method to be performed, and then executes to hit that breakpoint.
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 ‣ *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*	An array allocating memory for OOPs of objects that will be modified as a consequence of executing the specified code.
int	*alteredNumOops*	The number of OOPs in *alteredTheOops*. On input, the caller must set this to the maximum number of OOPs that will fit in *alteredTheOops*. Upon completion, this field indicates the number of OOPs actually written to *alteredTheOops*.
BoolType	*alteredCompleted*	Upon output, TRUE if the *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.

perform has the following fields:

`OopType`	*receiver*	
`char`	*pad[24]*	Make later elements same offset as executestr
`const char*`	*selector*	1 byte per character
`const OopType*`	*args*	

`int`	*numArgs*	
ushort	*environmentId*	

executestr has the following fields:

OopType	*contextObject*	
OopType	*sourceClass*	String, Utf8 or Unicode7 or DoubleByteString
OopType	*symbolList*	
int64	*sourceSize*	
const char*	*source*	1 or 2 bytes per char, client-native byte order
const OopType*	*args*	ignored unless ExecuteBlock
int	*numArgs*	ignored unless ExecuteBlock
ushort	*environmentId*	

continueArgs has the following fields:

OopType	*process*	
OopType	*replaceTopOfStack*	

Clamped Traversal Arguments - GciClampedTravArgsSType

GciClampedTravArgsSType holds the arguments for clamped traversal, and includes the following 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*	The value of retrievalFlags should be given by using the following mnemonics: GCI_RETRIEVE_DEFAULT and GCI_RETRIEVE_EXPORT Causes results to automatically be added to an export set using GciSaveObjs semantics. GCI_CLEAR_EXPORT Causes traverse to clear the export set using GciReleaseAllOops before filling traversal buffer. GCI_COMPRESS_TRAV Compress traversal buffers before transmissions between gem and rpc gci client, including the store traversal of this call, and any GciMoreTraversal calls used to finish this call. GCI_NOKILL_TRAV Start a traversal that will allow other GCI calls to preceed GciMoreTraversal calls that are part of this traversal. Only the start of another traversal will kill this traversal. GCI_TRAV_REFS_EXCLUDE_RESULT_OBJ Exclude execution result * from the traversal result. GCI_TRAV_REFS_EXCLUDE_EXPORTED_RESULT_OBJ Only give report for execution result if not already exported. Has no effect if GCI_TRAV_REFS_EXCLUDE_RESULT_OBJ is set..

Traversal Buffer - GciTravBufType

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

uint	*allocatedBytes*	The allocated size of the body.
uint	*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. Maximum size is 4GB, or, if compression is used, 2GB.

To create a new instance of GciTravBufType, use the function **GciAllocTravBuf** (page 98).

Functions

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.

User Action Information Structure - GciUserActionSType

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*	For internal use.

Chapter
7

GemBuilder C Function Reference

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

7.1 Function Summary Tables

Tables 7.1 through 7.10 summarize the GemBuilder C functions and the services that they provide to your application.

Table 7.1 Functions for Controlling Sessions and Transactions

GciAbort, GciNbAbort	Abort the current transaction.
GciBegin, GciNbBegin	Begin a new transaction.
GciCommit, GciNbCommit	Write the current transaction to the database.
GciDeclareAction	An alternative way to associate a C function with a Smalltalk user action.
GciEncrypt	Encrypt a password string.
GciGetSessionId	Find the ID number of the current user session.
GciInit	Initialize GemBuilder.
GciInitAppName	Override the default application configuration file name.
GciInstallUserAction	Associate a C function with a Smalltalk user action.
GciIsRemote	Determine whether the application is running linked or remotely.
GciLoadUserActionLibrary	Load an application user action library.

Table 7.1 Functions for Controlling Sessions and Transactions (Continued)

GciLogin, GciLoginEx, GciNbLoginEx	Start a user session.
GciLogout	End the current user session.
GciNbEnd, GciNbEndPoll	Test the status of nonblocking call in progress for completion.
GciProcessDeferredUpdates_	Process deferred updates to objects that do not allow direct structural update.
GciRtlIsLoaded	Report whether a GemBuilder library is loaded.
GciRtlLoad	Load a GemBuilder library.
GciRtlUnload	Unload a GemBuilder library.
GciServerIsBigEndian	Determine whether or not the server process is big-endian.
GciSessionIsRemote	Determine whether or not the current session is using a Gem on another machine.
GciSetCacheName_	Set the name that a linked application will be known by in the shared cache.
GciSetNet, GciSetNetEx	Set network parameters for connecting the user to the Gem and Stone processes.
GciSetSessionId	Set an active session to be the current one.
GciShutdown	Logout from all sessions and deactivate GemBuilder.
GciUserActionInit	Declare user actions for GemStone.
GciUserActionShutdown	Enable user-defined clean-up for user actions.

Table 7.2 Functions for Handling Errors and Interrupts and for Debugging

GciCallInProgress	Determine if a GemBuilder call is currently in progress.
GciClearStack	Clear the Smalltalk call stack.
GciContinue, GciNbContinue	Continue code execution in GemStone after an error.
GciContinueWith GciNbContinueWith	Continue code execution in GemStone after an error.
GciDbgEstablish	Specify the debugging function for GemBuilder to execute before most calls to GemBuilder functions.
GciDbgEstablishToFile	Write trace information for most GemBuilder functions to a file.
GciDbgLogString	Pass a message to a trace function.
GciEnableSignaledErrors	Establish or remove GemBuilder visibility to signaled errors from GemStone.

Table 7.2 Functions for Handling Errors and Interrupts and for Debugging

GciErr	Prepare a report describing the most recent GemBuilder error.
GciInUserAction	Determine whether or not the current process is executing a user action.
GciHardBreak	Interrupt GemStone and abort the current transaction.
GciLongJmp	Provides equivalent functionality to the corresponding longjmp() or _longjmp() function.
GciPollForSignal	Poll GemStone for signal errors without executing any Smalltalk methods.
GciPopErrJump	Discard a previously saved error jump buffer.
GciPushErrJump	Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.
GciRaiseException	Signal an error, synchronously, within a user action.
GciSetErrJump	Enable or disable the current error handler.
GciSetHaltOnError	Halt the current session when a specified error occurs.
Gci_SETJMP	(MACRO) Save a jump buffer in GemBuilder's error jump stack.
GciSoftBreak	Interrupt the execution of Smalltalk code, but permit it to be restarted.
GciStep	Continue code execution in GemStone with specified single-step semantics.

Table 7.3 Functions for Managing Object Bitmaps

GciAlteredObjs	Find all exported or dirty objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.
GciDirtyExportedObjs	Find all objects in the ExportedDirtyObjs set.
GciDirtyObjsInit	Begin tracking which objects in the session workspace change.
GciDirtySaveObjs	Find all exported or tracked objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.
GciDirtyTrackedObjs	Find all tracked objects that have changed and are therefore in the TrackedDirtyObjs set.
GciHiddenSetIncludesOop	Determines whether the given OOP is present in the specified hidden set.
GciReleaseAllGlobalOops	Remove all OOPS from the PureExportSet, making these objects eligible for garbage collection.
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.

Table 7.3 Functions for Managing Object Bitmaps (Continued)

GciReleaseAllTrackedOop s	Clear the GciTrackedObjs set, making all tracked OOPs eligible for garbage collection.
GciReleaseGlobalOops	Remove an array of GemStone OOPs from the PureExportSet, making them eligible for garbage collection.
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.
GciReleaseTrackedOops	Remove an array of OOPs from the GciTrackedObjs set, making them eligible for garbage collection.
GciSaveAndTrackObjs	Add objects to GemStone's internal GciTrackedObjs set to prevent them from being garbage collected.
GciSaveGlobalObjs	Add an array of OOPs to the PureExportSet, making them ineligible for garbage collection.
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.
GciTrackedObjsFetchAll-Dirty	Find all exported or tracked objects that have changed and are therefore in the ExportedDirtyObjs or TrackedDirtyObjs sets.
GciTrackedObjsInit	Reinitialize the set of tracked objects maintained by GemStone.

Table 7.4 Functions for Compiling and Executing Smalltalk Code in the Database

GciClassMethodForClass	Compile a class method for a class.
GciClassRemoveAll-Methods	Remove all methods from the method dictionary of a class.
GciCompileMethod	Compile a method.
GciExecute, GciNbExecute	Execute a Smalltalk expression contained in a GemStone String or Utf8 object.
GciExecuteFromContext	Execute a Smalltalk expression contained in a String or Utf8 object as if it were a message sent to another object.
GciExecuteStr, GciNbExecuteStr	Execute a Smalltalk expression contained in a C string.
GciExecuteStrFetchBytes, GciNbExecuteStrFetch-Bytes	Execute a Smalltalk expression contained in a C string, returning byte-format results.
GciExecuteStrFromCon-text, GciNbExecuteStrFromCon-text	Execute a Smalltalk expression contained in a C string as if it were a message sent to an object.

Table 7.4 Functions for Compiling and Executing Smalltalk Code in the Database

GciInstMethodForClass	Compile an instance method for a class.
GciPerform, GciNbPerform	Send a message to a GemStone object.
GciPerformNoDebug, GciNbPerformNoDebug	Send a message to a GemStone object, and temporarily disable debugging.
GciPerformSymDbg	Send a message to a GemStone object, using a String object as a selector.
GciPerformTraverse	First send a message to a GemStone object, then traverse the result of the message.

Table 7.5 Functions for Accessing Symbol Dictionaries

GciResolveSymbol	Find the OOP of the object to which a symbol name C string refers, in the context of the current session's user profile.
GciResolveSymbolObj	Find the OOP of the object to which a symbol name String or MultiByteString refers, in the context of the current session's user profile.
GciStrKeyValueDictAt	Find the value in a symbol KeyValue dictionary at the corresponding string key.
GciStrKeyValueDictAt Obj	Find the value in a symbol KeyValue dictionary at the corresponding object key.
GciStrKeyValueDictAt ObjPut	Store a value into a symbol KeyValue dictionary at the corresponding object key.
GciStrKeyValueDictAtP ut	Store a value into a symbol KeyValue dictionary at the corresponding string key.
GciSymDictAt	Find the value in a symbol dictionary at the corresponding string key.
GciSymDictAtObj	Find the value in a symbol dictionary corresponding to the key object.
GciSymDictAtObjPut	Store a value into a symbol dictionary at the corresponding object key.
GciSymDictAtPut	Store a value into a symbol dictionary at the corresponding string key.
GciTraverseObjs	Traverse an array of GemStone objects.

Table 7.6 Functions for Creating and Initializing Objects

GciCreateByteObj	Create a new byte-format object.
GciCreateOopObj	Create a new pointer-format object.
GciGetFreeOop	Allocate an OOP.
GciGetFreeOops	Allocate multiple OOPs.

Table 7.6 Functions for Creating and Initializing Objects (Continued)

GciGetFreeOopsEncoded	Allocate multiple OOPs.
GciNewByteObj	Create and initialize a new byte object.
GciNewCharObj	Create and initialize a new character object.
GciNewDateTime	Create and initialize a new date-time object.
GciNewOop	Create a new GemStone object.
GciNewOops	Create multiple new GemStone objects.
GciNewOopUsingObjRep	Create a new GemStone object from an existing object report.
GciNewString	Create a new String object from a C character string.
GciNewSymbol	Create a new Symbol object from a C character string.
GciNewUtf8String	Create a new Unicode string object from a UTF-8 encoded C character string.

Table 7.7 Functions and Macros for Converting Objects and 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 32 bit C character value to a GemStone Character object.
GciCTimeToDateTime	Convert a C date-time representation to the equivalent GemStone representation.
GciDateTimeToCTime	Convert a GemStone date-time representation to the equivalent C representation.
Gci_doubleToSmallDouble	Convert a C double to a SmallDouble object.
GciFetchDateTime	Convert the contents of a DateTime object and place the results in a C structure.
GciFetchUtf8Bytes_	Encode a String, MultiByteString, or Uft8 as UTF-8, and fetch the bytes of the encoded result.
GciFloatKind	Obtain the float kind corresponding to a C double value.
GciFltToOop	Convert a C double value to a SmallDouble or Float object.
GCI_I64_IS_SMALL_INT	(MACRO) Determine whether or not a 64-bit C integer is in the SmallInteger range.
GciI32ToOop	Convert a C 32-bit integer value to a GemStone object.
GciI64ToOop	Convert a C 64-bit integer value to a GemStone object.
GCI_OOP_IS_BOOL	(MACRO) Determine whether or not a GemStone object represents a GemStone Smalltalk Boolean.

Table 7.7 Functions and Macros for Converting Objects and Values (Continued)

GCI_OOP_IS_SMALL_INT	(MACRO) Determine whether or not a GemStone object represents a SmallInteger.
GCI_OOP_IS_SPECIAL	(MACRO) Determine whether or not a GemStone object has a special representation.
GciOopToBool	Convert a Boolean object to a C Boolean value.
GCI_OOP_TO_BOOL	(MACRO) Convert a Boolean object to a C Boolean value.
GciOopToChar16	Convert a Character object to a 16-bit C character value.
GciOopToChar32	Convert a Character object to a 32-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.
GciOopToFlt	Convert a SmallDouble, Float, or SmallFloat object to a C double.
GciOopToI32	Convert a GemStone object to a C 32-bit integer value.
GciOopToI64	Convert a GemStone object to a C 64-bit integer value.
GciPointerToByteArray	Given a C pointer, return a SmallInteger or ByteArray containing the value of the pointer.
GciStringToInteger	Convert a C string to a GemStone SmallInteger or LargeInteger object.

Table 7.8 Object Traversal and Path Functions and Macros

GCI_ALIGN	(MACRO) Align an address to a word boundary.
GciClampedTrav, GciNbClampedTrav	Traverse an array of objects, subject to clamps.
GciExecuteStrTrav, GciNbExecuteStrTrav	Execute a string and traverse the result of the execution.
GciFetchPaths	Fetch selected multiple OOPs from an object tree.
GciFindObjRep	Fetch an object report in a traversal buffer.
GciMoreTraversal, GciNbMoreTraversal	Continue object traversal, reusing a given buffer.
GciObjRepSize_	Find the number of bytes in an object report.
GciPerformTrav GciNbPerformTrav	First send a message to a GemStone object, then traverse the result of the message.
GciPerformTraverse	First send a message to a GemStone object, then traverse the result of the message.
GciSetTraversalBufSwizzli ng	Control swizzling of the traversal buffers.
GciStorePaths	Store selected multiple OOPs into an object tree.
GciStoreTrav GciNbStoreTrav	Store multiple traversal buffer values in objects.
GciStoreTravDo_, GciNbStoreTravDo_	Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object.
GciStoreTravDoTrav_ GciNbStoreTravDoTrav_	Combine in a single function the calls to GciStoreTravDo_ and GciClampedTrav, to store multiple traversal buffer values in objects, execute specified code, and traverse the result object.
GciStoreTravDoTravRefs_ GciNbStoreTravDoTravRef s_	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.
GciTraverseObjs, GciNbTraverseObjs	Traverse an array of GemStone objects.

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 7.9 Structural Access Functions and Macros

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

Table 7.9 Structural Access Functions and Macros (Continued)

GciIvNameToIdx	Fetch the index of an instance variable name.
GciNscIncludesOop	Determines whether the given OOP is present in the specified unordered collection.
GciObjExists	Determine whether or not a GemStone object exists.
GciObjInCollection	Determine whether or not a GemStone object is in a Collection.
GciObjIsCommitted	Determine whether or not an object is committed.
GciRemoveOopFromNsc	Remove an OOP from an NSC.
GciRemoveOopsFromNsc	Remove one or more OOPs from an NSC.
GciReplaceOops	Replace all instance variables in a GemStone object.
GciReplaceVaryingOops	Replace all unnamed instance variables in an NSC object.
GciSetVaryingSize	Set the size of a collection.
GciStoreByte	Store one byte in a byte object.
GciStoreBytes	(MACRO) Store multiple bytes in a byte object.
GciStoreBytesInstanceOf	Store multiple bytes in a byte object.
GciStoreChars	Store multiple ASCII characters in a byte object.
GciStoreIdxOop	Store one OOP in an indexable pointer object’s unnamed instance variable.
GciStoreIdxOops	Store one or more OOPs in an indexable pointer object’s unnamed instance variables.
GciStoreNamedOop	Store one OOP into an object’s named instance variable.
GciStoreNamedOops	Store one or more OOPs into an object’s named instance variables.
GciStoreOop	Store one OOP into an object’s instance variable.
GciStoreOops	Store one or more OOPs into an object’s instance variables.

Table 7.10 Utility Functions

GciAll7Bit	Determine if a String contains only 7-bit ASCII characters.
GciCompress	Compress the supplied data, which can be uncompressed with GciUncompress.
GciDecodeOopArray	Decode an OOP array that was previously run-length encoded.
GciDecSharedCounter	Decrement the value of a shared counter.
GciEnableFreeOopEncoding	Enable encoding of free OOPs sent between the client and the RPC Gem.
GciEnableFullCompression	Enable full compression between the client and the RPC version of GemBuilder.
GciEncodeOopArray	Encode an array of OOPs, using run-length encoding.

Table 7.10 Utility Functions (Continued)

GciFetchNumEncodedOops	Obtain the size of an encoded OOP array.
GciFetchNumSharedCounters	Obtain the number of shared counters available on the shared page cache used by this session.
GciFetchSharedCounterValuesNoLock	Fetch the value of multiple shared counters without locking them.
GciIncSharedCounter	Increment the value of a shared counter.
GciProduct	Return an 8-bit unsigned integer that indicates the GemStone/S product.
GciReadSharedCounter	Lock and fetch the value of a shared counter.
GciReadSharedCounterNoLock	Fetch the value of a shared counter without locking it.
GciSetSharedCounter	Set the value of a shared counter.
GciUncompress	Uncompress the supplied data, assumed to have been compressed with GciCompress.
GciVersion	Return a string that describes the GemBuilder version.

7.2 Executing the examples

The following topaz input defines classes and objects that are used in some of the examples for functions.

```
run
Array subclass: #Account
   instVarNames: #(#id #customer #salesRep) inDictionary: UserGlobals.
Object subclass: #Customer
   instVarNames: #(#name #address) inDictionary: UserGlobals.
Object subclass: #Address
   instVarNames: #(#street #city #state) inDictionary: UserGlobals.
%

run
Account compileMissingAccessingMethods.
Customer compileMissingAccessingMethods.
Address compileMissingAccessingMethods.
%

method: Address
asString
^ self street, ' ', self city, ', ', self state
%
method: Customer
asString
^ self name, ', ', self address asString.
```

```
%
method: Account
asString
^'Account ', self id asString, ' for ', self customer asString
%

run
UserGlobals at: #AllAccounts put: IdentityBag new.
%

run
| createBlk |
createBlk := [:data | | addr cust acct |
   addr := Address new street: (data at: 2); city: (data at: 3);
      state: (data at: 4); yourself.
   cust := Customer new name: (data at: 1);  address: addr;
      yourself.
   acct := Account new id: (data at: 5); customer: cust;
      salesRep: (data at: 6); yourself.
   acct addAll: (data at: 6).
   acct].
{
{'Sam Houston' . '2321 Walnut Dr' . 'Myrtle Point' .
   'OR' . 7212. 'Mark Rabinowitz'. { 'GS64' . 'GBS/VA'} } .
{'Maya Johnson' . '12 Main Str' .  'Ashland' . 'OR' .
   2323. 'Mark Rabinowitz' . {'GS64' . 'GBS/VW' . 'GemConnect'}} .
{'Mario Harar' . '12 N. First Str' . 'Washougal' . 'WA' .
   10073. 'Linda Waffle'. { 'GS64' } } }
     do:
         [:dataArr | AllAccounts add: (createBlk value: dataArr)].
System commitTransaction.
%
```

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).`

See Also

"Interrupting GemStone Execution" on page 30
GciCommit on page 117
GciNbAbort on page 236

GciAddOopToNsc

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

Syntax

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

Arguments

theNsc	The OOP of the NSC from which to remove the object *theOop*.
theOops	The OOPs to be added.

Description

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

Example

```
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 on page 95
GciNscIncludesOop on page 276
GciRemoveOopFromNsc on page 323
GciRemoveOopsFromNsc on page 324

GciAddOopsToNsc

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

Syntax

void **GciAddOopsToNsc**(

OopType	*theNsc*,
const OopType	*theOops*[ ],
int	*numOops*);

Arguments

theNsc	The OOP of the NSC to which to add the OOPs.
theOops	The array of OOPs to be added to the NSC.
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 on page 94
GciNscIncludesOop on page 276
GciRemoveOopFromNsc on page 323
GciRemoveOopsFromNsc on page 324

GCI_ALIGN

(MACRO) Align an address to a word boundary.

Syntax

uintptr_t * GCI_ALIGN(*argument*)

Arguments

argument	The pointer or integer to be aligned.

Return 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; see “Object Report Header - GciObjRepHdrSType” on page 71.

GciAll7Bit

Determine if a String contains only 7-bit ASCII characters.

Syntax

```
BoolType GciAll7Bit(
    const char *            aString,
    size_t *                stringLength);
```

Arguments

aString	A null-terminated string.
stringLength	Returns the length of *aString*, computed by `strlen(`*aString*`)`.

Return Value

Returns TRUE if each character in *aString* is in the 7-bit ASCII range, that is, a value <= 127; FALSE if any characters have values of 128 or higher.

Description

The function **GciAll7Bit** is used to test if a String is in the ASCII range, which will not involve encoding to UTF-8 for storage.

See Also

GciFetchUtf8Bytes_ on page 192

GciAllocTravBuf

Allocate and initialize a new GciTravBufType structure.

Syntax

```
(GciTravBufType *) GciAllocTravBuf(
    size_t                    allocationSize);
```

Arguments

allocationSize	The size of the traversal buffer.

Description

This function allocates and initializes a new GciTravBufType structure.

See Also

“Traversal Buffer - GciTravBufType” on page 78

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 );
```

Arguments

theOops	An array for the of 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*; on return, the number of actual OOPs in the *theOops* result.

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** on page 336). It may also track them by storing them in the GciTrackedObjs set (see **GciSaveAndTrackObjs** on page 334). 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

GciDirtyObjsInit on page 138
GciReleaseAllOops on page 317
GciReleaseOops on page 320
GciSaveAndTrackObjs on page 334
GciSaveObjs on page 336

GciAppendBytes

Append bytes to a byte object.

Syntax

```
void GciAppendBytes(
    OopType                theObject,
    int64                  numBytes,
    const ByteType *       theBytes );
```

Arguments

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

Description

This 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 on page 102
GciStoreBytes on page 357

GciAppendChars

Append a C string to a byte object.

Syntax

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

Arguments

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

Description

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

See Also

GciAppendBytes on page 101
GciStoreChars on page 361

GciAppendOops

Append OOPs to the unnamed variables of a collection.

Syntax

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

Arguments

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

Description

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

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

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

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

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

See Also

GciAddOopsToNsc on page 95
GciStoreOops on page 371

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 on page 93
GciCommit on page 117
GciNbBegin on page 237

GCI_BOOL_TO_OOP

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

Syntax

OopType **GCI_BOOL_TO_OOP**(*aBoolean*)

Arguments

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

Return 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* (OOP_FALSE). Any other C value translates to the GemStone Boolean object *true* (OOP_TRUE).

Example

```
BoolType GCI_BOOL_TO_OOP_example(void)
{
  // zero returns OOP_FALSE
  OopType result = GCI_BOOL_TO_OOP( 0 );
  if ( ! result == OOP_FALSE)
     { return false; }

  // any non-zero argument returns OOP_TRUE
  result = GCI_BOOL_TO_OOP( 99 );
  if ( ! result == OOP_TRUE)
     { return false; }

  return true;
}
```

See Also

GciOopToBool on page 286,
GCI_OOP_TO_BOOL on page 287

GciByteArrayToPointer

Given a result from **GciPointerToByteArray**, return a C pointer.

Syntax

void * **GciByteArrayToPointer**(
    OopType *arg*);

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 on page 306

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

GciInUserAction on page 220

GciCheckAuth

Gather the current authorizations for an array of database objects.

Syntax

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

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*.
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 **GciPerform**.

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

GCI_CHR_TO_OOP_

(MACRO) Convert a 32 bit C character value to a GemStone Character object.

Syntax

OopType **GCI_CHR_TO_OOP**(uint *aChar*)

OopType **GCI_CHR_TO_OOP_**(uint *aChar*)

Arguments

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

Return Value

The OOP of the GemStone Character object that is equivalent to *aChar*. If unit is larger than 0x10FFFF, then **GCI_CHR_TO_OOP_** returns OOP_ILLEGAL, while **GCI_CHR_TO_OOP** returns the Character with codePoint 0x10FFFF.

Description

This macro translates a 32-bit C character value into the equivalent GemStone Character object.

The variants differ according to how they handle out of range input arguments.

Example

```
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 on page 290

GciClampedTrav

Traverse an array of objects, subject to clamps.

Syntax

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

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 “Clamped Traversal Arguments - GciClampedTravArgsSType” on page 77 for details on the specific fields: the OOP for a ClampSpecification, traversal depth, traversal buffer, and retrieval flags. On return, **GciClampedTravArgsSType**->*travBuff* is updated. 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. 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

GciClampedTrav 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 396 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 on page 234
GciSaveObjs on page 336

GciClassMethodForClass

Compile a class method for a class.

Syntax

OopType **GciClassMethodForClass**(
  OopType *source,*
  OopType *aClass*,
  OopType *category*,
  OopType *symbolList*);

Arguments

source	The OOP of a Smalltalk string to be compiled as a class method.
aClass	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, or OOP_NIL. 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 (`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 a list of the optimized selectors.

To *remove* a class method, use **GciExecuteStr** (page 155) to execute Smalltalk code in GemStone, or **GciClassRemoveAllMethods** (page 114) to remove all methods.

Example

See “Executing the examples” on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void GciClassMethodForClass_example(void)
{
  // Assumes the setup code has been executed.

  OopType theClass = GciResolveSymbol("Customer", OOP_NIL);
  OopType oCateg = GciNewString("instance creation");
  OopType oMethodSrc = GciNewString(
      "newNamed: aName address: anAddress.
      | n | n := self new. n name: aName. n address: anAddress. ^ n");

  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

GciCompileMethod on page 118
GciInstMethodForClass on page 218

GciClassNamedSize

Find the number of named instance variables in a class.

Syntax

int **GciClassNamedSize**(
    OopType *aClass*);

Arguments

aClass	The OOP of the class from which to obtain information about instance variables. For reserved OOPs for Smalltalk kernel classes, see `$GEMSTONE/include/gcioop.ht`.

Return Value

Returns the number of named instance variables in the class, or zero if an error occurs.

Description

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

Example

```
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 on page 226

GciClassRemoveAllMethods

GciClassRemoveAllMethods_

Remove all methods from the method dictionary of a class.

Syntax

void **GciClassRemoveAllMethods**(
    OopType   *aClass*);

void **GciClassRemoveAllMethods_**(
    OopType   *aClass*,
    ushort   *environmentId*);

Arguments

aClass	The OOP of the class from which to remove the methods.
environmentId	The compilation environment from which to remove methods.

Description

GciClassRemoveAllMethods removes all methods from the method dictionary of the specified class, within environment 0, the default Smalltalk environment. To remove methods in other environments, use **GciClassRemoveAllMethods_**.

To remove an individual method, use **GciExecuteStr** (page 155) to execute Smalltalk code in GemStone.

GciClearStack

Clear the Smalltalk call stack.

Syntax

void **GciClearStack**(
    OopType *gsProcess*);

Arguments

gsProcess	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.

```
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 on page 122
GciSoftBreak on page 353

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 on page 93
GciBegin on page 104
GciNbCommit on page 239

GciCompileMethod

Compile a method.

Syntax

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

Arguments

source	The OOP of a Smalltalk string to be compiled as a method.
aClass	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, or OOP_NIL. 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 (`System myUserProfile` *symbolList*).
overrideSelector	If not OOP_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. GCI_COMPILE_CLASS_METH means compile a class method. These are listed in `gcicmn.ht`.
environmentId	The compilation environment for method lookup, normally 0.

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 a list of the optimized selectors.

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

See Also

GciClassMethodForClass on page 111
GciInstMethodForClass on page 218

GciCompress

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

Syntax

int **GciCompress**(

char *	*dest,*
uint *	*destLen,*
const char *	*source,*
uint	*sourceLen);*

Arguments

dest	Pointer to the buffer that will 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.

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 on page 399

GciContinue

Continue code execution in GemStone after an error.

Syntax

```
OopType GciContinue(
    OopType                     gsProcess );
```

Arguments

gsProcess	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 30.

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 **GciClearStack** on page 115.

See Also

GciClearStack on page 115
GciContinueWith on page 123
GciErr on page 150
GciNbContinue on page 240
GciSoftBreak on page 353

GciContinueWith

Continue code execution in GemStone after an error.

Syntax

OopType **GciContinueWith** (

OopType	*gsProcess*,
OopType	*replaceTopOfStack*,
int	*flags*,
GciErrSType *	*continueWithError*);

Arguments

gsProcess	The OOP of a GsProcess object (obtained as the value of the context field of an error report returned by GciErr).
replaceTopOfStack	f not OOP_ILLEGAL, replace the value at 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** on page 300.
continueWithError	If not NULL, continue execution by signalling this 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

GciClearStack on page 115
GciContinue on page 122
GciErr on page 150
GciNbContinueWith on page 241
GciSoftBreak on page 353

GciCreateByteObj

Create a new byte-format object.

Syntax

```
OopType GciCreateByteObj(
    OopType                 aClass,
    OopType                 objId,
    const ByteType *        values,
    int64                   numValues,
    int                     clusterId,
    BoolType                makePermanent );
```

Arguments

aClass	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 (OOP) *objId* previously obtained from **GciGetFreeOop** or **GciGetFreeOops**. For more about the semantics of such object identifiers, see **GciGetFreeOop** on page 202.

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, and LargeInteger, 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 on page 126
GciGetFreeOop on page 202
GciGetFreeOops on page 204

GciCreateOopObj

Create a new pointer-format object.

Syntax

```
OopType GciCreateOopObj(
    OopType              aClass,
    OopType              objId,
    const OopType *      values,
    int                  numValues,
    int                  clusterId,
    BoolType             makePermanent );
```

Arguments

aClass	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 **GciGetFreeOop** on page 202.

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, 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 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 on page 124
GciGetFreeOop on page 202
GciGetFreeOops on page 204

GciCTimeToDateTime

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

Syntax

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

Arguments

arg	The C time value to be converted.
result	A pointer to the C struct for 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.

See Also

GciDateTimeToCTime on page 129

GciDateTimeToCTime

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

Syntax

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

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.

See Also

GciCTimeToDateTime on page 128

GciDbgEstablish

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

Syntax

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

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 this function.

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*.

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.

To disable previous debugging routines, call **GciDbgEstablish** with an argument of NULL.

Example

```
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 on page 131
GciDbgLogString on page 132

GciDbgEstablishToFile

Write trace information for most GemBuilder functions to a file.

Syntax

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

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, call **GciDbgEstablishToFile** with an argument of NULL; alternatively, a call **GciShutdown** also teminates tracing.

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

See Also

GciDbgEstablish on page 130
GciDbgLogString on page 132

GciDbgLogString

Pass a message to a trace function.

Syntax

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

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 on page 130
GciDbgEstablishToFile on page 131

GciDeclareAction

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

Syntax

void **GciDeclareAction**(

const char*	*name*,
void*	*func*,
int	*nunArgs*,
uint	*flags*,
BoolType	*errorIfDuplicate*);

Arguments

name	The user action name (a case-insensitive, null-terminated string).
func	A pointer to the C user action function.
numArgs	The number of arguments in the C function.
flags	Flags that apply to the declaration: primarily 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

Chapter 4, "Writing User Actions", starting on page 45
"User Action Information Structure - GciUserActionSType" on page 79
GciInstallUserAction on page 217
GciInUserAction on page 220
GciUserActionInit on page 400
GciUserActionShutdown on page 401

GciDecodeOopArray

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

Syntax

```
int GciDecodeOopArray(
    OopType *          encodedOopArray,
    const int          numEncodedOops,
    OopType *          decodedOopArrray,
    const int          decodedOopsSize);
```

Arguments

encodedOopArray	An OOP array that was encoded by a call to **GciEncodeOopArray**.
numEncodedOops	The number of OOPs in *encodedOopArray.*
decodedOopArray	The decoded OOP array that had been run-length encoded.
decodedOopsSize	The maximum number of OOPs in *decodedOopArray*.

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 on page 177
GciEncodeOopArray on page 147
GciGetFreeOopsEncoded on page 206

GciDecSharedCounter

Decrement the value of a shared counter.

Syntax

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

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. On return, the new value of the shared counter after the decrement.
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.

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 on page 178
GciIncSharedCounter on page 213
GciSetSharedCounter on page 349
GciReadSharedCounter on page 314
GciReadSharedCounterNoLock on page 315
GciFetchSharedCounterValuesNoLock on page 189

GciDirtyExportedObjs

Find all objects in the ExportedDirtyObjs set.

Syntax

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

Arguments

theOops	An array to hold the dirty exported objects.
numOops	The maximum number of objects that can be put into *theOops* buffer. On return, 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 any of the following:

GciDirtyAlteredObjs
GciDirtyExportedObjs
GciDirtyObjsInit
GciDirtySaveObjs
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**, **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.

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 18
GciDirtyExportedObjs on page 136
GciDirtyObjsInit on page 138
GciDirtySaveObjs on page 139
GciDirtyTrackedObjs on page 141
GciSaveGlobalObjs on page 335
GciTrackedObjsFetchAllDirty on page 393

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**, **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.

See Also

GciDirtyExportedObjs on page 136
GciDirtySaveObjs on page 139
GciTrackedObjsFetchAllDirty on page 393

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 );
```

Arguments

theOops	An array to hold the dirty cached objects.
numOops	The maximum number of objects that can be put into *theOops* buffer. On return, 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

This function 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**.

The ExportedDirtyObjs set is cleared by calls to **GciDirtyAlteredObjs**, **GciDirtyExportedObjs**, **GciDirtySaveObjs** (this function), or **GciTrackedObjsFetchAllDirty**.

The TrackedDirtyObjs set is initialized by **GciTrackedObjsInit** .

The TrackedDirtyObjs set is 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 18
GciDirtyExportedObjs on page 136
GciDirtyObjsInit on page 138
GciDirtyTrackedObjs on page 141
GciTrackedObjsFetchAllDirty on page 393
GciSaveObjs on page 336

GciDirtyTrackedObjs

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

Syntax

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

Arguments

theOops	An array to hold the dirty tracked objects.
numOops	The maximum number of objects that can be put into *theOops* buffer. On return, 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 GciTrackedObjs set and have been made “dirty” since the GciTrackedObjs set was initialized or cleared. Functions that initialize or remove objects from the TrackedDirtyObjs set are **GciDirtyAlteredObjs**, **GciDirtySaveObjs**, **GciDirtyTrackedObjs** (this function), **GciTrackedObjsFetchAllDirty** and **GciTrackedObjsInit**.

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.

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 18
GciDirtyTrackedObjs on page 141
GciReleaseAllTrackedOops on page 318
GciSaveAndTrackObjs on page 334
GciTrackedObjsFetchAllDirty on page 393
GciTrackedObjsInit on page 395

Gci_doubleToSmallDouble

Convert a C double to a SmallDouble object.

Syntax

```
OopType Gci_doubleToSmallDouble(
    double                     aDouble );
```

Arguments

aDouble	A C double value

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 on page 201

GciEnableFreeOopEncoding

Enable encoding of free OOPs sent between the client and the RPC Gem.

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.

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.

GciEnableSignaledErrors

Establish or remove GemBuilder visibility to signaled errors from GemStone.

Syntax

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

Arguments

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

Return Value

This function returns TRUE if signaled errors are already visible when it is called.

Description

GemStone permits selective response to signal errors: RT_ERR_SIGNAL_ABORT, RT_ERR_SIGNAL_COMMIT, and RT_ERR_SIGNAL_GEMSTONE_SESSION. The default condition is to leave them all invisible. GemStone responds to each single kind of signal error only after an associated method of class System has been executed: `enableSignaledAbortError`, `enableSignaledObjectsError`, and `enableSignaledGemStoneSessionError` respectively.

After **GciInit** executes successfully, the GemBuilder default condition also leaves all signal errors invisible. The **GciEnableSignaledErrors** function permits GemBuilder to respond automatically to signal errors. However, GemStone must respond to each kind of error in order for GemBuilder to respond to it. Thus, if an application calls **GciEnableSignaledErrors** with *newState* equal to TRUE, then GemBuilder responds automatically to exactly the same kinds of signal errors as GemStone. If GemStone has not executed any of the appropriate System methods, then this call has no effect until it does.

When enabled, GemBuilder checks for signal errors at the start of each function that accesses the database. It treats any that it finds just like any other errors, through **GciErr** or the **GciLongJmp** 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 on page 150
GciPollForSignal on page 307

GciEncodeOopArray

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

Syntax

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

Arguments

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

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 on page 134
GciFetchNumEncodedOops on page 177
GciGetFreeOopsEncoded on page 206

GciEncrypt

Encrypt a password string.

Syntax

```
char * GciEncrypt(
      const char*          password,
      char                 outBuff[],
      unsigned int         outBuffSize);
```

Arguments

password	String containing a password.
outBuff	Array in which to put the encrypted password.
outBuffSize	The maximum number of bytes to place in *outBuff*.

Return Value

Returns a pointer to the first character of the encrypted password in *outBuff*[], or NULL if the encrypted password is larger than *outBuffSize*.

Description

This function encrypts a host or GemStone password for use in **GciSetNetEx** or **GciLogin** with password encyrption enabled.

Example

```
BoolType login_encrypt_example(void)
{
   // assume the netldi been started with -a -g so that host
   // userId and host password are not required.
   const char* stoneName    = "gs64stone";
   const char* gemService   = "gemnetobject";
   const char* gsUserName   = "DataCurator";

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

   char outBuf[1024];
   char * gsPassword = GciEncrypt("swordfish", outBuf,
       sizeof(outBuf));
   if ( gsPassword == NULL ) {
      printf("password encryption failed\n");
      return FALSE;
      }

   GciSetNet(stoneName, "", "", gemService);
   BoolType success = GciLoginEx(gsUserName, gsPassword,
      GCI_LOGIN_PW_ENCRYPTED | GCI_LOGIN_QUIET, 0);
   if (! success) {
     GciErrSType errInfo;
     if (GciErr(&errInfo)) {
       printf("login encrypted password failed; error %d, %s\n",
         errInfo.number, errInfo.message);
      }
   }
  return success;
}
```

See Also

GciLoginEx on page 230
GciSetNetEx on page 346

GciErr

Prepare a report describing the most recent GemBuilder error.

Syntax

```
BoolType GciErr(
    GciErrSType *            errorReport );
```

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 309 for information about using general-purpose error traps in GemBuilder. The section “Error Report Structure - GciErrSType” on page 69 describes the C structure for error reports.

See Also

GciClearStack on page 115
GciContinue on page 122
GciErr on page 150
GciLongJmp on page 233
GciPopErrJump on page 309
GciPushErrJump on page 312
GciRaiseException on page 313
GciSetErrJump on page 340

GciExecute

Execute a Smalltalk expression contained in a GemStone String or Utf8 object.

Syntax

OopType **GciExecute**(
    OopType *source*,
    OopType *symbolList*);

Arguments

source	The OOP of a String or Utf8 containing a sequence of one or more Smalltalk statements to be executed.
symbolList	The OOP of a GemStone symbol list. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. This is roughly equivalent to executing the body of a nameless procedure (method).

In most cases, you may find it more efficient to use **GciExecuteStr**. That function takes a C string as its argument, thus reducing the number of network round-trips required to execute the code. With **GciExecute**, you must first convert the source to a String or Utf8 object (see the following example.) If the source is already one of these kinds of 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 30.

Example

```
BoolType execute_example(void)
{
   OopType oString = GciNewString(" ^ 3 + 4 ");
   OopType result = GciExecute(oString, OOP_NIL);
   if (result == OOP_NIL) {
     printf("error from execution\n");
     return false;
   }

   BoolType conversionErr = FALSE;
   int val = GciOopToI32_(result, &conversionErr);
   if (conversionErr) {
      printf("Error converting result to C int\n");
      } else {
         printf("result = %d\n", val);
      }
      return ! conversionErr;
   }
```

See Also

GciExecuteFromContext on page 153
GciExecuteStr on page 155
GciExecuteStrFetchBytes on page 157
GciExecuteStrFromContext on page 159
GciNbExecute on page 245
GciPerform on page 296

GciExecuteFromContext

Execute a Smalltalk expression contained in a String or Utf8 object as if it were a message sent to another object.

Syntax

OopType **GciExecuteFromContext**(

OopType	*source,*
OopType	*contextObject,*
OopType	*symbolList*);

Arguments

source	The OOP of a String or Utf8 containing a sequence of one or more Smalltalk statements to be executed.
contextObject	OOP_ILLEGAL, or the OOP of any GemStone object. The code to be executed is compiled as if it were a method in the class of *contextObject*. A value of OOP_ILLEGAL or OOP_NO_CONTEXT means no context.
symbolList	The OOP of a GemStone symbol list. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. The source is executed as though *contextObject* were the receiver. That is, the pseudo-variable *self* will have the value *contextObject* during the execution. Messages in the source are executed as defined for *contextObject*.

For example, if *contextObject* is an instance of Association, the *source* can reference the pseudo-variables *key* and *value* (referring to the instance variables of the Association *contextObject*). If any pool dictionaries were available to Association, the *source* could reference them too.

In most cases, you may find it more efficient to use **GciExecuteStrFromContext**. That function takes a C string as its argument, thus reducing the number of network round-trips required to execute the code. With **GciExecuteFromContext**, you must first convert the source to a String or Utf8 object. If the source is already one of these kinds of 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 30.

See Also

GciExecuteStr on page 155
GciExecuteStrFetchBytes on page 157
GciExecuteStrFromContext on page 159
GciNbExecuteFromContextDbg_ on page 246
GciNbExecuteStrFromContext on page 251
GciPerform on page 296

GciExecuteStr

Execute a Smalltalk expression contained in a C string.

Syntax

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

Arguments

source	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
symbolList	The OOP of a GemStone symbol list. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution.

If the source is already a String object, you may find it more efficient to use **GciExecute**. That function takes the OOP of a String as its argument.

Because **GciExecuteStr** calls the virtual machine, the user can issue a soft break while this function is executing. For more information, see .“Interrupting GemStone Execution” on page 30.

Example

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

  // get the value associated with key "#GciStructsMd5" in that
  // symbolList ; expected to be a kind of String
  OopType keysum = GciExecuteStr("GciStructsMd5", symbolList);

  // fetch characters of the String
  char buf[1024];
  GciFetchChars_(keysum, 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("#GciStructsMd5 is %s \n", buf);
  }
}
```

See Also

GciExecute on page 151
GciExecuteStrFetchBytes on page 157
GciExecuteStrFromContext on page 159
GciNbExecuteStr on page 248
GciPerform on page 296

GciExecuteStrFetchBytes

Execute a Smalltalk expression contained in a C string, returning byte-format results.

Syntax

int64 **GciExecuteStrFetchBytes**(

const char *	*source*,
int64	*sourceSize*,
OopType	*sourceClass*,
OopType	*contextObject*,
OopType	*symbolList*,
ByteType *	*result*,
int64	*maxResultSize*);

Input

source	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
sourceSize	The number of bytes in the *source*, or -1. If *sourceSize* is -1, `strlen(`*source*`)` is used.
sourceClass	The OOP of a class to use for the compilation target for the text in *sourceStr*, typically OOP_CLASS_STRING, OOP_CLASS_Utf8, or OOP_CLASS_Unicode7.
contextObject	OOP_ILLEGAL, or the OOP of any GemStone object. The code to be executed is compiled as if it were a method in the class of *contextObject*. A value of OOP_ILLEGAL or OOP_NO_CONTEXT means no context.
symbolList	The OOP of a GemStone symbol list. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).
resultBuf	Array containing the bytes of the result of the execution.
maxResultSize	The maximum size of the resulting string.

Return Value

Returns the number of bytes returned in the result buffer, or -1 if an error occurred.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. The execution result, which should be a byte format object is returned in the **result* buffer.

Execution is in environment 0 using GCI_PERFORM_FLAG_ENABLE_DEBUG .

See Also

GciExecute on page 151
GciExecuteStr on page 155
GciExecuteStrFromContext on page 159
GciNbExecuteStrFetchBytes on page 249
GciPerform on page 296

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 );
```

Arguments

sourceStr	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
contextObject	The OOP of any GemStone object. The code to be executed is compiled as if it were a method in the class of *contextObject*. A value of OOP_ILLEGAL or OOP_NO_CONTEXT means no context.
symbolList	The OOP of a GemStone symbol list. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

Returns the OOP of the execution result. In case of error, this function returns OOP_NIL.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. The source is executed as though *contextObject* were the receiver. That is, the pseudo-variable self will have the value *contextObject* during the execution. Messages in the source are executed as defined for *contextObject*.

For example, if *contextObject* is an instance of Association, the source can reference the pseudo-variables *key* and *value* (referring to the instance variables of the Association *contextObject*). If any pool dictionaries were available to Association, the source could reference them too.

Because **GciExecuteStrFromContext** calls the virtual machine, the user can issue a soft break while this function is executing. For more information, see “Interrupting GemStone Execution” on page 30.

Example

```
void executeFromContextExample(void)
{
  // get the Assocation 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

GciExecute on page 151
GciExecuteStr on page 155
GciExecuteStrFetchBytes on page 157
GciNbExecuteStrFromContext on page 251
GciPerform on page 296

GciExecuteStrTrav

First execute a Smalltalk expression contained in a C string as if it were a message sent to an object, then traverse the result of the execution.

Syntax

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

Arguments

sourceStr	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
contextObject	OOP_ILLEGAL, or the OOP of any GemStone object. The code to be executed is compiled as if it were a method in the class of *contextObject*. A value of OOP_ILLEGAL or OOP_NO_CONTEXT means no context.
symbolList	The OOP of a GemStone symbol list. 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**, as described under “Clamped Traversal Arguments - GciClampedTravArgsSType” on page 77. On return, the instance of **GciClampedTravArgsSType**, with **GciClampedTravArgsSType**->*travBuff* filled. 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

GciExecute on page 151
GciExecuteStr on page 155
GciMoreTraversal on page 234

GciNbExecuteStrTrav on page 252
GciPerformTrav on page 303
GciPerformTraverse on page 304

GciFetchByte

Fetch one byte from an indexed byte object.

Syntax

```
ByteType GciFetchByte(
    OopType         theObject,
    int64           atIndex );
```

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_ on page 164
GciFetchChars_ on page 166
GciFetchOop on page 182
GciStoreByte on page 356

GciFetchBytes_

Fetch multiple bytes from an indexed byte object.

Syntax

int64 **GciFetchBytes_**(

OopType	*theObject*,
int64	*startIndex*,
ByteType	*theBytes*[ ],
int64	*numBytes*);

Arguments

theObject	The OOP of the GemStone byte object.
atIndex	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.
theBytes	The array of fetched bytes.
numBytes	The maximum number of bytes to return.

Return Value

Returns the number of bytes fetched. (This may be less than *numBytes*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function fetches multiple elements from a byte object starting at the specified index, using structural access. A common application of **GciFetchBytes_** would be to fetch a text string.

GciFetchBytes_ permits *theObject* to be a byte object with multiple bytes per character or digit, such as DoubleByteString, Float and LargeInteger. In this case, **GciFetchBytes_** provides automatic byte swizzling to client native byte order. For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26. For MultiByteStrings, *startIndex* must be aligned on character boundaries and *numbytes* must be a multiple of the number of bytes per character; for numeric objects *startindex* must be one and *numBytes* the size of the numeric class.

Example

This example will print a GemStone String of arbitrarily large size, by repeatedly fetching fixed size buffers.

```
BoolType fetchBytes_example(OopType anObj)
{
   int BUF_SIZE =  5000;
   char buff[BUF_SIZE];
   BoolType done = FALSE;
   int idx = 1;
   GciErrSType err;
   while (! done) {
      int64 numRet = GciFetchBytes_(anObj, idx, (ByteType*)buff,
               BUF_SIZE - 1);
      if (numRet == 0) {
         done = TRUE;  // hit end of object or error
         if (GciErr(&err)) {
            printf("error %d, %s\n", err.number, err.message);
            return false;
         }
      } else {
         buff[numRet] = '\0';
         printf("%s\n", buff);
         idx += numRet;
      }
   }
   return true;
}
```

See Also

GciFetchByte on page 163
GciFetchChars_ on page 166
GciFetchOop on page 182
GciStoreBytes on page 357

GciFetchChars_

Fetch multiple ASCII characters from an indexed byte object.

Syntax

int64 **GciFetchChars_**(

OopType	*theObject*,
int64	*startIndex*,
char *	*cString*,
int64	*maxSize*);

Arguments

theObject	The OOP of an object containing ASCII characters.
startIndex	The index of the first character to retrieve.
cString	Pointer to the location in which to store the returned string.
maxSize	Maximum number of characters to fetch.

Return Value

Returns the number of characters fetched.

Description

Equivalent to **GciFetchBytes_**, except that it is assumed that *theObject* contains ASCII text. The bytes fetched are stored in memory starting at *cString*. At most *maxSize* - 1 bytes will be fetched from the object, and a `\0` character will be stored in memory following the bytes fetched. The function returns the number of characters fetched, excluding the null terminator character, which is equivalent to `strlen(cString)` if the object does not contain any null characters. If an error occurs, the function result is 0, and the contents of *cString* are undefined.

See Also

GciFetchByte on page 163
GciFetchBytes_ on page 164
GciStoreChars on page 361

GciFetchClass

Fetch the class of an object.

Syntax

OopType **GciFetchClass**(
    OopType *theObject*);

Arguments

theObject	The OOP of the object.

Return Value

Returns the OOP of the object's class. In case of error, this function returns OOP_NIL.

The GemBuilder include file `$GEMSTONE/include/gcioop.ht` defines a C constant for each of the Smalltalk kernel classes.

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 SmallDouble
  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 on page 175
GciFetchObjImpl on page 181
GciFetchSize_ on page 190
GciFetchVaryingSize_ on page 198

GciFetchDateTime

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

Syntax

```
void GciFetchDateTime(
    OopType                  datetimeObj,
    GciDateTimeSType *       result );
```

Arguments

datetimeObj	The OOP of the object to fetch.
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 );
```

Arguments

theObject	The OOP of the 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 on page 171
GciStoreDynamicIv on page 362

GciFetchDynamicIvs

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

Syntax

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

Arguments

theObject	The OOP of the object.
buf	C pointer to the buffer that will contain the object's dynamic instance variables.
numOops	The maximum number of elements to return.

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 on page 170
GciStoreDynamicIv on page 362

GciFetchNamedOop

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

Syntax

OopType **GciFetchNamedOop**(
OopType *theObject*,
int *atIndex*);

Arguments

theObject	The OOP of the 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

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
BoolType fetchNamedOop_example(OopType anAccount)
{
   // fetch anAccount's salesRep, hard coding offset
   int indexOfSalesRep = 3;
   OopType oName1 = GciFetchNamedOop(anAccount, indexOfSalesRep);

   // fetch anAccount's salesRep without hardcoding offset
   int ivOffset = GciIvNameToIdx(GciFetchClass(anAccount), "salesRep");
   OopType oName2 = GciFetchNamedOop(anAccount, ivOffset);

   return oName1 = oName2;
}
```

See Also

GciFetchNamedOops on page 173
GciFetchVaryingOop on page 194
GciIvNameToIdx on page 226
GciStoreNamedOop on page 366

GciFetchNamedOops

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

Syntax

int **GciFetchNamedOops**(
    OopType *theObject*,
    int *startIndex*,
    OopType *theOops*[ ],
    int *numOops*);

Arguments

theObject	The OOP of the 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 size of the object, this function returns a byte array of size 0, but no error is generated.
theOops	The array of fetched OOPs.
numBytes	The maximum number of elements to return.

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

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

This example fetches an Account's id, customer, and salesRep.

```
BoolType fetchNamedOops_example(OopType anAccount)
{
   GciErrSType err;
   // fetch the number of named instvars

   int namedSize = GciFetchNamedSize(anAccount);
   if (namedSize == 0) {  return  false;  }

   OopType *oBuffer = (OopType*) malloc( sizeof(OopType) * namedSize );
   if (oBuffer == NULL) { return  false; }
```

```
      int numRet = GciFetchNamedOops(anAccount, 1, oBuffer, namedSize);
      if (GciErr(&err)) {
         printf("Error %d, %s\n", err.number, err.message);
         return false;
      }
      free(oBuffer);
      return (numRet == namedSize) ;
   }
```

See Also

GciFetchNamedOop on page 172
GciFetchVaryingOops on page 196
GciIvNameToIdx on page 226
GciStoreNamedOops on page 367

GciFetchNamedSize

Fetch the number of named instance variables in an object.

Syntax

int **GciFetchNamedSize**(
    OopType *theObject*);

Arguments

theObject	The OOP of the object.

Return Value

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

Description

This function returns the number of named instance variables in a GemStone object. See the example for **GciFetchNamedOops** on page 173.

GciFetchNameOfClass

Fetch the class name object for a given class.

Syntax

```
OopType GciFetchNameOfClass(
    OopType                  aClass );
```

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

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

Arguments

encodedOopArray	An OOP array that was encoded by a call to **GciEncodeOopArray**.
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 on page 134
GciEnableFreeOopEncoding on page 144
GciEncodeOopArray on page 147
GciGetFreeOopsEncoded on page 206

GciFetchNumSharedCounters

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

Syntax

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 on page 135
GciIncSharedCounter on page 213
GciSetSharedCounter on page 349
GciReadSharedCounter on page 314
GciReadSharedCounterNoLock on page 315
GciFetchSharedCounterValuesNoLock on page 189

GciFetchObjectInfo

Fetch information and values from an object.

Syntax

```
BoolType GciFetchObjectInfo(
    OopType                        theObject,
    GciFetchObjInfoArgsSType *args );
```

Arguments

theObject	OOP of any object with byte, pointer, or NSC format.
args	Pointer to an instance of **GciFetchObjInfoArgsSType** with the following argument fields:

GciObjInfoSType * *info*
Pointer to an instance of **GciObjInfoSType**, or NULL.

ByteType * *buffer*
Pointer to an area where byte or OOP values will be returned, or NULL.

int64 *startIndex*
The offset in the object at which to start fetching, a 1-based offset into the named and unnamed instance variables. Ignored if *bufSize* == 0 or buffer == NULL.

int64 *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 fetching a kind of BinaryFloat and greater than zero, it must be large enough to fetch the complete object.

int64 *numReturned*
On return, this contains the number of logical elements (bytes or OOPs) returned in *buffer*. Note that the size of (**OopType**) is 8 bytes.

int *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 **GciSaveObjs** on page 336).

Return Value

TRUE if successful, FALSE if an error occurs.

Description

GciFetchObjectInfo 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.

The offset in *startIndex* 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

If *theObject* is a byte object with multiple bytes per character or digit, such as DoubleByteString, LargeInteger, or Float, the results in *args->buffer* will be automatically byte swizzled to client native byte order. For more about byte swizzling, see “Byte-Swizzling of Binary Floating-Point Values” on page 26.

For MultiByteStrings, *args->startIndex* must be aligned on a character boundary and *args->bufSize* must be a multiple of the number of bytes per character in the string. For numeric objects, *args->startIndex* must be one and *args->bufSize* must be the size of the numeric class.

See Also

GciFetchOops on page 184
GciFetchBytes_ on page 164
GciFetchOop on page 182
GciSaveObjs on page 336

GciFetchObjImpl

Fetch the implementation of an object.

Syntax

```
int GciFetchObjImpl(
    OopType                 theObject );
```

Arguments

theObject	The OOP of the 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 "Object implementation" on page 73.

See Also

GciFetchClass on page 167
GciFetchNamedSize on page 175
GciFetchSize_ on page 190
GciFetchVaryingSize_ on page 198

GciFetchOop

Fetch the OOP of one instance variable of an object.

Syntax

OopType **GciFetchOop**(
    OopType *theObject*,
    int64 *atIndex*);

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

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples. This example illustrates fetching one named and one unnamed instance variable from an Account.

```
BoolType fetchOop_example(OopType anAccount)
{
   GciErrSType err;
   // fetch the number of named and unnamed instvars
   int namedSize = GciFetchNamedSize(anAccount);
   int64 varyingSize = GciFetchVaryingSize_(anAccount);

   // fetch anAccount's salesRep
   int indexOfSalesRep = 3;
   if (namedSize < indexOfSalesRep) { return false; }
   OopType salesRep = GciFetchOop(anAccount, indexOfSalesRep );
   if (GciErr(&err)) {
      printf("Error %d, %s\n", err.number, err.message);
      return false;   }

   // fetch the first product in the indexed variables
   if (varyingSize < 1) { return false; }
   OopType firstProduct = GciFetchOop(anAccount, namedSize + 1 );
```

```
      if (GciErr(&err)) {
         printf("Error %d, %s\n", err.number, err.message);
         return false;   }

      return true;
   }
```

See Also

GciFetchOops on page 184
GciStoreOop on page 369
GciStoreOops on page 371

GciFetchOops

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

Syntax

int **GciFetchOops**(

OopType	*theObject*,
int64	*startIndex*,
OopType	*theOops*[ ],
int	*numOops*);

Arguments

theObject	The OOP of the source object.
startIndex	The index into *theObject*'s unnamed and named instance variables at which to begin fetching. The index of the first instance variable is 1. Note that if *startIndex* is 1 greater than the size of the object, this function returns an array of size 0, but no error is generated.
theOops	The array of fetched OOPs.
numOops	The maximum number of OOPs to return.

Return Value

Returns the number of OOPs fetched. (This may be less than *numOops*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function fetches the OOPs of multiple instance variables from any object starting at the specified index, using structural access. It does not distinguish between named and unnamed instance variables. Indices are based at the beginning of the object's array of instance variables. In that array, any existing named instance variables are followed by any existing unnamed instance variables.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
BoolType fetchOops_example(OopType anAccount)
{
   GciErrSType err;
   // fetch the number of named and unnamed instvars
   int namedSize = GciFetchNamedSize(anAccount);
   int64 varyingSize = GciFetchVaryingSize_(anAccount);

   int mySize = namedSize + varyingSize;
   OopType oBuffer[mySize];
```

```
      // Two ways to fetch anAccount's products list
      GciFetchOops(anAccount, namedSize + 1, oBuffer, varyingSize);
      GciFetchVaryingOops(anAccount, 1, oBuffer, varyingSize);

      // Fetch the named instance variables PLUS the products list
      GciFetchOops(anAccount, 1, oBuffer, mySize);
      // oBuffer[0..namedSize-1] are named instVar values
      // oBuffer[namedSize] are varying instVar values

      if (GciErr(&err)) {
         printf("Error %d, %s\n", err.number, err.message);
         return false; }

     return true;
   }
```

See Also

GciFetchOops on page 184
GciFetchVaryingOop on page 194
GciStoreOops on page 371

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[ ] );
```

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.
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.

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:

```
      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.

Example 1: Calling sequence for a single object and a single path

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

The first example fetches just the street address from a single Account.

```
OopType fetchPaths_example1(OopType anAccount)
{
   GciErrSType errInfo;
   int path_size = 3;
   int aPath[path_size];
   aPath[0] = 2; //name
   aPath[1] = 2; //address
   aPath[2] = 1; //street

   OopType result;
   GciFetchPaths(&anAccount, 1, aPath, &path_size, 1, &result);
   if (GciErr(&errInfo)) {
      printf("Error %d, %s\n", errInfo.number, errInfo.message);
      return OOP_NIL;
   }
   return result;
}
```

Example 2: Calling sequence for multiple objects with a multiple paths

This example fetches the customer name, and the address city, from two separate accounts.

```
BoolType fetchPaths_example2(int numOfAccounts, OopType* resultAccounts)
{
   GciErrSType errInfo;
   OopType myAccts;
   myAccts = GciExecuteStr("AllAccounts", OOP_NIL);
   if (myAccts == OOP_NIL) { return false; }

   int numRet = GciFetchVaryingOops(myAccts, 1, resultAccounts,
               numOfAccounts);
   if (numRet != numOfAccounts) { return false; }

   int path_segments[2];
   path_segments[0] = 2;
   path_segments[1] = 3;
   int serialPath[6]; // size is sum of path_segments
   serialPath[0] = 2; //customer
   serialPath[1] = 1; //name
   serialPath[2] = 2; //customer
   serialPath[3] = 2; //address
   serialPath[4] = 2; //city

   OopType results[2 * numOfAccounts]; // size of path_segments * number
               of accounts
   GciFetchPaths(resultAccounts, numOfAccounts, serialPath,
               path_segments, 2, results);

   if (GciErr(&errInfo)) {
      printf("GciFetchPaths error %d, %s\n", errInfo.number,
               errInfo.message);
      return false;
   }
   return true;
}
```

See Also

GciStorePaths on page 373

GciFetchSharedCounterValuesNoLock

Fetch the value of multiple shared counters without locking them.

Syntax

int **GciFetchSharedCounterValuesNoLock**(

int	*startIndex,*
int64_t	*buffer*[ ],
size_t *	*maxReturn*);

Arguments

startIndex	The offset into the shared counters array of the first shared counter value to fetch.
buffer	Pointer to a buffer where the shared counter values will be stored. The buffer must be at least 8 * *maxReturn* bytes and the address must be aligned on an 8-byte boundary.
maxReturn	Pointer to a value that indicates the maximum number of shared counters to fetch.

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 *counterIdx* (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

GciDecSharedCounter on page 135
GciFetchNumSharedCounters on page 178
GciIncSharedCounter on page 213
GciReadSharedCounter on page 314
GciReadSharedCounterNoLock on page 315
GciSetSharedCounter on page 349

GciFetchSize_

Fetch the size of an object.

Syntax

int64 **GciFetchSize_**(
    OopType *theObject*);

Arguments

theObject The OOP of the object.

Return Value

Returns the size of *theObject*. In case of error, this function returns zero.

Description

This function obtains the size of an object from GemStone.

The result of this function depends on the object's implementation (see **GciFetchObjImpl**). For byte objects, this function returns the number of bytes in the object. (For Strings, this is the number of Characters in the String; for Floats, the size is 23.) For pointer objects, this function returns the number of named instance variables (**GciFetchNamedSize**) plus the number of indexed instance variables, if any (**GciFetchVaryingSize_**). For NSC objects, this function returns the cardinality of the collection. For special objects, the size is always zero.

This differs somewhat from the result of executing the Smalltalk method `Object>>size`, as shown in Table 7.11:

Table 7.11 Differences in Reported Object Size

Implementation	Object>>size (Smalltalk)	GciFetchSize_
0=Pointer	Number of indexed elements in the object (0 if not indexed)	Number of indexed elements PLUS number of named instance variables
1=Byte	Number of indexed elements in the object	Same as Smalltalk message "size"
2=NSC	Number of elements in the object	Same as Smalltalk message "size"
3=Special	0	0

Example

```
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 on page 167
GciFetchNamedSize on page 175
GciFetchObjImpl on page 181
GciFetchVaryingOop on page 194

GciFetchUtf8Bytes_

GciFetchUtf8Bytes__

Encode a String, MultiByteString, or Uft8 as UTF-8, and fetch the bytes of the encoded result.

Syntax

```
int64 GciFetchUtf8Bytes_(
    OopType             oopOfAString,
    int64               startIndex,
    ByteType *          theBytes,
    int64               numBytes,
    OopType *           utf8String,
    int                 flags);
int64 GciFetchUtf8Bytes__(
    OopType             oopOfAString,
    int64               startIndex,
    ByteType *          theBytes,
    int64               numBytes,
    OopType *           utf8String,
    int                 flags,
    int64               maxConvert);
```

Arguments

oopOfAString	The OOP of the GemStone String, MultiByteString or Utf8.
startIndex	The index into *aString* at which to begin encoding 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.
theBytes	The array of encoded bytes
numBytes	The maximum number of bytes to return.
utf8String	Pointer to the OOP of the encoded string.
flags	Bits in *flags* are defined by GciFetchUtf8Flags. The options are GCI_UTF8_FetchNormal = 0 generate an error on illegal codePoints in the input. GCI_UTF8_NoError = 0x2 return a message instead of signalling Exception.
maxConvert	The upper limit of the size in bytes of the instance of Utf8 in **utf8String*.

Return Value

Returns the number of bytes fetched. (This may be less than *numBytes*, depending upon the size of *theObject*.) In case of error, this function returns zero.

Description

This function encodes a kind of String, MultiByteString, or Utf8 into UTF-8. The encoded bytes are placed in the buffer *theBytes*, and the OOP of the encoded object is placed at *utf8String*.

The OOP of *utf8String* is also placed in the ExportSet. The caller must pass the OOP of *utf8String* to **GciReleaseOops** after fetching all bytes.

If all characters in aString are < 128, or if the class of aString is Utf8, then the behavior is the same as **GciFetchBytes_**. No encoding is done; the bytes of *aString* are place in *theBytes*. *utf8String* is the same as *aString*, and is not added to the ExportSet.

See Also

GciFetchBytes_ on page 164
GciNewUtf8String on page 275

GciFetchVaryingOop

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

Syntax

OopType **GciFetchVaryingOop**(
    OopType *theObject*,
    int64 *atIndex*);

Arguments

theObject	The OOP of the pointer or NSC object.
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

See "Executing the examples" on page 91 for the smalltalk code that supports the examples.

```
OopType fetchVaryingOop_example(OopType anAccount)
{
   GciErrSType err;
   int64 varyingSize = GciFetchVaryingSize_(anAccount);
   if (varyingSize < 1) { return OOP_NIL; }

   // fetch the first product in the indexed variables
   OopType firstProduct = GciFetchVaryingOop(anAccount, 1);
   if (GciErr(&err)) {
      printf("Error %d, %s\n", err.number, err.message);
      return OOP_NIL;   }

   return firstProduct;
}
```

See Also

GciFetchNamedOop on page 172
GciFetchOop on page 182

GciFetchVaryingOops on page 196
GciStoreIdxOop on page 363

GciFetchVaryingOops

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

Syntax

int **GciFetchVaryingOops**(

OopType	*theObject*,
int64	*startIndex,*
OopType	*theOops*[ ],
int	*numOops*);

Arguments

theObject	The OOP of the pointer or NSC object.
startIndex	The index into *theObject*'s elements at which to begin fetching. The index of the first unnamed instance variable is 1. Note that if *startIndex* is 1 greater than the number of unnamed instance variables of the object, this function returns a byte array of size 0, but no error is generated.
theOops	The array of fetched OOPs.
numBytes	The maximum number of elements to return.

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

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
BoolType fetchVaryingOops_example(OopType anAccount)
{
   GciErrSType err;
   int64 varyingSize = GciFetchVaryingSize_(anAccount);
   if (varyingSize < 1) { return false; }

   OopType oBuffer[varyingSize];

   // fetch OOPs for all indexed instvars
   int numRet = GciFetchVaryingOops(anAccount, 1, oBuffer, varyingSize);
```

```
      if (GciErr(&err)) {
         printf("Error %d, %s\n", err.number, err.message);
         return false; }

      return numRet = varyingSize;
   }
```

See Also

GciFetchNamedOops on page 173
GciFetchVaryingOop on page 194
GciFetchVaryingSize_ on page 198
GciStoreIdxOop on page 363
GciStoreIdxOops on page 364

GciFetchVaryingSize_

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

Syntax

int64 **GciFetchVaryingSize_**(
    OopType *theObject*);

Arguments

theObject	The OOP of the object.

Return Value

Returns the number of unnamed instance variables in *theObject*. In case of error, this function returns zero.

Description

This function obtains from GemStone the number of indexed variables in an indexable object or the number of unordered variables in an NSC.

Example

See an example under **GciFetchOop** on page 182.

See Also

GciFetchClass on page 167
GciFetchNamedSize on page 175
GciFetchObjImpl on page 181
GciFetchSize_ on page 190
GciSetVaryingSize on page 351

GciFindObjRep

Fetch an object report in a traversal buffer.

Syntax

GciObjRepHdrSType * **GciFindObjRep**(
    GciTravBufType * *travBuff*,
    OopType *theObject*);

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.

Example

```
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

GciTraverseObjs on page 396
GciObjRepSize_ on page 280

GciFloatKind

Obtain the float kind corresponding to a C double value.

Syntax

GciFloatKindEType **GciFloatKind**(
  double *aReal);

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 on page 201
GciOopToFlt on page 292

GciFltToOop

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

Syntax

OopType **GciFltToOop**(
    double *aDouble*);

Arguments

aDouble	A C double value

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 on page 292
Gci_doubleToSmallDouble on page 143

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. Then **GciCreate** can be used to satisfy the forward references and **GciCommit** can be attempted again. **GciAbort** removes all unsatisfied forward references from the session's object space, just as it removes any other uncommitted modifications.

As long as it remains an unresolved forward reference, the identifier returned by **GciGetFreeOop** can be used only as a parameter to the following function calls, under the given restrictions:

- As the *objID* of the object to be created
  `GciCreateByteObj`
- As the *objID* of the object to be created, or as an element of the value buffer
  `GciCreateOopObj`
- As an element of the value buffer only
  ```
  GciStoreOop
  GciStoreOops
  GciStoreIdxOop
  GciStoreIdxOops
  GciStoreNamedOop
  GciStoreNamedOops
  GciStoreTrav
  GciAppendOops
  GciAddOopToNsc
  GciAddOopsToNsc
  GciNewOopUsingObjRep
  ```
- As an element of *newValues* only
  `GciStorePaths`

See Also

GciCreateByteObj on page 124
GciCreateOopObj on page 126
GciGetFreeOops on page 204
GciGetFreeOopsEncoded on page 206

GciGetFreeOops

Allocate multiple OOPs.

Syntax

```
void GciGetFreeOops(
    int                  count,
    OopType *            resultOops );
```

Arguments

count	The number of OOPs to allocate. On return, the number of OOPs that were allocated.
resultOops	An array to hold the allocated OOPs.

Return Value

Returns an unused object identifier (OOP).

Description

Allocates object identifiers without creating objects.

If an object identifier returned from **GciGetFreeOops** is used as a value in a **GciStore**... call before it is used as the *objId* argument of a **GciCreate**... call, then an unresolved forward reference is created in object memory. This is a reference to an object that does not yet exist. This forward reference must be satisfied by using the identifier as the *objId* argument to a **GciCreate**... call before a **GciCommit** can be successfully executed.

If **GciCommit** is attempted prior to satisfying all unresolved forward references, an error is generated and **GciCommit** returns false. In this case, **GciCreate** can be used to satisfy the forward references and **GciCommit** can be attempted again. **GciAbort** removes all unsatisfied forward references from the session’s object space, just as it removes any other uncommitted modifications.

As long as it remains an unresolved forward reference, the identifier returned by **GciGetFreeOops** can be used only as a parameter to the following function calls, under the given restrictions:

- As the *objID* of the object to be created

  ```
  GciCreateByteObj
  ```

- As the *objID* of the object to be created, or as an element of the value buffer

  ```
  GciCreateOopObj
  ```

- As an element of the value buffer, only

  ```
  GciStoreOop
  GciStoreOops
  GciStoreIdxOop
  GciStoreIdxOops
  GciStoreNamedOop
  GciStoreNamedOops
  GciStoreTrav
  GciAppendOops
  GciAddOopToNsc
  GciAddOopsToNsc
  GciNewOopUsingObjRep
  ```

- As an element of *newValues*, only

  ```
  GciStorePaths
  ```

See Also

GciCreateByteObj on page 124
GciCreateOopObj on page 126
GciGetFreeOop on page 202
GciGetFreeOopsEncoded on page 206

GciGetFreeOopsEncoded

Allocate multiple OOPs.

Syntax

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

Arguments

count	The number of OOPs to allocate. On return, the number of OOPs that were allocated.
encodedOops	A pointer to memory for holding encoded OOPs. Must be at least the size specified by *count*.

Description

This function is identical to **GciGetFreeOops**, 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

GciCreateByteObj on page 124
GciCreateOopObj on page 126
GciDecodeOopArray on page 134
GciEncodeOopArray on page 147
GciFetchNumEncodedOops on page 177
GciGetFreeOop on page 202
GciGetFreeOops on page 204
GciGetFreeOopsEncoded on page 206
GciFetchNumEncodedOops on page 177

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

GciSetSessionId on page 348

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. **GciHardBreak** has no effect if called from within a User Action.

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 on page 353

GciHiddenSetIncludesOop

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

Syntax

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

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, including the *PureExportSet* and the *GciTrackedObjs* (among others). **GciHiddenSetIncludesOop** allows you to pass in an index to a specified hidden set to determine if the set includes a 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;
}
```

GciI32ToOop

Convert a C 32-bit integer value to a GemStone object.

Syntax

```
OopType GciI32ToOop(
    int                         anInt );
```

Arguments

anInt A C 32-bit signed integer.

Return Value

The **GciI32ToOop** function returns the OOP of a GemStone object whose value is equivalent to *anInt*.

Description

The **GciI32ToOop** function translates a C 32bit integer value into the equivalent GemStone object. This function always succeeds.

See Also

GciI64ToOop on page 212
GciOopToI32 on page 294

GCI_I64_IS_SMALL_INT

(MACRO) Determine whether or not a 64-bit C integer is in the SmallInteger range.

Syntax

BoolType **GCI_I64_IS_SMALL_INT**(*anInt*)

Arguments

anInt	A C 64-bit signed integer.

Return Value

Returns TRUE if the object is within the SmallInteger range, FALSE otherwise.

Description

This macro tests to see if *anInt* is within the SmallInteger range.

A SmallInteger has a 61 bit 2's complement integer and 3 tag bits. For a positive int64 argument, top 4 bits must be 2r0000 for argument to be within SmallInteger range; for a negative int64 argument, the top 4 bits must be 2r1111 for argument to be within SmallInteger range.

See Also

GCI_OOP_IS_SMALL_INT on page 284

GciI64ToOop

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

Syntax

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

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 to the export set as described under **GciSaveObjs** on page 336.

See Also

GciI32ToOop on page 210
GciOopToI64 on page 295

GciIncSharedCounter

Increment the value of a shared counter.

Syntax

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

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. On return, this will hold the new value after 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, it will remain set to INT_MAX.

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

See Also

GciDecSharedCounter on page 135
GciFetchNumSharedCounters on page 178
GciFetchSharedCounterValuesNoLock on page 189
GciReadSharedCounter on page 314
GciReadSharedCounterNoLock on page 315
GciSetSharedCounter on page 349

GciInit

Initialize GemBuilder.

Syntax

BoolType **GciInit**( );

Return Value

The function **GciInit** returns TRUE or FALSE to indicate successful or unsuccessful initialization of GemBuilder.

Description

This 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 **GciInitAppName**, 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 43.)

When GemBuilder is initialized on UNIX platforms, it establishes its own handler for SIGIO interrupts. See “Signal Handling in Your GemBuilder Application” on page 37 for information on **GciInit**’s handling of interrupts and pointers on making GemBuilder, application, and third-party handlers work together.

See Also

GciInitAppName on page 215

GciInitAppName

GciInitAppName_

Override the default application configuration file name.

Syntax

void **GciInitAppName**(
 const char * *applicationName*,
 BoolType *logWarnings*);

void **GciInitAppName_**(
 const char * *applicationName*,
 BoolType *logWarnings*,
 unsigned int *gemTOCOverrideKB*,
 int *gemNativeCodeOverride*);

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.
gemTOCOverrideKB	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**.
gemNativeCodeOverride	If non-zero, value overrides the GEM_NATIVE_CODE_ENABLED config file settings in configuration files read by **GciInit**.

Description

This 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.

The **GciInitAppName_** variant allows you to specify additional arguments that are used by the linkable GemBuilder application: the maximum temporary object cache size, and an override for the native code configuration.

You application may not call both **GciInitAppName** and **GciInitAppName_**.

See Also

GciInit on page 214

GciInstallUserAction

GciInstallUserAction_

Associate a C function with a Smalltalk user action.

Syntax

void **GciInstallUserAction**(
    GciUserActionSType *   *userAction*);

void **GciInstallUserAction_**(
    GciUserActionSType *   *userAction*,
    BoolType   *errorIfDuplicate*);

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 4, “Writing User Actions”, starting on page 45

See Also

Chapter 4, “Writing User Actions”, starting on page 45
“User Action Information Structure - GciUserActionSType” on page 79
GciDeclareAction on page 133
GciInUserAction on page 220
GciUserActionInit on page 400
GciUserActionShutdown on page 401

GciInstMethodForClass

Compile an instance method for a class.

Syntax

```
OopType GciInstMethodForClass(
    OopType                    source,
    OopType                    aClass,
    OopType                    category,
    OopType                    symbolList );
```

Arguments

source	The OOP of a Smalltalk string to be compiled as a instance method.
aClass	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, or OOP_NIL. 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 (`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.

Note that the Smalltalk virtual machine optimizes a small number of selectors, and you may not compile methods for any of those selectors. See the *Programming Guide* for a list of the optimized selectors.

To *remove* a class method, use **GciExecuteStr** (page 155) to execute an expression removing the method, or **GciClassRemoveAllMethods** (page 114) to remove all methods.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void instanceMethodExample(void)
{
  // Assumes the setup code has been executed.

  OopType theClass = GciResolveSymbol("Customer", OOP_NIL);
```

```
    OopType oCateg = GciNewString("Printing");

    OopType oMethodSrc = GciNewString("asString  ^ 'Customer named ', name
                 asString") ;

    GciInstMethodForClass(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

GciCompileMethod on page 118
GciClassMethodForClass on page 111

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 on page 107

GciIsKindOf

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

Syntax

```
BoolType GciIsKindOf(
    OopType              anObj,
    OopType              possibleClass );
```

Arguments

anObject	The object to be checked.
possibleClass	A class or class history to check the object against.

Return Value

GciIsKindOf returns TRUE when the class of *anObj* or any of its superclasses is in the class history of *possibleClass*. 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 *possibleClass* is simply a class (which is typical), **GciIsKindOf** uses *possibleClass*'s class history. When *possibleClass* is itself a class history, **GciIsKindOf** uses *possibleClass* 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 on page 222
GciIsSubclassOf on page 224
GciIsSubclassOfClass on page 225

GciIsKindOfClass

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

Syntax

```
BoolType GciIsKindOfClass(
    OopType                 anObject,
    OopType                 possibleClass );
```

Arguments

anObject	The object to be checked.
possibleClass	A class to check the object against.

Return Value

GciIsKindOfClass returns TRUE when the class of *anObject* or any of its superclasses is *possibleClass*. It returns FALSE otherwise.

Description

GciIsKindOfClass performs structural access that is equivalent to the `isKindOf:` method of the Smalltalk class Object. It compares *anObject*'s class and superclasses to see if any of them are the *possibleClass*.

Since **GciIsKindOfClass** does *not* consider class histories, it *cannot* help to support schema modification. To accomplish a similar operation when the relationship of types can change over time, use the **GciIsKindOf** function.

See Also

GciIsKindOf on page 221
GciIsSubclassOf on page 224
GciIsSubclassOfClass on page 225

GciIsRemote

Determine whether the application is running linked or remotely.

Syntax

BoolType **GciIsRemote**( )

Return Value

Returns TRUE if this application is running with GciRpc (the remote procedure call version of GemBuilder). Returns FALSE if this application is running with GciLnk (that is, if GemBuilder is linked with your GemStone session).

Description

This function reports whether the current application is using the GciRpc (remote procedure call) or GciLnk (linkable) version of GemBuilder.

GciIsSubclassOf

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

Syntax

```
BoolType GciIsSubclassOf(
    OopType          testClass,
    OopType          possibleClass );
```

Arguments

testClass	The class to be checked.
possibleClass	A class or class history to check the object against.

Return Value

GciIsSubclassOf returns TRUE when *testClass* or any of its superclasses is in the class history of *possibleClass*. It returns FALSE otherwise.

Description

GciIsSubclassOf performs structural access that is equivalent to the `isSubclassOf:` method of the Smalltalk class Behavior. It compares *testClass* and *testClass*'s superclasses to see if any of them are in a given class history. When *possibleClass* is simply a class (which is typical), **GciIsSubclassOf** uses *possibleClass*'s class history. When *possibleClass* is itself a class history, **GciIsSubclassOf** uses *possibleClass* directly.

Since **GciIsSubclassOf** does consider class histories, it *can* help to support schema modification by simplifying checks on the relationship of types when they can change over time. To accomplish a similar operation without seeing the effects of class histories, use the **GciIsSubclassOfClass** function.

See Also

GciIsKindOf on page 221
GciIsKindOfClass on page 222
GciIsSubclassOfClass on page 225

GciIsSubclassOfClass

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

Syntax

BoolType **GciIsSubclassOf**(
  OopType *testClass*,
  OopType *givenClass*);

Arguments

testClass	The class to be checked.
possibleClass	A class to check *testClass* against.

Return Value

GciIsSubclassOf returns TRUE when *testClass* or any of its superclasses is *possibleClass*. It returns FALSE otherwise.

Description

GciIsSubclassOfClass performs structural access that is equivalent to the `isSubclassOf:` method of the Smalltalk class Behavior. It compares *testClass* and *testClass*'s superclasses to see if any of them are the *possibleClass*.

Since **GciIsSubclassOfClass** does *not* consider class histories, it *cannot* help to support schema modification. To accomplish a similar operation when the relationship of types can change over time, use the **GciIsSubclassOf** function.

See Also

GciIsKindOf on page 221
GciIsKindOfClass on page 222
GciIsSubclassOf on page 224

GciIvNameToIdx

Fetch the index of an instance variable name.

Syntax

int **GciIvNameToIdx**(
    OopType *aClass*,
    const char *instVarName*[ ]);

Arguments

aClass	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 functions such as **GciFetchNamedOop** or **GciStoreNamedOop**.

Example

```
BoolType nameToIdx_example(void)
{
  // Find offset for maximum in an OutOfRange exception
  OopType theClass = GciResolveSymbol("OutOfRange", OOP_NIL);
  int idx = GciIvNameToIdx(theClass, "maximum");
  if (idx < 1) {
    printf("error during GciIvNameToIdx\n");
    return false;
  }
  return idx = 12;
}
```

See Also

GciFetchNamedOop on page 172
GciFetchNamedOops on page 173

GciLoadUserActionLibrary

Load an application user action library.

Syntax

```
BoolType GciLoadUserActionLibrary(
    const char *            uaLibraryName[ ],
    BoolType                mustExist,
    void **                 libHandlePtr,
    char                    infoBuf[ ],
    int64                   infoBufSize );
```

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 4, “Writing User Actions”, starting on page 45

If a library is loaded, *libHandlePtr* is set to a value that represents the loaded library, if *libHandlePtr* is not NULL. If *mustExist* is TRUE, then an error is generated if the library can not be found. If *mustExist* is FALSE, then the library does not need to exist. In this case, TRUE is returned and *libHandlePtr* is NULL if the library does not exist and non-NULL if it exists.

See Also

GciInstallUserAction on page 217
GciInUserAction on page 220
GciUserActionShutdown on page 401

GciLogin

Start a user session.

Syntax

BoolType **GciLogin**(
const char *gemstoneUsername*[ ],
const char *gemstonePassword*[ ]);

Arguments

gemstoneUsername	The user's GemStone user name (a null-terminated string).
gemstonePassword	The user's GemStone password (a null-terminated string).

Return Value

Returns true if login succeeded, false otherwise

Description

Users 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

```
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    = "gs64stone";
  const char* HostUserId   = "";
  const char* HostPassword = "";
  const char* GemService   = "gemnetobject";
  const char* gsUserName   = "DataCurator";
  const char* gsPassword   = "swordfish";

  // 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 "FMT_OID" number %d, %s\n",
            errInfo.category, errInfo.number, errInfo.message);
      }
    }
    return success;
  }
```

See Also

GciLoginEx on page 230
GciSetNet on page 344
GciSetNetEx on page 346

GciLoginEx

Start a user session with session configuration

Syntax

```
BoolType GciLoginEx(
    const char          gemstoneUsername[ ],
    const char          gemstonePassword[ ] ,
    unsigned int        loginFlags,
    int                 haltOnErrNum);
```

Arguments

gemstoneUsername	The user's GemStone user name (a null-terminated string).
gemstonePassword	The user's GemStone password (a null-terminated string).
loginFlags	Login flags, as described below
haltOnErrNum	A legacy error number; if nonzero, specifies a value for GEM_HALT_ON_ERROR config parameter.

Return Value

Returns true if login succeeded, false otherwise

Description

This function creates a user session and its corresponding transaction workspace. The current network parameters, as specified by **GciSetNet**, are used to establish the user's GemStone session.

Other login attributes are provided by *loginFlags*. The value of *loginFlags* should be given using the following GemBuilder mnemonics:

GCI_LOGIN_PW_ENCRYPTED indicates the gemstonePassword is encrypted. When specifying this flag, you must encrypt the password using **GciEncrypt**.

GCI_LOGIN_IS_SUBORDINATE Creates the new session as a child of the current session. It will be terminated if the current session terminates. The primary use of this is for GciInterface, and is only valid for RPC sessions.

GCI_LOGIN_FULL_COMPRESSION_ENABLED the login will use full compression.

GCI_LOGIN_ERRS_USE_REF_SET sets the OOPs of objects related to errors to be put into the ReferencedSet, rather than the PureExportSet.

GCI_LOGIN_QUIET suppresses printing of banner information during login.

GCI_CLIENT_DOES_SESSION_INIT after login, the session will manually execute GciPerform(OOP_CLASS_GSCURRENT_SESSION, "initialize", NULL, 0); otherwise, the VM will automatically executes that code as part of login.

GCI_TS_CLIENT the client will use the thread-safe API. Using this flag, GciLoginEx is equivalent to GciTsLogin, from the gcits.hf (not included in this documentation).

GCI_PASSWORDLESS_LOGIN specifies login using Kerberos.

GCI_LOGIN_SOLO species to login solo, which does not require a Stone to be running.

GCI_WINDOWS_CLIENT is use on Windows client application.

If haltOnErrNum is set to a GemStone error number, and the session encounters the error associated with that error number, the session will halt.

Example

For an example, see **GciEncrypt** on page 148

See Also

GciEncrypt on page 148
GciLogin on page 228
GciSetNet (page 344)
GciSetNetEx (page 346)

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 207 for more information.)

See Also

GciGetSessionId on page 207
GciLogin on page 228
GciSetSessionId on page 348

GciLongJmp

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

Syntax

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

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 on page 150
GciPopErrJump on page 309
GciPushErrJump on page 312
GciSetErrJump on page 340
Gci_SETJMP on page 343

GciMoreTraversal

Continue object traversal, reusing a given buffer.

Syntax

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

Arguments

travBuff	A buffer in which the results of the traversal will be placed. For details, see "Traversal Buffer - GciTravBufType" on page 78.

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.

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 396 describes the organization of traversal buffers in detail.

GciMoreTraversal provides automatic byte swizzling, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void moreTraversalExample(void)
{
  // Assumes setup code has been run

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

  int totalCount = 0;
  // traverse the AllAccounts collection to 4 levels deep
  BoolType done = GciTraverseObjs(&rootObj, 1, buf, 4);
  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 on page 96
GciFindObjRep on page 199
GciObjRepSize_ on page 280
GciNbMoreTraversal on page 254
GciTraverseObjs on page 396

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 on page 93
GciBegin on page 104
GciNbBegin on page 237
GciNbCommit on page 239

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

GciBegin on page 104
GciNbAbort on page 236
GciNbCommit on page 239

GciNbClampedTrav

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

Syntax

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

Arguments

theOops	The array of OOPs for the objects to be traversed.
numOops	The number of elements in *theOops*.
travArgs	Pointer to an instance of GciClampedTravArgsSType. See **GciClampedTrav** on page 110 for documentation on the fields in *travArgs.* On return, the result is in the *travBuff* field of the *travArgs* instance of GciClampedTravArgsSType.

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

This 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 on page 110

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

This 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

GciCommit on page 117
GciNbAbort on page 236
GciNbBegin on page 237

GciNbContinue

Continue code execution in GemStone after an error (nonblocking).

Syntax

void **GciNbContinue**(
    OopType   *gsProcess*);

Arguments

gsProcess	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

GciContinue on page 122
GciNbContinueWith on page 241

GciNbContinueWith

Continue code execution in GemStone after an error (nonblocking).

Syntax

```
void GciNbContinueWith (
    OopType             process,
    OopType             replaceTopOfStack,
    int                 flags,
    GciErrSType *       error );
```

Arguments

gsProcess	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 value at 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** on page 300.
continueWithError	If not NULL, continue execution by signalling this 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

GciContinueWith on page 123
GciNbContinue on page 240

GciNbEnd

GciNbEnd_

Test the status of nonblocking call in progress for completion.

Syntax

GciNbProgressEType **GciNbEnd**(
  void ** *result*);

GciNbProgressEType **GciNbEnd_**(
  int64* *result*);

Arguments

result	The address at which **GciNbEnd** or **GciNbEnd_** should place a pointer to the result of the nonblocking call when it is complete. The actual result is that of the corresponding equivalent blocking GCI call. With **GciNbEnd**, the result may be either 4 byte or 8 byte, and the caller must be aware of whether this is 4 byte or 8 byte, and dereference result accordingly on big endian machines. **GciNbEnd_** produces result type of the corresponding blocking GCI call, typically int, BoolType or OopType, and may be accessed by normal C casting without regard to byte order of the CPU.

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, GciNbEnd_,** or **GciNbEndPoll** 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 one of these functions too often or not often enough.

Once an operation is complete, you are permitted to call **GciNbEnd** or a related function repeatedly. It returns GCI_RESULT_READY and a pointer to the same result each time, until you call a nonblocking function again. It is an error to call **GciNbEnd** or a related function before you call any nonblocking functions at all. Use the **GciCallInProgress** function to determine whether or not there is a GemBuilder call currently in progress.

Example

```
void 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

GciNbEndPoll on page 244
GciCallInProgress on page 107

GciNbEndPoll

Test the status of nonblocking call in progress for completion, with timeout.

Syntax

```
GciNbProgressEType GciNbEndPoll(
    int64*                  result,
    int                     timeoutMs);
```

Arguments

result	The address at which **GciNbEnd**Poll should place a pointer to the result of the nonblocking call when it is complete. The actual result is that of the corresponding equivalent blocking GCI call. The result will be 8 bytes, and correct on big endian machines.

Return Value

The **GciNbEndPoll** 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 **GciNbEndPoll, GciNbEnd,** or **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 **GciNbEndPoll** too often or not often enough.

GciNbEndPoll allows you to specify a timeout, and will wait for *timeoutMs* milliseconds before returning status. This avoids too-frequent polling for long-running code.

Once an operation is complete, you are permitted to call **GciNbEndPoll** repeatedly. It returns GCI_RESULT_READY and a pointer to the same result each time, until you call a nonblocking function again. It is an error to call **GciNbEndPoll** 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.

See Also

GciNbEnd on page 242
GciCallInProgress on page 107

GciNbExecute

Execute a Smalltalk expression contained in a String object (nonblocking).

Syntax

```
void GciNbExecute(
    OopType                    source,
    OopType                    symbolList );
```

Arguments

sourceStr	The OOP of a String containing a sequence of one or more Smalltalk statements to be executed.
symbolList	The OOP of a GemStone symbol list. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

The **GciNbExecute** function, unlike **GciExecute**, does not have a return value. However, when the executed operation is complete, you can access a value identical in meaning to the return value of **GciExecute** by using the argument to **GciNbEnd**.

Description

The **GciNbExecute** function is equivalent in effect to **GciExecute**. However, **GciNbExecute** permits the application to proceed with non-GemStone tasks while the Smalltalk expression is executed, and **GciExecute** does not.

See Also

GciExecute on page 151
GciNbEnd on page 242
GciNbExecuteStr on page 248
GciNbExecuteStrFetchBytes on page 249
GciNbExecuteStrFromContext on page 251
GciNbPerform on page 255

GciNbExecuteFromContextDbg_

Execute a Smalltalk expression contained in a String object (nonblocking) as if it were a message sent to an object (nonblocking).

Syntax

```
void GciNbExecuteFromContextDbg_(
    OopType              source,
    OopType              contextObject,
    OopType              symbolList,
    int                  flags,
    ushort               environmentId);
```

Arguments

source	The OOP of a String or Utf8 containing a sequence of one or more Smalltalk statements to be executed.
contextObject	The OOP of a GemStone object to use as the compilation context. If OOP_NO_CONTEXT, the code is compiled as an anonymous method in which self == nil. Otherwise, the compilation is done as if the code is an instance method of the class of *contextObj*.
symbolList	The OOP of a GemStone symbol list. 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*).
flags	Values for flags are described in gcicmn.ht,and can include: ▸ 0 (default) disables the debugger during execution. ▸ GCI_PERFORM_FLAG_ENABLE_DEBUG = 1 allows debugging, making this function behave like GciPerform. ▸ GCI_PERFORM_FLAG_DISABLE_ASYNC_EVENTS = 2 disables asynchronous events. ▸ GCI_PERFORM_FLAG_SINGLE_STEP = 3 places a single-step breakpoint at the start of the method to be performed, and then executes to hit that breakpoint.
environmentId	The compilation environment for method lookup, normally 0.

Return Value

The **GciNbExecuteFromContextDbg_** function, unlike **GciExecuteFromContext**, 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 **GciExecuteFromContext** by using the argument to **GciNbEnd**.

Description

The **GciNbExecuteFromContextDbg_** function is equivalent in effect to **GciExecuteFromContext**. However, **GciNbExecuteFromContextDbg_** permits the application to proceed with non-GemStone tasks while the Smalltalk expression is executed, and **GciExecuteFromContext** does not.

See Also

GciExecuteFromContext on page 153
GciExecuteStrFromContext on page 159
GciNbEnd on page 242
GciNbExecute on page 245
GciNbExecuteStr on page 248
GciNbExecuteStrFetchBytes on page 249
GciNbPerform on page 255

GciNbExecuteStr

Execute a Smalltalk expression contained in a C string (nonblocking).

Syntax

```
void GciNbExecuteStr(
    const char          source[ ],
    OopType             symbolList );
```

Arguments

sourceStr	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
symbolList	The OOP of a GemStone symbol list. 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

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

GciExecuteStr on page 155
GciNbEnd on page 242
GciNbExecute on page 245
GciNbExecuteStrFetchBytes on page 249
GciNbExecuteStrFromContext on page 251
GciNbPerform on page 255

GciNbExecuteStrFetchBytes

Execute a Smalltalk expression contained in a C string, returning byte-format results (nonblocking).

Syntax

void **GciNbExecuteStrFetchBytes**(

const char *	*sourceStr*,
int64	*sourceSize*,
OopType	*sourceClass*,
OopType	*contextObject*,
OopType	*symbolList*,
ByteType *	*result*,
int64	*maxResultSize*);

Arguments

sourceStr	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
sourceSize	The number of bytes in the *source*, or -1. If *sourceSize* is -1, strlen(*sourceStr*) is used.
sourceClass	The OOP of a class to use for the compilation target for the text in *sourceStr*, typically OOP_CLASS_STRING, OOP_CLASS_UNICODE7, or OOP_CLASS_Utf8.
contextObject	The OOP of a GemStone object to use as the compilation context. If OOP_NO_CONTEXT, the code is compiled as an anonymous method in which self == nil. Otherwise, the compilation is done as if the code is an instance method of the class of *contextObj*.
symbolList	The OOP of a GemStone symbol list. 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*).
result	Array containing the bytes of the result of the execution.
maxResultSize	Maximum size of *result*.

Return Value

The **GciNbExecuteStrFetchBytes** function, unlike **GciExecuteStrFetchBytes**, 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 **GciExecuteStrFetchBytes** by using the argument to **GciNbEnd**.

Description

This function sends an expression (or sequence of expressions) to GemStone for execution. The execution result, which should be a byte format object is returned in the **result* buffer.

See Also

GciExecuteStrFetchBytes on page 157
GciNbEnd on page 242
GciNbExecute on page 245
GciNbExecuteStr on page 248
GciNbExecuteStrFromContext on page 251
GciNbPerform on page 255

GciNbExecuteStrFromContext

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

Syntax

void **GciNbExecuteStrFromContext**(
    const char   *sourceStr*[ ],
    OopType   *contextObject*,
    OopType   *symbolList*);

Arguments

sourceStr	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
contextObject	The OOP of a GemStone object to use as the compilation context. If OOP_NO_CONTEXT, the code is compiled as an anonymous method in which self == nil. Otherwise, the compilation is done as if the code is an instance method of the class of *contextObj*.
symbolList	The OOP of a GemStone symbol list. 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

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

GciExecuteStrFromContext on page 159
GciNbEnd on page 242
GciNbExecute on page 245
GciNbExecuteStr on page 248
GciNbExecuteStrFetchBytes on page 249
GciNbPerform on page 255

GciNbExecuteStrTrav

First execute a Smalltalk expression contained in a C string as if it were a message sent to an object, then traverse the result of the execution (nonblocking).

Syntax

void **GciNbExecuteStrTrav**(
    const char *sourceStr*[ ],
    OopType *contextObject*,
    OopType *symbolList*,
    GciClampedTravArgsSType **travArgs*);

Arguments

sourceStr	A null-terminated string containing a sequence of one or more Smalltalk statements to be executed.
contextObject	The OOP of a GemStone object to use as the compilation context. If OOP_NO_CONTEXT or OOP_ILLEGAL, the code is compiled as an anonymous method in which self == nil. Otherwise, the compilation is done as if the code is an instance method of the class of *contextObj*.
symbolList	The OOP of a GemStone symbol list. 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*).
travArgs	Pointer to an instance of GciClampedTravArgsSType. See **GciExecuteStrTrav** on page 161 for field definitions.

Return Value

The **GciNbExecuteStrTrav** function, unlike **GciExecuteStrTrav**, does not have a return value. However, when the traversal operation is complete, you can access a value identical in meaning to the return value of **GciExecuteStrTrav** by using the argument to **GciNbEnd**.

Description

The **GciNbExecuteStrTrav** function is equivalent in effect to **GciExecuteStrTrav**. However, **GciNbExecuteStrTrav** permits the application to proceed with non-GemStone tasks while the traversal is completed, and **GciExecuteStrTrav** does not.

See Also

GciExecuteStrTrav on page 161 includes information details on traversal.
GciNbEnd on page 242
GciNbExecute on page 245
GciPerformTraverse on page 304

GciNbLoginEx

Start a user session with session configuration (nonblocking)

Syntax

```
BoolType GciNbLoginEx(
    const char          gemstoneUsername[ ],
    const char          gemstonePassword[ ] ,
    unsigned int        loginFlags,
    int                 haltOnErrNum);
```

Arguments

gemstoneUsername	The user's GemStone user name (a null-terminated string).
gemstonePassword	The user's GemStone password (a null-terminated string).
loginFlags	Login flags, as described below
haltOnErrNum	A legacy error number; if nonzero, specifies a value for GEM_HALT_ON_ERROR config parameter.

Return Value

Returns true if login succeeded, false otherwise

Description

This function creates a user session and its corresponding transaction workspace. The current network parameters , as specified by **GciSetNet**, are used to establish the new GemStone session.

Other login attributes are provided by *loginFlags*. The value of *loginFlags* should be given using the GemBuilder mnemonics descibed under **GciLoginEx** on page 230.

If *haltOnErrNum* is set to a GemStone error number, and the session encounters the error associated with that error number, the new session will halt.

Once this call is invoked, the current session is set to the not-yet-logged in session. Calls to **GciNbEnd** apply to the not-yet-logged in session.

See Also

GciLoginEx on page 230
GciNbEnd on page 242

GciNbMoreTraversal

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

Syntax

```
void GciNbMoreTraversal(
    GciTravBufType *            travBuff);
```

Arguments

travBuff	A buffer in which the results of the traversal will be placed. For details, see “Traversal Buffer - GciTravBufType” on page 78.

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, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see “Byte-Swizzling of Binary Floating-Point Values” on page 26.

See Also

GciMoreTraversal on page 234
GciNbTraverseObjs on page 264
GciObjRepSize_ on page 280
GciTraverseObjs on page 396

GciNbPerform

Send a message to a GemStone object (nonblocking).

Syntax

void **GciNbPerform**(

OopType	*receiver*,
const char	*selectorStr*[ ],
const OopType	*args*[ ],
int	*numArgs*);

Arguments

receiver	The OOP of the receiver of the message.
selectorStr	A string that defines the message selector. For keyword selectors, all keywords are concatenated in the string. (For example, `at:put:`).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.

Return Value

The **GciNbPerform** function, unlike **GciPerform**, does not have a return value. However, when the performed operation is complete, you can access a value identical in meaning to the return value of **GciPerform** by using the argument to **GciNbEnd**.

Description

The **GciNbPerform** function is equivalent in effect to **GciPerform**. However, **GciNbPerform** permits the application to proceed with non-GemStone tasks while the message is executed, and **GciPerform** does not.

See Also

GciNbEnd on page 242
GciNbExecute on page 245
GciNbPerformNoDebug on page 257
GciNbPerformTrav on page 258
GciPerform on page 296

GciNbPerformFetchBytes

Send a message to a GemStone object, returning byte results (nonblocking).

Syntax

```
void GciNbPerformFetchBytes(
    OopType               receiver,
    const char            selector[ ],
    const OopType         args[ ],
    int                   numArgs,
    ByteType *            result,
    ssize_t               maxResultSize);
```

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.
result	Array containing the bytes of the result of the execution.
maxResultSize	Maximum size of *result*.

Return Value

The **GciNbPerformFetchBytes** function, unlike **GciPerformFetchBytes**, 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 **GciPerformFetchBytes** by using the argument to **GciNbEnd**.

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), and puts the result, which should be a byte format object, in the *result buffer.

See Also

GciNbExecuteStrFetchBytes on page 249
GciNbPerform on page 255

GciNbPerformNoDebug

Send a message to a GemStone object, and temporarily disable debugging (nonblocking).

Syntax

void **GciNbPerformNoDebug**(

OopType	*receiver*,
const char	*selector*[ ],
const OopType	*args*[ ],
int	*numArgs*,
int	*flags*);

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. Use one or more of the GemBuilder mnemonics described under **GciPerformNoDebug** on page 300.

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

GciNbEnd on page 242
GciNbExecute on page 245
GciNbPerform on page 255
GciPerformNoDebug on page 300

GciNbPerformTrav

First send a message to a GemStone object, then traverse the result of the message (nonblocking).

Syntax

```
void GciNbPerformTrav(
    OopType                       receiver,
    const char *                  selector,
    const OopType *               args,
    int                           numArgs,
    GciClampedTravArgsSType *travArgs );
```

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.
travArgs	Pointer to an instance of GciClampedTravArgsSType. See **GciClampedTrav** on page 110 for documentation of the fields in *travArgs*. On return, the results are in the first object in the resulting *travBuffs* field.

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 on page 303
GciNbPerform on page 255
GciClampedTrav on page 110

GciNbStoreTrav

Stores multiple traversal buffer values in objects (nonblocking).

Syntax

```
void GciNbStoreTrav(
    GciTravBufType *        travBuff,
    int                     behaviorFlag );
```

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, defined 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, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see “Byte-Swizzling of Binary Floating-Point Values” on page 26.

See Also

GciNbEnd on page 242
GciNbMoreTraversal on page 254
GciNbStoreTrav on page 259
GciNbStoreTravDoTrav_ on page 261
GciNbTraverseObjs on page 264
GciStoreTrav on page 376

GciNbStoreTravDo_

Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object (non-blocking).

Syntax

void **GciNbStoreTravDo_**(
    GciStoreTravDoArgsSType * *storeTravArgs*);

Arguments

storeTravArgs	An instance of **GciStoreTravDoArgsSType**. For details, refer to the discussion of **GciStoreTravDo_** on page 379

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

GciNbEnd on page 242
GciNbMoreTraversal on page 254
GciNbStoreTrav on page 259
GciNbStoreTravDoTrav_ on page 261
GciStoreTravDo_ on page 379

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).

Syntax

void **GciNbStoreTravDoTrav**_(
    GciStoreTravDoArgsSType * *storeTravArgs*,
    GciClampedTravArgsSType **clampedTravArgs*);

Arguments

storeTravArgs	An instance of **GciStoreTravDoArgsSType**. For details, refer to the discussion of **GciStoreTravDo_** on page 379.
clampedTravArgs	An instance of **GciClampedTravArgsSType**. For details, refer to the discussion of **GciClampedTrav** on page 110.

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

GciNbEnd on page 242
GciNbMoreTraversal on page 254
GciNbStoreTrav on page 259
GciNbStoreTravDo_ on page 260
GciNbStoreTravDoTravRefs_ on page 262
GciStoreTravDoTrav_ on page 381

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).

Syntax

```
void GciNbStoreTravDoTravRefs_(
    const OopType *              oopsNoLongerReplicated,
    int                          numNotReplicated,
    const OopType *              oopsGcedOnClient,
    int                          numGced,
    GciStoreTravDoArgsSType *    storeTravArgs,
    GciClampedTravArgsSType *    clampedTravArgs );
```

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*.
storeTravArgs	An instance of **GciStoreTravDoArgsSType**. For details, refer to the discussion of **GciStoreTravDo_** on page 379.
clampedTravArgs	An instance of **GciClampedTravArgsSType**. For details, see the discussion of **GciClampedTrav** on page 110.

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 **GciNbEnd**

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

GciNbEnd on page 242
GciNbMoreTraversal on page 254
GciNbStoreTrav on page 259
GciNbStoreTravDo_ on page 260
GciNbStoreTravDoTrav_ on page 261
GciStoreTravDoTravRefs_ on page 382

GciNbTraverseObjs

Traverse an array of GemStone objects (nonblocking).

Syntax

```
void GciNbTraverseObjs(
    const OopType           theOops[ ],
    int                     numOops,
    GciTravBufType *        travBuff,
    int                     level );
```

Arguments

theOops	An array of OOPs representing the objects to be traversed.
numOops	The number of elements in *theOops*.
travBuff	A buffer in which the results of the traversal will be placed. For details, see "Traversal Buffer - GciTravBufType" on page 78.
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.

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, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26.

See Also

GciNbEnd on page 242
GciNbMoreTraversal on page 254
GciNbStoreTrav on page 259
GciNbStoreTravDo_ on page 260
GciNbStoreTravDoTrav_ on page 261
GciTraverseObjs on page 396

GciNewByteObj

Create and initialize a new byte object.

Syntax

```
OopType GciNewByteObj(
    OopType                 aClass,
    const ByteType *        value,
    int64                   valueSize );
```

Arguments

aClass	The OOP of the class of which the new object is an instance. *aClass* must be a class whose format is BYTE.
cString	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.

See Also

GciNewCharObj on page 266
GciNewOop on page 268
GciNewString on page 273

GciNewCharObj

Create and initialize a new character object.

Syntax

OopType **GciNewCharObj**(
    OopType   *aClass*,
    const char *   *cString*);

Arguments

aClass	The OOP of the class of which the new object is an instance. *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.

See Also

GciNewByteObj on page 265
GciNewOop on page 268
GciNewString on page 273

GciNewDateTime

Create and initialize a new date-time object.

Syntax

```
OopType GciNewDateTime(
    OopType                    theClass,
    const GciDateTimeSType *   timeVal );
```

Arguments

theClass	The class of the object to be created, which must be OOP_CLASS_DATE_TIME or a subclass.
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. For details on GciDateTimeSType, see “Date/Time Structure- GciDateTimeSType” on page 68.

GciNewOop

Create a new GemStone object.

Syntax

OopType **GciNewOop**(
    OopType *aClass*);

Arguments

aClass	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. See `$GEMSTONE/include/gcioop.ht` for the C constants that are defined for GemStone 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

```
OopType newOop_example(void)
{
  // create a new instance of String
  OopType result = GciNewOop(OOP_CLASS_STRING);
  return result;
}
```

See Also

GciNewOop on page 268
GciNewString on page 273
GciNewSymbol on page 274
GciNewUtf8String on page 275

GciNewOops

Create multiple new GemStone objects.

Syntax

void **GciNewOops**(
    int   *numOops*,
    const OopType   *classes*[ ],
    const int64   *idxSize*[ ],
    OopType   *result*[ ]);

Arguments

numOops	The number of new objects to be created.
classes	An Array of the OOPs of the classes of each new object, 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. See `$GEMSTONE/include/gcioop.ht` for the C constants that are defined for GemStone kernel classes.
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	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 *classes* 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 *classes* contains the OOP of a class with special implementation (such as Boolean), then the corresponding element in *result* is OOP_NIL. See `$GEMSTONE/include/gcioop.ht` for the C constants that are defined for GemStone kernel classes.

GciNewOops generates an error when either of the following conditions is TRUE for any object:

- *idxSize* `< 0`
- `(`*idxSize* `+ number_of_named_instance_variables) > maxSmallInt`

Example

```
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"\n",
     newObjs[0], newObjs[1], newObjs[2]);
  }
}
```

See Also

GciNewOop on page 268
GciNewString on page 273
GciNewSymbol on page 274
GciNewUtf8String on page 275

GciNewOopUsingObjRep

Create a new GemStone object from an existing object report.

Syntax

void **GciNewOopUsingObjRep**(
    GciObjRepSType *   *anObjectReport*);

Arguments

anObjectReport	Pointer to an object report, which will be modified to 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 "Object Report Structure - GciObjRepSType" on page 71.

GciNewOopUsingObjRep provides automatic byte swizzling for Float and SmallFloat objects. (For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26.)

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

```
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

GciNewOops on page 269
GciNewString on page 273
GciNewSymbol on page 274
GciNewUtf8String on page 275
GciNewString on page 273

GciNewString

Create a new String object from a C character string.

Syntax

```
OopType GciNewString(
    const char *                    cString );
```

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*);

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.

GciNewUtf8String

GciNewUtf8String_

Create a new Unicode string object from a UTF-8 encoded C character string.

Syntax

```
OopType GciNewUtf8String(
    const char *        unicodeCString,
    BoolType            utf8OrUnicode );

OopType GciNewUtf8String_(
    const char *        unicodeCString,
    size_t              nBytes,
    BoolType            utf8OrUnicode );
```

Arguments

unicodeCString	Pointer to a null-terminated UTF-8 encoded character string.
nBytes	The length in bytes of the UTF-8 encoded character string.
utf8OrUnicode	Boolean indicating whether to create an instance of Utf8 or of a Unicode class. If *utf8OrUnicode* = 0, return an instance of Utf8. If *utf8OrUnicode*=1, return an instance of Unicode7, Unicode16, or Unicode32, the minimal character size required to represent *unicodeCString.*

Return Value

The OOP of the newly created object.

Description

Returns a new instance of Utf8, Unicode7, Unicode16, or Unicode32, with the value that the UTF-8 encoded *unicodeCString* points to.

With the **GciNewUtf8String_** option, you provide the length of the C string; with **GciNewUtf8String**, `strlen()` is used to compute the size.

GciNscIncludesOop

Determines whether the given OOP is present in the specified unordered collection.

Syntax

```
BoolType GciNscIncludesOop(
    OopType         theNsc,
    OopType         theOop );
```

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

```
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 on page 94
GciAddOopsToNsc on page 95
GciRemoveOopFromNsc on page 323
GciRemoveOopsFromNsc on page 324

GciObjExists

Determine whether or not a GemStone object exists.

Syntax

```
BoolType GciObjExists(
    OopType                  theObject );
```

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 );
```

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                         theOop );
```

Arguments

theOop	The OOP of an object.

Return Value

GciObjIsCommitted returns TRUE if the object *theOop* is committed, FALSE otherwise.

Description

The **GciObjIsCommitted** function determines if the given object is committed or not.

See Also

GciObjExists on page 277

GciObjRepSize_

Find the number of bytes in an object report.

Syntax

size_t **GciObjRepSize_**(*anObjectReport*)
const GciObjRepHdrSType * *anObjectReport*;

Arguments

anObjectReport	A pointer to an object report returned by **GciFindObjRep**.

Return Value

Returns the size of the specified object report.

Description

This function calculates the number of bytes in an object report. Before your application allocates memory for a copy of the object report, it can call this function to obtain the size of the report.

Example

See “Executing the examples” on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void objRepSize_example(void)
{
  OopType rootObj = GciResolveSymbol("AllAccounts", OOP_NIL);
  GciTravBufType *buf = GciAllocTravBuf(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 on page 199
GciMoreTraversal on page 234
GciTraverseObjs on page 396

GCI_OOP_IS_BOOL

(MACRO) Determine whether or not a GemStone object represents a GemStone Smalltalk Boolean.

Syntax

BoolType **GCI_OOP_IS_BOOL**(*theOop*)

Arguments

theOop	The OOP of the object to test.

Return 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 object.

See Also

GCI_BOOL_TO_OOP on page 105

GCI_OOP_IS_CHAR

(MACRO) Determine whether or not a GemStone object represents a Character.

Syntax

BoolType **GCI_OOP_IS_SMALL_CHAR**(*theOop*)

Arguments

theOop	The OOP of the object to test.

Return Value

A C Boolean value. Returns TRUE if the object represents a GemStone Character, FALSE otherwise.

Description

This macro tests to see if *theOop* represents a Character.

See Also

GCI_OOP_IS_BOOL on page 281
GCI_OOP_IS_SMALL_FRACTION on page 283
GCI_OOP_IS_SMALL_INT on page 284
GCI_OOP_IS_SPECIAL on page 285

GCI_OOP_IS_SMALL_FRACTION

(MACRO) Determine whether or not a GemStone object represents a SmallFraction.

Syntax

BoolType **GCI_OOP_IS_SMALL_FRACTION**(*theOop*)

Arguments

theOop	The OOP of the object to test.

Return Value

A C Boolean value. Returns TRUE if the object represents a SmallFraction, FALSE otherwise.

Description

This macro tests to see if *theOop* represents a SmallFraction.

See Also

GCI_OOP_IS_BOOL on page 281
GCI_OOP_IS_CHAR on page 282
GCI_OOP_IS_SMALL_INT on page 284
GCI_OOP_IS_SPECIAL on page 285

GCI_OOP_IS_SMALL_INT

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

Syntax

BoolType **GCI_OOP_IS_SMALL_INT**(*theOop*)

Arguments

theOop	The OOP of the object to test.

Return 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 on page 281
GCI_OOP_IS_CHAR on page 282
GCI_OOP_IS_SMALL_FRACTION on page 283
GCI_OOP_IS_SPECIAL on page 285

GCI_OOP_IS_SPECIAL

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

Syntax

BoolType **GCI_OOP_IS_SPECIAL**(*theOop*)

Arguments.

theOop	The OOP of the object to test.

Return 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. GemStone specials are objects in which the OOP encodes the value, including Boolean, Character, SmallInteger, SmallDouble, and SmallFraction.

See Also

GCI_OOP_IS_BOOL on page 281
GCI_OOP_IS_CHAR on page 282
GCI_OOP_IS_SMALL_FRACTION on page 283
GCI_OOP_IS_SMALL_INT on page 284

GciOopToBool

Convert a Boolean object to a C Boolean value.

Syntax

BoolType **GciOopToBool**(
    OopType *theOop*);

Arguments

theOop	The OOP of the Boolean object to be translated into a C Boolean value.

Return Value

Returns the C Boolean value that corresponds to the GemStone object. In case of error, this function returns FALSE.

Description

This function translates a GemStone Boolean object into the equivalent C Boolean value.

Example

```
BoolType 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 on page 105

GCI_OOP_TO_BOOL

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

Syntax

GCI_OOP_TO_BOOL(*theOop*)

Arguments

theOop	The OOP of the Boolean object to be translated into a C Boolean value.

Return 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** on page 286. For the definition of GCI_OOP_TO_BOOL, see `$GEMSTONE/include/gcicmn.ht`

See Also

GCI_BOOL_TO_OOP on page 105

GciOopToChar16

Convert a Character object to a 16-bit C character value.

Syntax

```
unsigned int GciOopToChar16(
    OopType                    theOop );
```

Arguments

theOop	The OOP of the Character 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 on page 289
GciOopToChr on page 290

GciOopToChar32

Convert a Character object to a 32-bit C character value.

Syntax

unsigned int **GciOopToChar32**(
    OopType *theOop*);

Arguments

theOop	The OOP of the Character 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 on page 288
GciOopToChr on page 290

GciOopToChr

Convert a Character object to a C character value.

Syntax

char **GciOopToChr**(
    OopType *theOop*);

Arguments

theOop 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.

Attempting to convert a GemStone Character that is outside the range of C characters will result in an error.

Description

This function translates a GemStone Character object into the equivalent C character value.

Example

```
BoolType oopToChr_example(void)
{
  OopType theOop = GCI_CHR_TO_OOP('a');
  int aChar = GciOopToChr(theOop);
  if ( aChar == 0 ) {
    printf("GciOopToChr failed\n");
    return false;
    }
    return (aChar == (int) 'a');
}
```

See Also

GCI_CHR_TO_OOP on page 109
GciOopToChar16 on page 288
GciOopToChar32 on page 289
GCI_OOP_TO_CHR on page 291

GCI_OOP_TO_CHR

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

Syntax

GCI_OOP_TO_CHR(*theOop*)

Arguments

theOop	The OOP of the Character object to be translated into a C character value.

Return 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 on page 288
GciOopToChr on page 290

GciOopToFlt

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

Syntax

double **GciOopToFlt**(
    OopType *theObject*);

Arguments

theObject The OOP of the specified SmallDouble, Float, or SmallFloat object.

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 on page 201
Gci_doubleToSmallDouble on page 143

GciOopToI32

GciOopToI32_

Convert a GemStone object to a C 32-bit integer value.

Syntax

```
int GciOopToI32(
      OopType                    theObject );

int GciOopToI32_(
      OopType                    theObject,
      BoolType *                 error );
```

Arguments

theObject	The OOP of the specified Integer object.
error	TRUE if *theObject* does not fit in the result type or is not an Integer, otherwise unchanged.

Return Value

The **GciOopToI32** and **GciOopToI32_** functions return the C 32-bit integer value that is equivalent to the value of *theObject*.

Description

The **GciOopToI32** and **GciOopToI32_** functions translate 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; **GciOopToI32_** does not generate an error, but places a boolean in the error argument.

See Also

GciOopToI64 on page 295

GciOopToI64

GciOopToI64_

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

Syntax

```
int64 GciOopToI64(
      OopType                  theObject );

int64 GciOopToI64_(
      OopType                  theObject,
      BoolType *               error );
```

Arguments

theObject	The OOP of the Integer object.
error	TRUE if *theObject* does not fit in the result type or is not an Integer, otherwise unchanged.

Return Value

The **GciOopToI64** and **GciOopToI64_** functions return the C int64_t value that is equivalent to the value of *theObject*.

Description

The **GciOopToI64** and **GciOopToI64_** functions translate a GemStone object into the equivalent C 64-bit integer value.

The object identified by *theObject* must be a SmallInteger or a LargeInteger. If the object is not one of these kinds, **GciOopToI64** generates an error; **GciOopToI64_** places a boolean in the argument error.

See Also

GciOopToI32 on page 294

GciPerform

Send a message to a GemStone object.

Syntax

```
OopType GciPerform(
    OopType             receiver,
    const char          selector[ ],
    const OopType       args[ ],
    int                 numArgs );
```

Arguments

receiver	The OOP of the receiver of the message.
selector	A string that defines the message selector. For keyword selectors, all keywords are concatenated in the string. (For example, `at:put:`).
args	An array of OOPs. Each element in the array corresponds to an argument for the message. If there are no message arguments, use a dummy OOP here.
numArgs	The number of arguments to the message. For unary selectors (messages with no arguments), *numArgs* is zero.

Return Value

Returns the OOP of the result of Smalltalk execution. In case of error, this function returns OOP_NIL.

Description

This function sends a message (that is, the selector along with any keyword arguments and their corresponding values) to the specified receiver (an object in the GemStone database). Because **GciPerform** calls the virtual machine, you can issue a soft break while this function is executing. For more information, see "Interrupting GemStone Execution" on page 30.

Example

```
BoolType perform_example()
{
   GciErrSType errInfo;
   OopType userGlobals = GciResolveSymbol("UserGlobals", OOP_NIL);

   OopType argList[2];
   argList[0] = GciNewSymbol("myNumber");
   argList[1] = GciI32ToOop(55);

   OopType result = GciPerform(userGlobals, "at:put:", argList, 2);
   if (result == OOP_NIL) {
      if (GciErr(&errInfo)) {
            printf("GciPerform failed; error %d, %s\n", errInfo.number,
                  errInfo.message);
            return false;
         }
      }
      BoolType err = false;
      int64 val = GciOopToI64_(result, &err);
      if (err) {
         printf("GciOopToI64 failed\n");
         return false;
      }
      return (val = 55);
   }
```

See Also

GciExecuteStr on page 155
GciNbPerform on page 255
GciPerformNoDebug on page 300
GciPerformSymDbg on page 302
GciPerformTrav on page 303
GciPerformTraverse on page 304

GciPerformFetchBytes

Send a message to a GemStone object, returning byte results.

Syntax

```
OopType GciPerformFetchBytes(
    OopType              receiver,
    const char           selector[ ],
    const OopType        args[ ],
    int                  numArgs,
    ByteType *           result,
    ssize_t              maxResultSize);
```

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.
result	Array containing the bytes of the result of the execution.
maxResultSize	Maximum size of *result*.

Return Value

Returns number of bytes returned in *result, or -1 if an error is available to be fetched with GciErr.

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), and puts the result, which should be a byte format object, in the *result buffer.

Because **GciPerformFetchBytes** calls the virtual machine, you can issue a soft break while this function is executing.

See Also

GciExecuteStrFetchBytes on page 157
GciNbPerformFetchBytes on page 256
GciPerform on page 296
GciPerformTrav on page 303
GciPerformTraverse on page 304

GciPerformNoDebug

Send a message to a GemStone object, and temporarily disable debugging.

Syntax

OopType **GciPerformNoDebug**(

OopType	*receiver*,
const char	*selector*[ ],
const OopType	*args*[ ],
int	*numArgs*,
int	*flags*);

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. Use one or more of the GemBuilder mnemonics described below.

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.

Values for flags are described in gcicmn.ht,and can include:

- 0 (default) disables the debugger during execution.
- GCI_PERFORM_FLAG_ENABLE_DEBUG = 1 allows debugging, making this function behave like GciPerform.
- GCI_PERFORM_FLAG_DISABLE_ASYNC_EVENTS = 2 disables asynchronous events.
- GCI_PERFORM_FLAG_SINGLE_STEP = 3 places a single-step breakpoint at the start of the method to be performed, and then executes to hit that breakpoint.

See Also

GciExecuteStr on page 155
GciNbPerformNoDebug on page 257
GciPerform on page 296
GciPerformSymDbg on page 302
GciPerformTrav on page 303
GciPerformTraverse on page 304

GciPerformSymDbg

Send a message to a GemStone object, using a String object as a selector.

Syntax

OopType **GciPerformSymDbg**(

OopType	*receiver,*
OopType	*selector*,
const OopType	*args*[ ],
int	*numArgs,*
int	*flags*);

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.

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

GciExecute on page 151
GciPerform on page 296

GciPerformTrav

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

Syntax

```
BoolType GciPerformTrav(
    OopType                    receiver,
    const char *               selector,
    const OopType *            args,
    int                        numArgs,
    GciClampedTravArgsSType *travArgs );
```

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 **GciClampedTrav** on page 110 for documentation of the fields in *travArgs*. 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

GciClampedTrav on page 110
GciNbPerformTrav on page 258
GciPerform on page 296
GciPerformTraverse on page 304

GciPerformTraverse

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

Syntax

```
BoolType GciPerformTraverse(
    OopType              receiver,
    const char           selector[ ],
    const OopType        args[ ],
    int                  numArgs,
    GciTravBufType *     travBuff,
    int                  level );
```

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.
travBuff	A traversal buffer in which the results of the traversal are placed.
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.

Return Value

Returns FALSE if the traversal is not yet completed, but further traversal would cause the *travBuffSize* to be exceeded. If the *travBuffSize* is reached before the traversal is complete, you can then call **GciMoreTraversal** to proceed from the point where *travBuffSize* was exceeded.

Returns TRUE if there are no more objects to be returned by subsequent calls to **GciMoreTraversal**.

Description

This function does a perform, then traverses the result of the perform.

GciPerformTraverse provides automatic byte swizzling, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see “Byte-Swizzling of Binary Floating-Point Values” on page 26.

Example

The following two functions are equivalent:

```
BoolType performTraverse_example1(void)
{
  OopType receiver = GciResolveSymbol("AllUsers", OOP_NIL);
  OopType arg = GciI32ToOop(1);
  GciTravBufType *buf = GciAllocTravBuf(80000);

  BoolType atEnd = GciPerformTraverse(receiver, "at:", &arg, 1, buf, 2);
  return atEnd;
}
BoolType performTraverse_example2(void)
{
  OopType receiver = GciResolveSymbol("AllUsers", OOP_NIL);
  OopType arg = GciI32ToOop(1);
  OopType obj = GciPerform(receiver, "at:", &arg, 1);

  GciTravBufType *buf = GciAllocTravBuf(80000);
  BoolType atEnd = GciTraverseObjs(&obj, 1, buf, 2);
  return atEnd;
}
```

See Also

GciClampedTrav on page 110
GciMoreTraversal on page 234
GciNbPerformTrav on page 258
GciPerform on page 296
GciPerformTrav on page 303
GciTraverseObjs on page 396

GciPointerToByteArray

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

Syntax

```
OopType GciPointerToByteArray(
    void *                       pointer );
```

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 on page 106

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 on page 146
GciErr on page 150

GciPollSocketForRead

Wait for the specified socket to be read-ready.

Syntax

int **GciPollSocketForRead**(
    int *socketFd*,
    int *timeoutMs*);

Arguments

socketFd	A file descriptor for a socket.
timeoutMs	Milliseconds to wait for a response before timing out and returning 0.

Return Value

This function returns 0 if timed out, 1 if socket is ready for read, and an int < 0 if an error occurred. The result in this cases is the negated errno value.

Description

Wait *timeoutMs* milliseconds for the specified socket to be read-ready or to have an error.

This function retrys the poll on EINTR , even if a SIGTERM was received.

This is a thread safe function, and has no relationship to the current GCI session .

GciPopErrJump

Discard a previously saved error jump buffer.

Syntax

void **GciPopErrJump**(
    GciJmpBufSType * *jumpBuffer*);

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

```
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 on page 150
GciPushErrJump on page 312
GciSetErrJump (page 340)
GciLongJmp on page 233

GciProcessDeferredUpdates_

Process deferred updates to objects that do not allow direct structural update.

Syntax

int64 **GciProcessDeferredUpdates_**( )

Return Value

Returns the number of objects that had deferred updates.

Description

This function processes updates to instances of classes that have the noStructuralUpdate bit set, including AbstractDictionary, Bag, Set, and their subclasses. After operations that modify an instance of once of these classes, either **GciProcessDeferredUpdates_** must be called, or the final **GciStoreTrav** must have GCI_STORE_TRAV_FINISH_UPDATES set.

The following GemBuilder calls operate on instances whose classes have noStructuralUpdate set: **GciCreateOopObj**, **GciStoreTrav, GciStore...Oops, GciAdd...Oops, GciReplace...Oops**. Behavior of other GemBuilder update calls on such instances is undefined.

An attempt to commit automatically executes a deferred update.

Executing a deferred update before all forward references are resolved can produce errors that require the application to recover by doing a **GciAbort** or **GciLogout**.

An OOP buffer used to update the varying portion of an object with noStructuralUpdate must contain the OOPs to be added to the varying portion of the object, with two exceptions:

- If the object is a kind of KeyValueDictionary that does not store Associations, the buffer must contain (key, value) pairs.
- If the object is a kind of AbstractDictionary that stores Associations or (key, Association) pairs, the value buffer must contain Associations.

See Also

GciStoreTrav on page 376

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. This will always return 3 in 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.

Currently-defined integers are:
 1 — 32-bit GemStone/S
 2 — GemStone/S 2G (discontinued product)
 3 — GemStone/S 64 Bit

Any 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 on page 403

GciPushErrJump

Associate GemBuilder error handling with a jump buffer by pushing a jump buffer onto the stack.

Syntax

void **GciPushErrJump**(
    GciJmpBufSType *   *jumpBuffer*);

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 **GciErr** to poll for GBC errors.

Example

For an example of how **GciPushErrJump** is used, see **GciPopErrJump** on page 309.

See Also

GciErr on page 150
GciLongJmp on page 233
GciPopErrJump on page 309
GciSetErrJump on page 340
Gci_SETJMP on page 343

GciRaiseException

Signal an error, synchronously, within a user action.

Syntax

void **GciRaiseException**(
    const GciErrSType * *error*);

Arguments

error 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. In order to signal an error on the Smalltalk client, this function must be invoked.

This function has no effect when executed outside of a user action.

Example

```
OopType res = GciNewOop(anOopClass);
GciErrSType theErr;
// the term res==OOP_NIL is a performance optimization
if (res == OOP_NIL && GciErr(&theErr)) {
   GciRaiseException(&theErr);
   }
```

GciReadSharedCounter

Lock and fetch the value of a shared counter.

Syntax

BoolType **GciReadSharedCounter**(
    int *counterIdx,*
    int64_t * *value*);

Arguments

counterIdx	The offset into the shared counters array of the value to fetch.
value	Pointer to a value for 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

GciDecSharedCounter on page 135
GciFetchNumSharedCounters on page 178
GciFetchSharedCounterValuesNoLock on page 189
GciIncSharedCounter on page 213
GciReadSharedCounterNoLock on page 315
GciSetSharedCounter on page 349

GciReadSharedCounterNoLock

Fetch the value of a shared counter without locking it.

Syntax

```
BoolType GciReadSharedCounterNoLock(
    int                     counterIdx,
    int64_t *               value);
```

Arguments

counterIdx	The offset into the shared counters array of the value to fetch.
value	Pointer to a value for 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

GciDecSharedCounter on page 135
GciFetchNumSharedCounters on page 178
GciFetchSharedCounterValuesNoLock on page 189
GciIncSharedCounter on page 213
GciReadSharedCounter on page 314
GciSetSharedCounter on page 349

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 will not need any of them again.

See Also

"Garbage Collection" on page 18
GciReleaseAllGlobalOops on page 316
GciReleaseAllOops on page 317
GciReleaseGlobalOops on page 319
GciReleaseOops on page 320
GciSaveObjs on page 336

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 will not need any of them again.

See Also

“Garbage Collection” on page 18
GciReleaseAllGlobalOops on page 316
GciReleaseGlobalOops on page 319
GciReleaseOops on page 320
GciSaveGlobalObjs on page 335
GciSaveObjs on page 336

GciReleaseAllTrackedOops

Clear the GciTrackedObjs set, making all tracked OOPs eligible for garbage collection.

Syntax

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 will not need them again.

See Also

GciReleaseAllGlobalOops on page 316
GciReleaseOops on page 320
GciReleaseTrackedOops on page 322
GciSaveAndTrackObjs on page 334

GciReleaseGlobalOops

Remove an array of GemStone OOPs from the PureExportSet, making them eligible for garbage collection.

Syntax

void **GciReleaseGlobalOops**(
    const OopType *theOops*[ ],
    int *numOops*);

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 will not need them again.

See Also

"Garbage Collection" on page 18
GciReleaseAllGlobalOops on page 316
GciReleaseOops on page 320
GciSaveGlobalObjs on page 335

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

```
void GciReleaseOops(
    const OopType            theOops[ ],
    int                      numOops );
```

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 have other references, or will not need them again.

Example

```
void releaseOops_example(void)
{
   OopType oArray = GciExecuteStr("UserGlobals at: #myArray put: (Array
            new: 5)", OOP_NIL);

   OopType ivs[3];
   ivs[0] = GciNewString("This is a string");
   ivs[1] = GciI32ToOop(5699); // a SmallInteger; don't need to release
   ivs[2] = GciFltToOop(9.0e6); // a Float or SmallDouble
   GciStoreOops(oArray, 1, ivs, 3);

   // release newly created objects,so that if oArray is removed from
   // UserGlobals by other application code, these new objects can
```

```
      // be garbage collected.
      OopType releaseBuf[3];
      releaseBuf[0] = ivs[0];  // a String
      releaseBuf[1] = ivs[2];  // might be a Float
      releaseBuf[2] = oArray;  // the array instance
      GciReleaseOops(releaseBuf, 3);
      GciErrSType err;
      if (GciErr(&err)) {
         printf("Error %d, %s\n", err.number, err.message); }
   }
```

See Also

"Garbage Collection" on page 18
GciReleaseAllGlobalOops on page 316
GciReleaseAllOops on page 317
GciSaveGlobalObjs on page 335
GciSaveObjs on page 336

GciReleaseTrackedOops

Remove an array of OOPs from the GciTrackedObjs set, making them eligible for garbage collection.

Syntax

```
void GciReleaseTrackedOops(
    const OopType             theOops[ ],
    int                       numOops );
```

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 will not need them again.

See Also

GciReleaseAllTrackedOops on page 318
GciSaveAndTrackObjs on page 334
GciTrackedObjsInit on page 395

GciRemoveOopFromNsc

Remove an OOP from an NSC.

Syntax

```
BoolType GciRemoveOopFromNsc(
    OopType                theNsc,
    OopType                theOop );
```

Arguments

theNsc	The OOP of the NSC from which to remove the OOP.
theOops	The OOP of the object to be removed from the NSC.

Return Value

Returns FALSE if *theOop* was not present in the NSC. Returns TRUE if *theOop* was present.

Description

This function removes an OOP from the unordered variables of an NSC, using structural access.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
BoolType removeOopFromNsc_example(void)
{
   OopType anAccount = GciExecuteStr("AllAccounts detect: [:ea | ea
               id=1073", OOP_NIL);
   OopType aColl = GciResolveSymbol("AllAccounts", OOP_NIL);

   BoolType wasPresent = GciRemoveOopFromNsc(aColl, anAccount);

   /* release because it is the result from an execute */
   GciReleaseOops(&anAccount, 1);
   return wasPresent;
}
```

See Also

GciAddOopToNsc on page 94
GciAddOopsToNsc on page 95
GciNscIncludesOop on page 276
GciRemoveOopsFromNsc on page 324

GciRemoveOopsFromNsc

Remove one or more OOPs from an NSC.

Syntax

```
BoolType GciRemoveOopsFromNsc(
      OopType                     theNsc,
      const OopType               theOops[ ],
      int                         numOops );
```

Arguments

theNsc	The OOP of the NSC from which the OOPs will be removed.
theOops	The array of OOPs to be removed from the NSC.
numOops	The number of OOPs to remove.

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

See “Executing the examples” on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
BoolType removeOopsFromNsc_example(void)
{
   OopType subColl = GciExecuteStr(
      "AllAccounts select:[ea|ea id > 2000 ]", OOP_NIL);

   OopType buf[10];
   int numRet = GciFetchVaryingOops(subColl, 1, buf, 10);
   /* buf contains at most 10 accounts with ids > 2000 */

   OopType aColl = GciResolveSymbol("AllAccounts", OOP_NIL);
   BoolType allPresent = GciRemoveOopsFromNsc(aColl, buf, numRet);

   /* release because it is the result from an execute */
   GciReleaseOops(&subColl, 1);
   return allPresent;
}
```

See Also

GciAddOopToNsc on page 94
GciAddOopsToNsc on page 95
GciNscIncludesOop on page 276
GciRemoveOopFromNsc on page 323

GciReplaceOops

Replace all instance variables in a GemStone object.

Syntax

```
void GciReplaceOops(
    OopType                  theObj,
    const OopType            theOops[ ],
    int                      numOops );
```

Arguments

theObj	The object whose instance variables are to be replaced.
theOops	An array of OOPs used as the replacements.
numOops	The number of elements in *theOops*.

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 on page 327
GciStoreIdxOops on page 364
GciStoreNamedOops on page 367
GciStoreOops on page 371

GciReplaceVaryingOops

Replace all unnamed instance variables in an NSC object.

Syntax

```
void GciReplaceVaryingOops(
    OopType              theNsc,
    const OopType        theOops[ ],
    int                  numOops );
```

Arguments

theNsc	The NSC object whose unnamed instance variables are replaced.
theOops	An array of OOPs used as the replacements.
numOops	The number of elements in *theOops***.**

Description

GciReplaceVaryingOops uses structural access to replace all unnamed instance variables in the NSC object.

See Also

GciReplaceOops on page 326
GciStoreIdxOops on page 364
GciStoreNamedOops on page 367
GciStoreOops on page 371

GciResolveSymbol

Find the OOP of the object to which a symbol name C string refers, in the context of the current session's user profile.

Syntax

OopType **GciResolveSymbol**(

const char *	*cString*,
OopType	*symbolList*);

Arguments

cString	The name of a Symbol as a C character string.
symbolList	The OOP of a GemStone symbol list, an instances of OOP_CLASS_SYMBOL_LIST. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

The OOP of the object that corresponds to the specified string.

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 on page 329

GciResolveSymbolObj

Find the OOP of the object to which a symbol name String or MultiByteString refers, in the context of the current session's user profile.

Syntax

OopType **GciResolveSymbolObj**(
  OopType *aStringObj*,
  OopType *symbolList*);

Arguments

aStringObj	The OOP of a kind of String or MultiByteString.
symbolList	The OOP of a GemStone symbol list, an instances of OOP_CLASS_SYMBOL_LIST. A value of OOP_NIL means to use the default symbol list for the current GemStone session (that is, `System myUserProfile` *symbolList*).

Return Value

The OOP of the Symbol that corresponds to the specified String.

Description

Attempts to resolve *aStringObj* 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 for a String instead of a C string.

See Also

GciResolveSymbol on page 328

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 for C Shared Libraries” on page 43 for more information.

See Also

GciRtlLoad on page 331
GciRtlUnload on page 333

GciRtlLoad

Load a GemBuilder library.

Syntax

BoolType **GciRtlLoad**(	
BoolType	*useRpc,*
const char *	*path,*
char	*errBuf*[ ],
size_t	*errBufSize*);

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 for C Shared Libraries" on page 43 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: the appropriate subdirectory of $GEMSTONE, depending on the platform and if it is 32-bit or 64-bit.

If a GemBuilder library is already loaded, the call fails.

Example

```
void load_library(void)
{
  char errBuf[1024];
  BoolType result = GciRtlLoad(true, "$GEMSTONE/lib", errBuf,
                 sizeof(errBuf));
  if (result == FALSE)
     printf("Library load failed, %s\n", errBuf);
}
```

See Also

GciRtlIsLoaded on page 330
GciRtlUnload on page 333

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 for C Shared Libraries” on page 43 for more information.

See Also

GciRtlIsLoaded on page 330
GciRtlLoad on page 331

GciSaveAndTrackObjs

Add objects to GemStone's internal GciTrackedObjs set to prevent them from being garbage collected.

Syntax

```
void GciSaveAndTrackObjs(
    const OopType          theOops[ ],
    int                    numOops );
```

Arguments

theOops	An array of OOPs.
numOops	The number of elements in *theOops***.**

Description

The **GciSaveAndTrackOops** 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 **GciSaveAndTrackOops** 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

"Garbage Collection" on page 18
GciDirtyTrackedObjs on page 141
GciReleaseAllOops on page 317
GciReleaseAllTrackedOops on page 318
GciReleaseTrackedOops on page 322
GciSaveObjs on page 336
GciTrackedObjsInit on page 395
GciTrackedObjsFetchAllDirty on page 393
GciDirtySaveObjs on page 139
GciDirtyTrackedObjs on page 141

GciSaveGlobalObjs

Add an array of OOPs to the PureExportSet, making them ineligible for garbage collection.

Syntax

```
void GciSaveGlobalObjs(
    const OopType             theOops[ ],
    int                       numOops );
```

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 18
GciReleaseAllGlobalOops on page 316
GciReleaseAllOops on page 317
GciReleaseGlobalOops on page 319
GciReleaseOops on page 320
GciSaveObjs on page 336

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

```
void GciSaveObjs(
    const OopType          theOops[ ],
    int                    numOops );
```

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 18
GciReleaseAllGlobalOops on page 316
GciReleaseAllOops on page 317
GciReleaseGlobalOops on page 319
GciReleaseOops on page 320
GciSaveGlobalObjs on page 335

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

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

BoolType **GciSetCacheName_**(
    char * *name*);

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.

GciSetErrJump

Enable or disable the current error handler.

Syntax

BoolType **GciSetErrJump**(
  BoolType *aBoolean*);

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 **GciErr** on page 150 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
}
```

See Also

GciErr on page 150
GciLongJmp on page 233
GciPopErrJump on page 309
GciPushErrJump on page 312
GciSetErrJump on page 340

GciSetHaltOnError

Halt the current session when a specified error occurs.

Syntax

```
int GciSetHaltOnError(
    int                         errNum );
```

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.

To unset after a previous call to **GciSetHaltOnError**, invoke this function with an *errNum* of zero.

See Also

GciContinue on page 122
GciDbgEstablish on page 130

Gci_SETJMP

(MACRO) Save a jump buffer in GemBuilder's error jump stack.

Syntax

void **Gci_SETJMP**(
    GciJmpBufSType *   *jumpBuffer*);

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.

See Also

GciErr on page 150
GciLongJmp on page 233
GciPopErrJump on page 309
GciPushErrJump on page 312
GciSetErrJump on page 340

GciSetNet

Set network parameters for connecting the user to the Gem and Stone processes.

Syntax

```
void GciSetNet(
    const char        stoneNameNrs[ ],
    const char        HostUserId[ ],
    const char        HostPassword[ ],
    const char        gemServiceNrs[ ] );
```

Arguments

stoneNameNrs	The NRS for the Stone repository to login into.
hostUserId	UNIX host login user Id. This may be NULL or an empty string if the Gem will run using the userId of the NetLDI process .
hostPassword	Password of the UNIX user, or NULL if not specifying *hostUserId*.
gemServiceNrs	The NRS for the GemService, gemnetobject or a custom gem script. If an empty string, specifies a linked login.

Description

Your application, your GemStone session (Gem), and the database monitor (Stone) can all run in separate processes, on separate machines in your network. **GciSetNet** 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.

stoneNameNrs 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.

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
```

gemServiceNrs 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. Each time an application user logs in to GemStone (after the first time in linked applications), the GemStone service must create a new Gem. *gemServiceNrs* is required except in the special case of a linked application that limits itself to one GemStone login per application process. In this special case, specify *gemServiceNrs* 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 - the GemStone username and password will be provided as arguments to **GciLogin**. *HostUserId* and *HostPassword* provide authentication for such tasks as creating a Gem and establishing communications with a Stone, and are optional in some configuration. 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.

If the *HostUserId* argument is set to an empty C string or a NULL pointer, GemBuilder will try to find a username and password for authentication on a host machine in your network initialization file. To prevent GemBuilder from looking for authentication information in the network initialization file, supply a valid non-empty C string for the *HostUserId* argument, and 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.

Example

For an example of how **GciSetNet** is used, see the example for **GciLogin** on page 228.

See Also

GciEncrypt on page 148
GciLogin on page 228
GciLoginEx on page 230
GciSetNetEx on page 346

GciSetNetEx

GciSetNetEx_

Set network parameters for connecting the user to the Gem and Stone processes, allowing encryption.

Syntax

```
void GciSetNetEx(
    const char          stoneNameNrs[ ],
    const char          hostUserId[ ],
    const char          hostPassword[ ],
    const char          gemServiceNrs[ ] ,
    BoolType            hostPasswordIsEncrypted);

BoolType GciSetNetEx_(
    const char          StoneNameNrs[ ],
    const char          hostUserId[ ],
    const char          hostPassword[ ],
    const char          gemServiceNrs[ ] ,
    BoolType            hostPasswordIsEncrypted,
    char *              errString,
    size_t              maxErrSize);
```

Arguments

stoneNameNrs	The NRS for the Stone repository to login into.
hostUserId	UNIX host login user Id. This may be NULL or an empty string if the Gem will run using the userId of the NetLDI process .
hostPassword	Password of the UNIX user, or NULL if not specifying *hostUserId*.
gemServiceNrs	The NRS for the GemService, gemnetobject or a custom gem script. If an empty string, specifies a linked login.
hostPwIsEncrypted	TRUE if the value for *hostPassword* has been encrypted using **GciEncrypt**.
errString	If there is a syntax error in the NRS for the *StoneName* or *GemService*, GciSetNetEx_ returns FALSE and sets *errString* to the details of the error.
maxErrSize	Maximum number of bytes of error message to return.

Return Value

GciSetNetEx has no return value. **GciSetNetEx_** returns TRUE if the network parameters were set correctly, FALSE if there was a syntax error in *stoneNameNrs* or *gemServiceNrs.*

Description

This function is similar to **GciSetNet**, but allows specifying additional behavior. For details on how to specify the StoneName and GemService using NRS, and the requirements for HostUserId, HostPassword, see **GciSetNet** on page 344.

GciSetNetEx allows you to specify that the host password you send is encrypted. The host password may be encrypted using **GciEncrypt**. To encrypt the GemStone password, use **GciLoginEx** and specify the associated login flag.

Example

For an example of how **GciSetNet** is used, see **GciLogin** on page 228.

See Also

GciEncrypt on page 148
GciLogin on page 228
GciLoginEx on page 230
GciSetNet on page 344

GciSetSessionId

Set an active session to be the current one.

Syntax

void **GciSetSessionId**(
    GciSessionIdType *sessionId*);

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.

See Also

GciGetSessionId on page 207
GciLogin on page 228

GciSetSharedCounter

Set the value of a shared counter.

Syntax

```
BoolType GciSetSharedCounter(
    int             counterIdx,
    int64_t *       value);
```

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

GciDecSharedCounter on page 135
GciFetchNumSharedCounters on page 178
GciFetchSharedCounterValuesNoLock on page 189
GciIncSharedCounter on page 213
GciReadSharedCounter on page 314
GciReadSharedCounterNoLock on page 315

GciSetTraversalBufSwizzling

Control swizzling of the traversal buffers.

Syntax

```
BoolType GciSetTraversalBufSwizzling(
    BoolType                    enabled );
```

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 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**

For more information on swizzling, see “Byte-Swizzling of Binary Floating-Point Values” on page 26.

GciSetVaryingSize

Set the size of a collection.

Syntax

void **GciSetVaryingSize**(
OopType *collection*,
int64 *size);*

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_ on page 198

GciShutdown

Logout from all sessions and deactivate GemBuilder.

Syntax

void **GciShutdown**( )

Description

This function is intended to be called by image exit routines, such as the **on_exit** system call. In the linkable GemBuilder, **GciShutdown** calls **GciLogout**. In the RPC version, it logs out all sessions connected to the Gem process and shuts down the networking layer, thus releasing all memory allocated by GemBuilder.

It is especially important to call this function explicitly on any computer whose operating system does not automatically deallocate resources when a process quits. This effect is found on 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. This is useful, for example, if the a Smalltalk method is invoked that enters an infinite loop.

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.

GciHardBreak has no effect if called from within a User Action.

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.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);
        }
      }
    } while (! done);

    // restore previous SIGINT handler
    sigaction(SIGINT, &oldHandler, NULL);
  }
```

See Also

GciClearStack on page 115
GciContinue on page 122
GciHardBreak on page 208

GciStep

GciStep_

Continue code execution in GemStone with specified single-step semantics.

Syntax

```
OopType GciStep(
    OopType                 gsProcess,
    int                     level );

OopType GciStep_(
    OopType                 gsProcess,
    int                     level
    BoolType                through);
```

Arguments

gsProcess	The OOP of a GsProcess object (obtained as the value of the context field of an error report returned b y **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
through	When *level* =1 and this argument is TRUE, provides step through semantics, stopping in blocks for which the top of the stack frame is the home method.

Return Value

Returns the OOP of the result of the Smalltalk execution. Returns OOP_ILLEGAL in case of error.

Description

This 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**(*gsProcess*, "stackDepth", NULL, 0), this function generates an error.

GciStoreByte

Store one byte in a byte object.

Syntax

void **GciStoreByte**(
    OopType   *theObject*,
    int64   *atIndex*,
    ByteType   *theByte*);

Arguments

theObject	The OOP of the GemStone byte object into which the store will be done.
startIndex	The index into theObject at which to begin storing bytes.
theByte	The 8-bit value to be stored.

Description

This 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 on page 163
GciStoreBytes on page 357

GciStoreBytes

(MACRO) Store multiple bytes in a byte object.

Syntax

void **GciStoreBytes**(

OopType	*theObject*,
int64	*startIndex*,
const ByteType	*theBytes*[ ],
int64	*numBytes*);

Arguments

theObject	The OOP of the GemStone byte object, into which the bytes will be stored.
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.

Description

This 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. For an object with multiple bytes per character or digit, *theBytes* is expected to be in client native byte order, and will be swizzled if needed by the server.

Error Conditions

GciStoreBytes raises an error if *theObject* is a Float or SmallFloat. Use **GciStoreBytesInstanceOf** instead for Float or SmallFloat objects.

Example

```
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 on page 163
GciFetchBytes_ on page 164
GciStoreByte on page 356
GciStoreBytesInstanceOf on page 359
GciStoreChars on page 361

GciStoreBytesInstanceOf

Store multiple bytes in a byte object.

Syntax

void **GciStoreBytesInstanceOf**(

OopType	*theClass*,
OopType	*theObject*,
int64	*startIndex*,
const ByteType	*theBytes*[ ],
int64	*numBytes*);

Arguments

theClass	The OOP of the class of the GemStone byte object.
theObject	The OOP of the GemStone byte object into which the bytes will be stored.
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

Description

This function uses structural access to store multiple elements from a C array into a byte object, beginning at a specified index. A common application of **GciStoreBytesInstanceOf** would be to store a Float or LargeInteger object.

GciStoreBytesInstanceOf provides automatic byte swizzling for objects such as Float, LargeInteger, and DoubleByteString that use multiple bytes per digit or character. For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26. For these objects, *theBytes* is assumed to be in client native byte order. For DoubleByteStrings, *startIndex* must be aligned on character boundaries and *numBytes* must be a multiple of the number of bytes per character; for numeric objects *startIndex* must be one and the *numBytes* the size of the numeric class.

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

```
void storeBytesInstof_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 on page 163
GciFetchBytes_ on page 164
GciStoreByte on page 356
GciStoreBytes on page 357
GciStoreChars on page 361

GciStoreChars

Store multiple ASCII characters in a byte object.

Syntax

```
void GciStoreChars(
      OopType                    theObject,
      int64                      startIndex,
      const char *               aString );
```

Arguments

theObject	The OOP of the GemStone byte object, into which the chars will be stored.
startIndex	The index into *theObject* at which to begin storing bytes.
aString	The string to be stored.

Description

This 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

```
void storeChars_example(void)
{
  OopType oString = GciNewOop(OOP_CLASS_STRING);

  GciStoreChars(oString, 1, "some string data");
}
```

See Also

GciFetchBytes_ on page 164
GciStoreBytes on page 357

GciStoreDynamicIv

Create or change the value of an object's dynamic instance variable.

Syntax

void **GciStoreDynamicIv**(
    OopType *theObject*,
    OopType *aSymbol,*
    OopType *value*);

Arguments

theObject	The OOP of the object.
aSymbol	Specifies the name of the dynamic instance variable.
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 on page 170
GciFetchDynamicIvs on page 171

GciStoreIdxOop

Store one OOP in an indexable pointer object's unnamed instance variable.

Syntax

void **GciStoreIdxOop**(
OopType *theObject*,
int64 *atIndex*,
OopType *theOop*);

Arguments

theObject	The OOP of a pointer object, into which the *theOop* will be stored
atIndex	The index into *theObject* at which to store *theOop*.
theOop	The OOP to be stored.

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 **GciAddOopToNsc** on page 94.)

Example

```
void storeIdxOop_example(void)
{
   GciErrSType err;
   OopType oArray = GciExecuteStr("Array new: 5", OOP_NIL);
   OopType oSet = GciExecuteStr("Set with: 'GBS/VA' with: 'GBS/VW'",
               OOP_NIL);

   // store the Set into the 3rd slot of the Array
   GciStoreIdxOop(oArray, 3, oSet);

   // release results of execution
   GciReleaseOops(&oArray, 1);
   GciReleaseOops(&oSet, 1);
}
```

See Also

GciAddOopToNsc on page 94
GciFetchVaryingOop on page 194
GciFetchVaryingOops on page 196
GciStoreIdxOops on page 364

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*);

Arguments

theObject	The OOP of a pointer object into which *theOops* will be stored.
startIndex	The index into *theObject* at which to begin storing OOPs.
theOops	The array of OOPs to be stored.
numOops	The number of elements to store

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 **GciAddOopsToNsc** on page 95.)

Example

This example creates a new Array containing the first 10 elements of Globals.

```
BoolType storeIdxOops_example(void)
{
   GciErrSType err;
   OopType oArray = GciExecuteStr("Array new: 10", OOP_NIL);
   OopType oGlobals = GciExecuteStr("Globals", OOP_NIL);
   int buf_size = 10;
   OopType oBuffer[buf_size];

   // Fetch first 10 elements in Globals
   int numRet = GciFetchVaryingOops(oGlobals, 1, oBuffer, buf_size);

   // and put these in oArray
   GciStoreIdxOops(oArray, 1, oBuffer, numRet);
   if (GciErr(&err)) {
      printf("Error %d, %s\n", err.number, err.message);
      return false;   }

   // release results of execution
   GciReleaseOops(&oArray, 1);
   return true;
}
```

See Also

GciAddOopToNsc on page 94
GciFetchVaryingOops on page 196
GciReplaceOops on page 326
GciReplaceVaryingOops on page 327
GciStoreIdxOops on page 364
GciStoreNamedOops on page 367
GciStoreOops on page 371

GciStoreNamedOop

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

Syntax

void **GciStoreNamedOop**(

OopType	*theObject*,
int64	*atIndex*,
OopType	*theOop*);

Arguments

theObject	The Object into which to store the OOP.
startIndex	The index into *theObject*'s named instance variables at which to begin storing OOPs.
theOops	The array of OOPs to be stored.

Description

This function stores a single OOP into an object's named instance variable at the specified index, using structural access.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void storeNamedOop_example(OopType anAccount)
{
   OopType newName = GciNewString("Josiah Chang");

   //  assign new value to instvar at a hardcoded offset
   int indexOfSalesRep = 3;
   GciStoreNamedOop(anAccount, indexOfSalesRep, newName);

   //  assign new value to instvar without knowing offset
   GciStoreNamedOop(anAccount,
         GciIvNameToIdx(GciFetchClass(anAccount), "salesRep"),
         newName);
}
```

See Also

GciFetchNamedOop on page 172
GciFetchVaryingOop on page 194
GciStoreIdxOop on page 363
GciStoreNamedOops on page 367

GciStoreNamedOops

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

Syntax

void **GciStoreNamedOops**(

OopType	*theObject*,
int64	*startIndex*,
const OopType	*theOops*[ ],
int	*numOops*);

Arguments

theObject	The Object in which to store the OOP.
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.

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

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void storeNamedOops_example(OopType anAddress)
{
   // allocation buffer for unknown number of instvars
   int namedSize = GciFetchNamedSize(anAddress);
   if (namedSize == 0) { return ; }
   OopType *oBuffer = (OopType*) malloc( sizeof(OopType) * namedSize );
   if (oBuffer == NULL) { return; }

   // fetch existing values
   int numRet = GciFetchNamedOops(anAddress, 1, oBuffer, namedSize);
   if (numRet != namedSize) {
      printf("error during fetch\n");
      return;
   }

   // update street and city, but not state
   OopType newValue = GciNewString("4545 45th Str");
   int ivOffset = GciIvNameToIdx(GciFetchClass(anAddress), "street");
```

```
      oBuffer[ivOffset  - 1 ] = newValue;
      newValue = GciNewString("Tualatin");
      ivOffset = GciIvNameToIdx(GciFetchClass(anAddress), "city");
      oBuffer[ivOffset  - 1 ] = newValue;

      // store changes
      GciStoreNamedOops(anAddress, 1, oBuffer, namedSize);
   }
```

See Also

GciAddOopToNsc on page 94
GciFetchNamedOops on page 173
GciFetchVaryingOops on page 196
GciReplaceOops on page 326
GciReplaceVaryingOops on page 327
GciStoreIdxOops on page 364
GciStoreNamedOop on page 366
GciStoreOops on page 371

GciStoreOop

Store one OOP into an object's instance variable.

Syntax

void **GciStoreOop**(
OopType *theObject*,
int64 *atIndex*,
OopType *theOop*);

Arguments

theObject	The Object in which to store the OOP.
startIndex	The index into *theObject*'s named or unnamed instance variables 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
theOops	The OOP to be stored.

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 **GciAddOopToNsc** on page 94.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void storeOop_example(OopType anAccount)
{
   // Two ways to assign new value to a named instance variable of
            anAccount
   OopType newName = GciNewString("Jeni Johnson-Klien");
   int indexOfSalesRep = 3;
   GciStoreOop(anAccount, indexOfSalesRep, newName);
   GciStoreNamedOop(anAccount, indexOfSalesRep, newName);

   // Two ways to assign a product string to the first indexed instance
            variable
   OopType newProduct = GciNewString("GemConnect");
   GciStoreOop(anAccount, GciFetchNamedSize(anAccount) + 1, newProduct);
   GciStoreIdxOop(anAccount, 1, newProduct);
}
```

See Also

GciAddOopToNsc on page 94
GciFetchNamedOop on page 172
GciFetchVaryingOop on page 194
GciFetchOops on page 184
GciStoreOops on page 371

GciStoreOops

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

Syntax

void **GciStoreOops**(
OopType *theObject*,
int64 *startIndex*,
const OopType *theOops*[ ],
int *numOops*);

Arguments

theObject	The Object in which to store the OOPs.
startIndex	The index into *theObject*'s named or unnamed instance variables 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 OOP to be stored.
numOops	The number of OOPs to store

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 **GciAddOopsToNsc** on page 95.

Example

See "Executing the examples" on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

```
void storeOops_example(OopType anAccount)
{
   int namedSize = GciFetchNamedSize(anAccount);
   int64 instSize = GciFetchSize_(anAccount);

   // allow space in buffer for storing into first varying instVar plus
   // appending a new varying instVar
   int64 bufVaryingSize = instSize - namedSize + 1;

   int64 bufSize = namedSize + bufVaryingSize;
   OopType *buf = (OopType*) malloc(sizeof(OopType) * bufSize);
   if (buf == NULL) {
      printf("malloc failure");
      return;
   }
```

```
     GciFetchOops(anAccount, 1, buf, instSize);

     OopType newName = GciNewString("Josiah Chang");
     int indexOfSalesRep = GciIvNameToIdx(GciFetchClass(anAccount),
                  "salesRep");
     buf[indexOfSalesRep - 1] = newName;

     OopType firstProduct = GciNewString("GS64");
     OopType newProduct = GciNewString("GemConnect");

    // assign first element
    buf[namedSize] = firstProduct;

    // append new product
    int64 newSize = instSize + 1;
    buf[newSize - 1] = newProduct;

    // now store all the instVars back to the repository
    GciStoreOops(anAccount, 1, buf, newSize);
  }
```

See Also

GciAddOopsToNsc on page 95
GciFetchNamedOops on page 173
GciFetchVaryingOop on page 194
GciFetchOops on page 184
GciRemoveOopsFromNsc on page 324
GciReplaceOops on page 326
GciReplaceVaryingOops on page 327
GciStoreIdxOops on page 364
GciStoreNamedOops on page 367
GciStoreOops on page 371

GciStorePaths

Store selected multiple OOPs into an object tree.

Syntax

BoolType **GciStorePaths**(

const OopType	*theOops*[ ],
int	*numOops*,
const int	*paths*[ ],
const int	*pathSizes*[ ],
int	*numPaths*,
const OopType	*newValues*[ ],
int *	*failCount*);

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*.
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 *i*th 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.

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:

```
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.

Example of a single object and a single path

See “Executing the examples” on page 91 for the Smalltalk code that defines the classes and objects that support the examples.

This example updates the street address along one path for a single account.

```
BoolType storePaths_example1(OopType anAccount)
{
   GciErrSType errInfo;
   int path_size = 3;
   int aPath[path_size];
   aPath[0] = 2; //customer
   aPath[1] = 2; //address
   aPath[2] = 1; //street

   // update anAccount.customer.address.street with the newValue
   OopType newValue = GciNewString("312 North Rd");
   int failCount;
   GciStorePaths(&anAccount, 1, aPath, &path_size, 1, &newValue,
               &failCount);
   if (failCount != 0) {
      printf("GciStorePaths fail count is %d\n", failCount);
      return false;
      }
   if (GciErr(&errInfo)) {
      printf("GciStorePaths error %d, %s\n", errInfo.number,
               errInfo.message);
      return false;
    }
   return true;
}
```

Example with multiple objects and multiple paths

This example updates several values along two paths for two Accounts.

```
BoolType storePaths_example2(int numOfAccounts, OopType* resultAccounts)
{
   GciErrSType errInfo;
   OopType myAccts;
   myAccts = GciExecuteStr("AllAccounts", OOP_NIL);
   if (myAccts == OOP_NIL) { return false; }

   int numRet = GciFetchVaryingOops(myAccts, 1, resultAccounts,
               numOfAccounts);
   if (numRet != numOfAccounts) { return false; }

   int path_segments[2];
   path_segments[0] = 2;
   path_segments[1] = 3;
   int serialPath[6]; // size is sum of path_segments
   serialPath[0] = 2; //customer
   serialPath[1] = 1; //name
   serialPath[2] = 2; //customer
   serialPath[3] = 2; //address
   serialPath[4] = 2; //city

   OopType newValues[2 * numOfAccounts];
      // size of path_segments * number of accounts
   newValues[0] = GciNewString("Samual B. Houston");
   newValues[1] = GciNewString("Paisley");
   newValues[2] = GciNewString("Maya J. Johnson");
   newValues[3] = GciNewString("Bend");

   int failCount;
   GciStorePaths(resultAccounts, numOfAccounts, serialPath,
      path_segments, 2, newValues, &failCount);

   if (failCount != 0) {
      printf("GciStorePaths fail count is %d\n", failCount);
      return false;
      }
   if (GciErr(&errInfo)) {
      printf("GciStorePaths error %d, %s\n", errInfo.number,
               errInfo.message);
      return false;
    }
   return true;
}
```

See Also

GciFetchPaths on page 186

GciStoreTrav

Store multiple traversal buffer values in objects.

Syntax

void **GciStoreTrav**(
    GciTravBufType * *travBuff*,
    int *behaviorFlag*);

Arguments

travBuff	A traversal buffer, which contains object data to be stored.
behaviorFlag	Flag bits that determines how the objects should be handled. Either use GCI_STORE_TRAV_DEFAULT, or any combination of the other flags. GCI_STORE_TRAV_DEFAULT = 0 Default behavior: use "add" semantics for varying instvars of NSC; if the object to be stored into does not exist, give error. GCI_STORE_TRAV_NSC_REP = 0x1, Use REPLACE semantics for varying instvars of NSC GCI_STORE_TRAV_CREATE = 0x2 If an object to be stored into does not exist, create the new object and add it to the PureExportSet .

Description

This function stores data from the traversal buffer *travBuff* into multiple GemStone objects. A traversal buffer is a specialized C structure that optimizes transfer of data. **GciStoreTrav**, by providing a way to modify and create a number of new object in one command, allows you to reduce the number of GemBuilder calls that are required for your application program to store complex objects in the database.

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**. A **GciObjRepSType** includes a header with the OOP and class of the object, and a variable-length data area. **GciStoreTrav** stores data object by object, using one object report at a time.

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 only modifies 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. The new object is initialized with the data in its object report.

To create a new object based on an object report, you need to provide the OOP of the object, using **GciGetFreeOops**.

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.

When an existing NSC is modified, 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* includes 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. See "Byte-Swizzling of Binary Floating-Point Values" on page 26.

This function raises an error if the traversal buffer contains a report for any object of special implementation format; special objects cannot be modified.

Use of Object Reports

GciStoreTrav 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 "Object Report Header - GciObjRepHdrSType" on page 71). **GciStoreTrav** uses the fields in each object report as follows:

report->header.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.

report->header.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.

report->header.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.

report->header.objId
The OOP of the object to be stored.

report->header.oclass
Used only when creating a new object, to identify its intended class.

report->header.objectSecurityPolicyId
The ID of the object's security policy.

report->header.clearBits()
Must be called before any of the following:

report->header.setObjImpl()
You must call *rpt->hdr.setObjImpl* to set this field to be consistent with the object's implementation. Formats usable here are GC_FORMAT_OOP, GC_FORMAT_BYTE, and GC_FORMAT_NSC.

report->header.setInvariant()
Boolean value. Call *rpt->hdr.setInvariant(TRUE)* if you want this object to be made invariant after the store specified by *report** is completed.

report->valueBufferBytes()
The value buffer of an object of byte format.

report->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 on page 234
GciNbStoreTrav on page 259
GciNewOopUsingObjRep on page 271
GciProcessDeferredUpdates_ on page 310
GciStoreTravDo_ on page 379
GciTraverseObjs on page 396

GciStoreTravDo_

Store multiple traversal buffer values in objects, execute the specified code, and return the resulting object.

Syntax

OopType **GciStoreTravDo_**(
    GciStoreTravDoArgsSType * *storeTravArgs*);

Arguments

storeTravArgs	An instance of **GciStoreTravDoArgsSType**.

Return Value

Returns the OOP of the result of executing the specified code. In case of error, this function returns OOP_NIL.

Description

GciStoreTravDo_, like **GciStoreTrav**, stores data from a traversal buffer into multiple GemStone objects, and also executes supplied code, all in the same network round-trip.

For details of the fields in the instance of **GciStoreTravDoArgsSType**, see "Store Traversal Arguments - GciStoreTravDoArgsSType" on page 75. The fields of this structure include the traversal buffer and flags as described under "GciStoreTrav" on page 376.

GciStoreTravDoArgsSType includes the *doPerform* field to indicate different perform options.

- If *doPerform* is 0, this function executes a string using *args->u.executestr*, with the semantics of "GciExecuteFromContext" on page 153.
- If *doPerform* is any other value, then executes a perform using *args->u.perform*, with the semantics of "GciPerformNoDebug" on page 300.

The remaining fields supply needed output after the function has completed. Read *alteredTheOops* to get the OOPs of the objects that were modified and 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.

Similarly to **GciStoreTrav**, **GciStoreTravDo_** provides automatic byte swizzling, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26.

Handling Error Conditions

If you get a run time error while executing **GciStoreTravDo_**, we recommend that you abort the current transaction.

See Also

GciAlteredObjs on page 99
GciExecuteStrFromContext on page 159
GciMoreTraversal on page 234
GciNbStoreTravDo_ on page 260
GciNewOopUsingObjRep on page 271
GciPerformNoDebug on page 300
GciProcessDeferredUpdates_ on page 310
GciStoreTrav on page 376
GciTraverseObjs on page 396

GciStoreTravDoTrav_

Combine in a single function the calls to **GciStoreTravDo_** and **GciClampedTrav**, to store multiple traversal buffer values in objects, execute specified code, and traverse the result object.

Syntax

BoolType **GciStoreTravDoTrav_**(
    GciStoreTravDoArgsSType * *storeTravArgs*,
    GciClampedTravArgsSType **clampedTravArgs*);

Arguments

storeTravArgs	An instance of **GciStoreTravDoArgsSType**. For details, refer to the discussion of **GciStoreTravDo_** on page 379.
clampedTravArgs	An instance of **GciClampedTravArgsSType**. For details, refer to the discussion of **GciClampedTrav** on page 110.

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 the descriptions for **GciStoreTravDo_** on page 379 and **GciClampedTrav** on page 110 for details.

See Also

GciClampedTrav on page 110
GciStoreTrav on page 376
GciStoreTravDo_ on page 379

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.

Syntax

int **GciStoreTravDoTravRefs_**(
const OopType * *oopsNoLongerReplicated*,
int *numNotReplicated*,
const OopType * *oopsGcedOnClient*,
int *numGced,*
GciStoreTravDoArgsSType * *storeTravArgs*,
GciClampedTravArgsSType **clampedTravArgs*);

Argumentsv

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*.
storeTravArgs	An instance of **GciStoreTravDoArgsSType**, as described under “Store Traversal Arguments - GciStoreTravDoArgsSType” on page 75.
clampedTravArgs	An instance of **GciClampedTravArgsSType**. For details, see **Clamped Traversal Arguments - GciClampedTravArgsSType** on page 77, and the discussion of **GciStoreTrav** on page 376. There is one exception: in this function, *retrievalFlags* GCI_RETRIEVE_EXPORT and GCI_CLEAR_EXPORT are ignored.

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.

GciStoreTravDoArgsSType includes the *doPerform* field to indicate different perform options. The doPerform, which also determines which nested structure is used in the union field.

- If *doPerform* is 0, this function executes a string using *args->u.executestr*, with the semantics of "GciExecute" on page 151.
- If *doPerform* is 1, then executes a perform using *args->u.perform*, with the semantics of "GciPerform" on page 296.
- If *doPerform* 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 *doPerform* 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 *doPerform* is 4, resume execution of a suspended Smalltalk Process using *stdArgs->u.continueArgs*, with the semantics of "GciContinueWith" on page 123.

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

GciAlteredObjs on page 99
GciClampedTrav on page 110
GciExecuteStrFromContext on page 159
GciMoreTraversal on page 234
GciNbStoreTravDoTravRefs_ on page 262
GciPerformNoDebug on page 300
GciStoreTrav on page 376
GciStoreTravDo_ on page 379
GciStoreTravDoTrav_ on page 381
GciTraverseObjs on page 396

GciStringToInteger

Convert a C string to a GemStone SmallInteger or LargeInteger object.

Syntax

```
OopType GciStringToInteger(
    const char*                 string,
    int64                       stringSize );
```

Arguments

string	The C string to be translated into a GemStone SmallInteger or LargeInteger object.
stringSize	The length of *string*.

Return Value

Returns the OOP of the GemStone SmallInteger or LargeInteger 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 or LargeInteger object that has the same value.

Leading blanks are ignored. Trailing non-digits are ignored.

GciStrKeyValueDictAt

Find the value in a symbol KeyValue dictionary at the corresponding string key.

Syntax

```
void GciStrKeyValueDictAt(
    OopType             theDict,
    const char *        keyString,
    OopType *           value );
```

Arguments

theDict	The OOP of a SymbolKeyValueDictionary.
keyString	A string that matches a key in *theDict*.
value	A pointer to the variable that is to receive the OOP of the returned value.

Description

Returns the value in symbol KeyValue dictionary *theDict* that corresponds to key *keyString*. If an error occurs or *keyString* is not found, *value* is OOP_ILLEGAL. KeyValue dictionaries do not have associations, so no association is returned. **GciStrKeyValueDictAt** is equivalent to **GciStrKeyValueDictAtObj** except that the key is a character string, not an object.

See Also

GciStrKeyValueDictAtObj on page 386
GciStrKeyValueDictAtPut on page 388

GciStrKeyValueDictAtObj

Find the value in a symbol KeyValue dictionary at the corresponding object key.

Syntax

void **GciStrKeyValueDictAtObj**(
    OopType *theDict*,
    OopType *keyObj*,
    OopType * *value*);

Arguments

theDict	The OOP of a SymbolKeyValueDictionary.
keyObj	The OOP of a key in *theDict*.
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

GciStrKeyValueDictAt on page 385
GciStrKeyValueDictAtObjPut on page 387

GciStrKeyValueDictAtObjPut

Store a value into a symbol KeyValue dictionary at the corresponding object key.

Syntax

void **GciStrKeyValueDictAtObjPut**(

OopType	*theDict*,
OopType	*keyObj*,
OopType	*theValue*);

Arguments

theDict	The OOP of a SymbolKeyValueDictionary into which the object is to be stored.
keyObj	The OOP of a key under which the value is to be stored.
theValue	The OOP of the object to be stored in the SymbolKeyValueDictionary.

Description

Adds object *theValue* to symbol KeyValue dictionary *theDict* with key *keyObj*. Equivalent to the Smalltalk expression:

```
theDict at: keyObj put: theValue
```

See Also

GciStrKeyValueDictAtObj on page 386
GciStrKeyValueDictAtPut on page 388

GciStrKeyValueDictAtPut

Store a value into a symbol KeyValue dictionary at the corresponding string key.

Syntax

```
void GciStrKeyValueDictAtPut(
    OopType              theDict,
    const char *         keyString,
    OopType              theValue );
```

Arguments

theDict	The OOP of a SymbolKeyValueDictionary into which the object is to be stored.
keyObj	The string key under which the value 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 on page 385
GciStrKeyValueDictAtObjPut on page 387

GciSymDictAt

Find the value in a symbol dictionary at the corresponding string key.

Syntax

```
void GciSymDictAt(
      OopType                  theDict,
      const char *             keyString,
      OopType *                value,
      OopType *                association );
```

Arguments

theDict	The OOP of a SymbolDictionary.
keyObj	The OOP of a key in *theDict*.
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

GciStrKeyValueDictAt on page 385
GciSymDictAtObj on page 390
GciSymDictAtPut on page 392

GciSymDictAtObj

Find the value in a symbol dictionary corresponding to the key object.

Syntax

```
void GciSymDictAtObj(
    OopType            theDict,
    OopType            keyObj,
    OopType *          value,
    OopType *          association );
```

Arguments

theDict	The OOP of a SymbolDictionary.
keyObj	The OOP of a key in *theDict*.
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:

```
^ { theDict at: keyObj . theDict associationAt: keyObj }
```

See Also

GciStrKeyValueDictAtObj on page 386
GciSymDictAt on page 389
GciSymDictAtObjPut on page 391

GciSymDictAtObjPut

Store a value into a symbol dictionary at the corresponding object key.

Syntax

```
void GciSymDictAtObjPut(
    OopType          theDict,
    OopType          keyObj,
    OopType          theValue );
```

Arguments

theDict	The OOP of a SymbolDictionary into which the object is to be stored.
keyObj	The OOP of a 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

GciStrKeyValueDictAtObjPut on page 387
GciSymDictAtObj on page 390
GciSymDictAtPut on page 392

GciSymDictAtPut

Store a value into a symbol dictionary at the corresponding string key.

Syntax

```
void GciSymDictAtPut(
    OopType                 theDict,
    const char *            keyString,
    OopType                 theValue );
```

Arguments

theDict	The OOP of a SymbolDictionary into which the object is to be stored.
keyString	The string 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 *keyString*. Equivalent to **GciSymDictAtObjPut**, except the key is a character string, not an object.

See Also

GciStrKeyValueDictAtPut on page 388
GciSymDictAt on page 389
GciSymDictAtObjPut on page 391

GciTrackedObjsFetchAllDirty

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

Syntax

```
void GciTrackedObjsFetchAllDirty(
    OopType         exportedDirty,
    int64 *         numExportedDirty,
    OopType         trackedDirty,
    int64 *         numTrackedDirty );
```

Arguments

exportedDirty	OOP of the collection (an instance of either IdentitySet or IdentityBag) that will contain the objects in the ExportedDirtyObjs set.
numExportedDirty	Pointer to an integer that returns the number of objects in the *exportedDirty* collection.
trackedDirty	OOP of the collection (an instance of either IdentitySet or IdentityBag) that will contain the objects in the TrackedDirtyObjs set.
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 18
GciDirtyExportedObjs on page 136
GciDirtyObjsInit on page 138
GciDirtySaveObjs on page 139
GciDirtyTrackedObjs on page 141
GciSaveAndTrackObjs on page 334
GciTrackedObjsInit on page 395

GciTrackedObjsInit

Reinitialize the set of tracked objects maintained by GemStone.

Syntax

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 on page 139
GciDirtyTrackedObjs on page 141
GciSaveAndTrackObjs on page 334
GciTrackedObjsFetchAllDirty on page 393

GciTraverseObjs

Traverse an array of GemStone objects.

Syntax

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

Arguments

theOops	An array of OOPs representing the objects to be traversed.
numOops	The number of elements in *theOops*.
travBuff	A buffer in which the results of the traversal will be placed. For details, see "Traversal Buffer - GciTravBufType" on page 78.
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.

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.

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, unless **GciSetTraversalBufSwizzling** is used to disable swizzling. For more about byte swizzling, see "Byte-Swizzling of Binary Floating-Point Values" on page 26.

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 under "Use of Object Reports" on page 377.

To create a traversal buffer, use **GciAllocTravBuf**. This function takes an argument of the size of traversal buffer to allocate. Multiple the number of object *m* by the object size to determine the size buffer you require. If the traversal buffer is not large enough, you will need to perform **GciMoreTraversal**.

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 FALSE, 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 TRUE 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.objectSecurityPolicyId	`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, 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 **GciMoreTraversal** on page 234.

See Also

GciFindObjRep on page 199
GciMoreTraversal on page 234
GciNbTraverseObjs on page 264
GciNewOopUsingObjRep on page 271
GciObjRepSize_ on page 280
GciStoreTrav on page 376

GciUncompress

Uncompress the supplied data, assumed to have been compressed with **GciCompress**.

Syntax

int **GciUncompress**(

char *	*dest*,
uint *	*destLen*,
const char *	*source*,
uint	*sourceLen*);

Arguments

dest	Pointer to the buffer 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.

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`.

Description

GciUncompress passes the supplied inputs unchanged to the `uncompress` function in the GNU zlib library Version, and returns the result exactly as the GNU `uncompress` function returns it.

See Also

GciCompress on page 120

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.

See Also

Chapter 4, "Writing User Actions", starting on page 45
GciDeclareAction on page 133
GciInstallUserAction on page 217
GciInUserAction on page 220
GciUserActionShutdown on page 401

GciUserActionShutdown

Enable user-defined clean-up for user actions.

Syntax

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.

See Also

Chapter 4, "Writing User Actions", starting on page 45
GciDeclareAction on page 133
GciInstallUserAction on page 217
GciInUserAction on page 220
GciUserActionInit on page 400

GciUtf8To8Bit

Convert UTF-8 input to 8 bit data.

Syntax

BoolType **GciUtf8To8bit**(

```
int GciUncompress(
    const char *            src,
    char *                  dest,
    ssize_t *               destSize );
```

Description

GciUtf8To8bit converts Utf8 input in **src* to 8 bit data in **dest*. If all code points in *src are in the range 0..255, and the result fits in destSize-1, returns TRUE and *dest is null terminated, otherwise returns FALSE.

GciVersion

Return a string that describes the GemBuilder version.

Syntax

const char* **GciVersion**( )

Return Value

A null-terminated string, containing period-separated fields that describe the specific release of GemBuilder.

Description

GciVersion gets version information from the shared libraries. The string returned describes the GCI version (that is, the version of the shared libraries, not the version of a GemStone repository).

The version number is provided as a series of period-delimited fields within the string, with the first field being the major version number, the second field the minor version number, and so on. There may be any number of additional fields for a particular version of the GCI. Following this is the build information.

More detailed version information, including Stone version information, can be retrieved using System class methods such as `System class >> stoneVersionReport` and `System class >> gemVersionReport`.

Example

```
BoolType version_example(void)
{
   const char* buf;
   buf = GciVersion();
   printf("Version is %s\n", buf);
   return ( buf[0] == '3' );
}
```

An example of the string that is returned is:

```
Version is 3.4.0 build 64bit-42647
```

See Also

GciProduct on page 311

GciX509Login

Start an X.509-authenticated user session.

Syntax

BoolType **GciX509Login**(
    GciX509LoginArg *loginParameters*);

Arguments

loginParameters Instance of the C++ class GciX509LoginArg. This is defined as:

```
class CLS_EXPORT GciX509LoginArg
   public:
   const char  *netldiHostOrIp;
   const char  *netldiNameOrPort;
   const char  *privateKey;
   const char  *cert;
   const char  *caCert;
   const char  *extraGemArgs;
   const char  *dirArg;
   const char  *logArg;
   unsigned int loginFlags;
   BoolType argsArePemStrings;
   BoolType executedSessionInit; // output
```

If *argsArePemStrings* is true, the *privateKey*, *cert*, and *caCert* are strings in PEM format. If false, these are strings containing the name of a file that is in PEM format.

Return Value

Returns true if login succeeded, false otherwise

Description

This function creates a user session and its corresponding transaction workspace, using authentication using X.509 certificates. This login requires that the NetLDIs be setup to use X509 authentication, as described in the GemStone/S 64 Bit X509-Secured GemStone Administration Guide. Specifically, note that the Stone name and User name and password are not included; these are obtained from the certificates.

See Also

GciEncrypt on page 148
GciLogin on page 228
GciSetNet (page 344)
GciSetNetEx (page 346)

Chapter

8 GemStone C Statistics Interface

This chapter 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.

8.1 Developing a GCSI Application

The command lines in this chapter 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:

- `$GEMSTONE/include/shrpcstats.ht` — Defines all cache statistics. (For a list of cache statistics, refer to the "Monitoring GemStone" chapter of the *System Administration Guide for GemStone/S 64 Bit*.)
- `$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/libgcsi-3.5.0-64.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/libgcsi-3.5.0-64.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.

8.2 GCSI Data Types

The following C types are used by GCSI functions. The file shrpcstats.ht 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:

```
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:

```
#define COMMON_STAT_COUNT
  (sizeof(ShrPcCommonStatSType)/sizeof(int))

#define SHRPC_STAT_COUNT
  (sizeof(ShrPcMonStatSType)/sizeof(int) + COMMON_STAT_COUNT)

#define GEM_STAT_COUNT
  (sizeof(ShrPcGemStatSType)/sizeof(int) + COMMON_STAT_COUNT)

#define PGSVR_STAT_COUNT
  (sizeof(ShrPcPgsvrStatSType)/sizeof(int) + COMMON_STAT_COUNT)

#define STN_STAT_COUNT
  (sizeof(ShrPcStnStatSType)/sizeof(int) + COMMON_STAT_COUNT)
```

GcsiAllStatsForMask

Get all cache statistics for a specified set of processes.

Syntax

int **GcsiAllStatsForMask**(

unsigned int	*mask*;
GcsiResultSType *	*result*;
int *	*resultSize*);

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.ht`; see "GCSI Errors" on page 421.

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:

```
unsigned int mask = SHRPC_MONITOR | SHRPC_STONE;
```

GcsiAttachSharedCache

Attach to the specified shared page cache.

Syntax

int **GcsiAttachSharedCache**(
    const char * *fullCacheName*;
    char * *errBuf*;
    size_t *errBufSize*);

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 "GCSI Errors" on page 421.

See Also

GcsiAttachSharedCacheForStone on page 411
GcsiDetachSharedCache on page 412

GcsiAttachSharedCacheForStone

Attaches this process to the specified shared page cache.

Syntax

int **GcsiAttachSharedCacheForStone**(
    const char * *stoneName*,
    char * *errBuf*,
    size_t *errBufSize*);

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`; see "GCSI Errors" on page 421.

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 410) to specify the full name of the shared cache.

See Also

GcsiAttachSharedCache on page 410
GcsiDetachSharedCache on page 412

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 “GCSI Errors” on page 421.

See Also

GcsiAttachSharedCache on page 410
GcsiAttachSharedCacheForStone on page 411

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

int **GcsiFetchMaxProcessesInCache**(
    int * *maxProcesses*);

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`; see “GCSI Errors” on page 421.

GcsiInit

Initialize the library. This function must be called before all other GCSI functions.

Syntax

void **GcsiInit**(void);

GcsiStatsForGemSessionId

Get the cache statistics for the given Gem session id.

Syntax

```
int GcsiStatsForGemSessionId(
    int                 sessionId;
    GcsiResultSType *   result);
```

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 "GCSI Errors" on page 421.

See Also

GcsiStatsForGemSessionWithName on page 416
GcsiStatsForPgsvrSessionId on page 417
GcsiStatsForProcessId on page 418
GcsiStatsForShrPcMon on page 419
GcsiStatsForStone on page 420

GcsiStatsForGemSessionWithName

Get the cache statistics for the first Gem in the cache with the given cache name.

Syntax

```
int GcsiStatsForGemSessionWithName(
    char *                      gemName;
    GcsiResultSType *           result);
```

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 "GCSI Errors" on page 421.

See Also

GcsiStatsForGemSessionId on page 415
GcsiStatsForPgsvrSessionId on page 417
GcsiStatsForProcessId on page 418
GcsiStatsForShrPcMon on page 419
GcsiStatsForStone on page 420

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

```
int GcsiStatsForPgsvrSessionId(
    int                     sessionId;
    GcsiResultSType *       result);
```

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.ht`; see "GCSI Errors" on page 421.

See Also

GcsiStatsForGemSessionId on page 415
GcsiStatsForGemSessionWithName on page 416
GcsiStatsForProcessId on page 418
GcsiStatsForShrPcMon on page 419
GcsiStatsForStone on page 420

GcsiStatsForProcessId

Get the cache statistics for the given process ID.

Syntax

```
int GcsiStatsForProcessId(
    int                    pid;
    GcsiResultSType *      result;
```

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.ht`; see “GCSI Errors” on page 421.

See Also

GcsiStatsForGemSessionId on page 415
GcsiStatsForGemSessionWithName on page 416
GcsiStatsForPgsvrSessionId on page 417
GcsiStatsForShrPcMon on page 419
GcsiStatsForStone on page 420

GcsiStatsForShrPcMon

Get the cache statistics for the shared page cache monitor process for this shared page cache.

Syntax

int **GcsiStatsForShrPcMon**(
    GcsiResultSType * *result*);

Arguments

result	Pointer to a GcsiResultSType structure.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in `gcsierr.ht`; see “GCSI Errors” on page 421.

See Also

GcsiStatsForGemSessionId on page 415
GcsiStatsForGemSessionWithName on page 416
GcsiStatsForPgsvrSessionId on page 417
GcsiStatsForProcessId on page 418
GcsiStatsForStone on page 420

GcsiStatsForStone

Get the cache statistics for the Stone if there is a Stone attached to this shared page cache.

Syntax

int **GcsiStatsForStone**(
    GcsiResultSType *   *result*);

Arguments

result	Pointer to a GcsiResultSType structure.

Return Value

Returns 0 if successful; otherwise returns an error code, as defined in `gcsierr.ht`; see “GCSI Errors” on page 421.

See Also

GcsiStatsForGemSessionId on page 415
GcsiStatsForGemSessionWithName on page 416
GcsiStatsForPgsvrSessionId on page 417
GcsiStatsForProcessId on page 418
GcsiStatsForShrPcMon on page 419

GCSI Errors

The following errors are defined for the GemStone C Statistics Interface.

Table 1 GCSI Errors

Error Name	Definition
GCSI_SUCCESS	The requested operation was successful.
GCSI_ERR_NO_INIT	GcsiInit() must be called before any other Gcsi functions.
GCSI_ERR_CACHE_ALREADY_ATTACHED	The requested shared cache is already attached.
GCSI_ERR_NOT_FOUND	The requested session or process was not found.
GCSI_ERR_BAD_ARG	An invalid argument was passed to a Gcsi function.
GCSI_ERR_CACHE_CONNECTION_SEVERED	The connection to the shared cache was lost.
GCSI_ERR_NO_STONE	Stone statistics were requested on a cache with no stone process.
GCSI_ERR_CACHE_NOT_ATTACHED	No shared page cache is currently attached.
GCSI_ERR_NO_MORE_HANDLES	The maximum number of shared caches are attached.
GCSI_ERR_CACHE_ATTACH_FAILED	The attempt to attach the shared cache has failed.
GCSI_ERR_WATCHER_THREAD_FAILED	The cache watcher thread could not be started.
GCSI_ERR_CACHE_WRONG_VERSION	The shared cache version does not match that of the libgcsi*xx*.so library.