

Geodesic Systems, Inc.
© Copyright 1994-2002



Programmer's Reference Version 6.0

Table of Contents

INTRODUCTION	3
Integration with User's Guide.....	3
Using this Reference.....	3
GETTING STARTED	5
Introduction.....	5
Quick Reference.....	6
Export Control Classification Number	6
Enabling Geodesic Runtime	7
The Geodesic Runtime Product License.....	9
PLATFORM SPECIFIC INFORMATION	10
HP-UX	11
IBM AIX.....	13
Linux.....	16
Microsoft Windows.....	18
Sun Solaris.....	20
TUNING RUNTIME'S PERFORMANCE	22
When to Tune Runtime's Performance.....	22
When Not to Tune Runtime's Performance	23
How to Tune Runtime's Performance	23
Controlling When Garbage Collection Occurs	24
Telling Runtime Which Memory to Scan for Pointers.....	25
Tuning Runtime's Virtual Memory Performance	26
Ignoring Interior Pointers Beyond First Page	27
Multi-Threading	28
Mixing Automatic and Manual Memory Management	29
TUNING RUNTIME'S RELIABILITY	32
Automatically Fixing Premature Frees.....	32
Intercepting the C Run-Time DLLs and the Microsoft Windows API	33
PROGRAMMER'S GUIDE	34
Configuring Geodesic Runtime	34
Geodesic Runtime Object File.....	37
The Geodesic Runtime Library	38
Geodesic Runtime Header File.....	39
Geodesic Runtime and the Standard Memory Management Primitives.....	40
Geodesic Runtime Restrictions	43

API REFERENCE.....	45
Geodesic Runtime Types	45
Geodesic Runtime Macros.....	46
Geodesic Runtime Functions	47
Deprecation Information	74
TROUBLESHOOTING GUIDE.....	75
Troubleshooting Injecting with Geodesic Runtime	75
Troubleshooting Linking with Geodesic Runtime	77
Troubleshooting Running With Geodesic Runtime	79
APPENDIX	82
Using Geodesic Runtime with Third-Party Tools and Libraries.....	82
NOTICES.....	89
Proprietary Rights.....	89
INDEX	90

Introduction

INTEGRATION WITH USER'S GUIDE

The user's guide contains all the information necessary to get Runtime installed and running with your application. This reference will help you integrate Runtime with your application via the code. Some headings are duplicated in this reference as in the user's guide – those sections contain expanded information.

USING THIS REFERENCE

Throughout the manual, certain typefaces and icons are used to make the information specific to your needs pop out easily.

Typefaces

.Most of the text is in the font of this sentence.

Programming language, filenames, command line text, function calls or parameters of a function are shown in the font of this sentence.

Icons

Operating Systems

The following icons are used to denote platform specific comments.



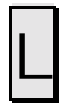
Unix



HP-UX



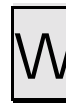
IBM AIX



Linux



Sun Solaris



Microsoft
Windows

Notes



Note – just a basic note to the user.



C++ - the comment is for C++ programs only.



Warning – take heed to the comment.



SMP – this comment or function relates to the SMP product only.



Advanced – this comment or function is for advanced users.

Getting Started

INTRODUCTION

Geodesic Runtime brings speed and reliability to under-performing applications. It can instantly turn a hopelessly unstable application into a bulletproof system. It helps servers handle unexpected spikes in user demand and it can make any program work better and with lower memory requirements, all in real time.

Since you can't find or fix every problem before your software ships, Runtime can fix them while your deployed program runs. With our exclusive Runtime Diagnosis and Remediation™ technology, Runtime transparently detects and resolves memory errors in a running program. Runtime automatically analyzes your program's data and returns unused memory to the operating system.

By injecting or linking your program, Runtime can automatically analyze data structures and identify allocated memory that is no longer being referenced. When Runtime identifies such leaked data, it safely frees it, protecting your program against memory leaks and preventing catastrophic system crashes.

The SMP version of Runtime is not only thread-safe but also thread-efficient. It achieves this by using independent heaps. Different threads can automatically allocate from different heaps and still safely free blocks allocated in another thread. This means you can call `malloc()` and `free()` from competing threads with minimal lock contention.

Runtime uses an extremely high-performance allocator that is much faster than other allocators, giving you an additional benefit of your program running faster under Runtime. In addition, Runtime's allocator is highly resistant to fragmentation, allowing you to write and run long-running programs without worrying about memory fragmentation. Runtime is completely customizable.

QUICK REFERENCE

To use Runtime, follow these steps. Details for each step are in the “Platform Specific Information” section for your operating system, starting on page 10:

- 1) Install Runtime as described in the Installation section specific to your platform.
- 2) Determine which library you want to use. See the list of libraries and object files for your platform.
- 3) Determine whether you want to use Runtime by injection or linking. Use the table under “Enabling Geodesic Runtime” on page 7 to make your decision.
 - a) Injection is described in “Injecting Runtime into Your Program” on page 7.
 - b) To link your application with Runtime, see “Linking Runtime to Your Program” on page 8.
 - c) Be sure to incorporate the Runtime license described in “The Geodesic Runtime Product License” on page 9.



If you are developing a shared library or DLL, be sure to link or inject Runtime into the main executable of the process that uses the DLL.

EXPORT CONTROL CLASSIFICATION NUMBER

Runtime components have an Export Control Classification Number (ECCN) of EAR99. This is the least restrictive classification. Some users will need to know this for shipping with our deployment libraries. (If you aren’t deploying outside of the United States, you won’t need this information.)

ENABLING GEODESIC RUNTIME

There are two methods to use Runtime with your application: injecting and linking. Usually, the decision between the two methods depends on whether or not you have access to the source or object code. If you have access to the code, link Runtime. If you do not have access to the code, inject the Runtime libraries.

The differences between linking and injecting are as follows:

	Injecting	Linking
Use of...		
Runtime APIs		X
Runtime Environment Variables	X	X
Runtime C Source License		X
Runtime Text License	X	X
Static Runtime Libraries		X
Shared Runtime Libraries	X	X
Application is linked with Static C-Run-Time		X
Child processes use Runtime		
forked process (always)	X	X
exec process (user decision)	X	
Runtime becomes part of executable		X
Deployment		
Requires specific Runtime library be distributed	X	X
Requires additional setup on customer's end	X (note 2)	(note 3)



1. The HP injector does not require additional setup on the customer's end.
2. Linking with MS Windows does require additional setup on the customer's end.

Injecting Runtime into Your Program (Recommended)

You should use injection if you want to use Runtime with a third-party application for which you lack access to the source or object code. You may use injection with applications you have developed, but it may be easier simply to relink with Runtime (see "Enabling Geodesic

Runtime” on page 7 or your specific operating instructions in the appendix). You cannot use injection if you intend to distribute your application with Runtime.



Under injection, only the environment variables are available; you will not be able to use the Runtime functions. To control their behavior, use the environment variables described under “Geodesic Runtime Environment Variables” in the user’s guide.

Linking Runtime to Your Program

Introduction

The methods of linking vary by operating system and compiler. Linking the Runtime libraries into an application allows the use of Runtime’s garbage collection and fast memory allocation features. This improves the performance of the application and protects it against memory related failures and data corruptions.

Linking attaches the Runtime libraries to the user’s application so that Runtime can perform its interception, diagnosis, remediation, monitoring, and enhancement services.

When to Use Linking

You can link Runtime when you have access to the source code and should if you intend to distribute your application with Runtime.

Linking Instructions

For details, see your specific platform instructions:

For HP-UX, see page 11.

For IBM AIX, see page 13.

For Linux, see page 16.

For Microsoft Windows, see page 18.

For Sun Solaris, see page 20.

THE GEODESIC RUNTIME PRODUCT LICENSE

Runtime product licenses come in two different forms. The following table shows the different licenses and their uses; you need only select one type of license.

Product License	Compile / Link for Own Use	Compile / Link for Distribution	Injection
Plain text file (Recommended)	X		X
C source file	X	X	

Text Licenses (Recommended)

The text license required for injection is not the same as any C source file license you may have been provided with for linking with Runtime libraries. A text license file will be composed only of key/value pairs, with some comment lines.

You can place this file anywhere you choose, but you must register its name and location using the `GS_LICENSE_FILE` environment variable, see the user's guide.



If you do not set this variable, Runtime will look for `gslicense.txt` in the system directory (e.g. `c:\winnt\system32`).



On Unix, except for HP-UX, when using injection, the `-c` option can also be used to specify the location of the text license file. See the appropriate appendix for your operating system for more information.

C Source Licenses

Text licenses will work with both injection and linking. Use the source file only if your usage situation requires it. To link Runtime with your application, you can build your application with the Runtime C source license, the appropriate Runtime library, and the Runtime object file.

Platform Specific Information

The section contains instructions and information specific to each operating system we support.

- HP-UX, page 11
- IBM AIX, page 13
- Linux, page 16
- Microsoft Windows, page 18
- Sun Solaris, page 20

HP-UX



Footprint reduction is not supported on the HP-UX platform.

Libraries and Object Files

Below is a guide to the distributed library files. The object file is `gs.o` and must be linked in addition to the library.

The libraries have the threading type embedded in the name. Each library name, before the extension, will be postfixed with `_st` (single-threaded), `_mt` (multi-threaded), or `_smp` (SMP tuned) designating the threading type.

Product	Single-Threaded	Multi-Threaded	SMP
Ha_runtime_up			
Static	<code>libha_st.a</code>	<code>libha_mt.a</code>	
Shared	<code>libha_st.sl</code>	<code>libha_mt.sl</code>	
Ha_runtime_smp			
Static			<code>libha_smp.a</code>
Shared			<code>libha_smp.sl</code>

Linking Runtime

Runtime should only be linked into the finished application, not into individual libraries that go into your executable.

- 1) Open the Makefile for your application
- 2) Locate your Makefile's linking directive
- 3) Add the Runtime Files:
 - a) Add the object file `gs.o` to the link line.
 - b) Add the appropriate Runtime library to the link line, after user- and third-party libraries and before the C-runtime library. See "Libraries and Object Files" on page 11 for a list of available libraries.



We recommend you use the Runtime shared library.



Since UNIX uses a POSIX standard, a 1-pass linker, our library needs to be placed before the C-runtime library in the link line. As long as it stays before the C-runtime library, it should be placed as late in the library list in the link line as possible. This is so that user and third party libraries may compute their links without the interference of our library.

- c) The `-lpthread` option must also be specified to the linker for multi-threaded and SMP libraries.

For example, if you want to link your program with the multi-threaded, shared version of the High Availability Runtime library, the link stage of your compilation should look something like this:

```
% aCC -o my_app -lpthread [other flags] [object files] gs.o  
gslicense.o [other libraries] <ROOT_DIR>/lib/libha_mt.sl
```



If you used a static version of Runtime's multi-threaded or SMP library, you must add the `-lrt` option to the command line in addition to the `-lpthread` option.

- 4) You may verify that the application has been linked properly by using the `nm` utility, for example:

```
% nm my_app | grep "gsMalloc"
```

- a) Output results: application is linked properly.
b) No output results: application is not linked properly.
- 5) Set `GS_LICENSE_FILE` to the location of your text license. See "The Geodesic Runtime Product License" on page 9 for more information.

IBM AIX

Libraries and Object Files

Below is a guide to the distributed library files. The object file is `gs.o` and must be linked in addition to the library.

The libraries have the threading type embedded in the name. Each library name, before the extension, will be postfixed with `_st` (single-threaded), `_mt` (multi-threaded), or `_smp` (SMP tuned) designating the threading type.

Product	Single-Threaded	Multi-Threaded	SMP
Ha_runtime_up Static Shared	libha_st_stat.a libha_st.a	libha_mt_stat.a libha_mt.a	
Ha_runtime_smp Static Shared			libha_smp_stat.a libha_smp.a

Linking Runtime

Runtime should only be linked into the finished application, not into individual libraries that go into your executable.

Prior to using linking the first time, you'll need to have run the `makeintercept` script for AIX, `<ROOT_DIR>/etc/intercept/makeintercept`.

- 1) Open the Makefile for your application
- 2) Link with the Runtime files
 - a) In the link command for your application, link your program with the Runtime object file and the appropriate library. If you are not sure which libraries are available, see "Libraries and Object Files" on page 13.



We recommend you use the Runtime shared library.

- b) When using the Runtime shared library, add `-bexpall` to the link line. This allows Runtime to see the symbols in the license file.



If you don't wish to export all symbols with `-bexpall`, create a file `<export_file>` that contains one line:

```
GeodesicLicenses
```

Replace “`-bexpall`” with “`-bE:<export_file>`”.

The name of the `<export_file>` doesn't matter. We often use `gslicense.exp`.

- c) (optional) Set the location of the libraries with `LIBPATH` on the command line. Use `-blibpath:<location>`.

For example, if you want to link your program with the multi-threaded, shared version of High Availability Runtime library:

```
xlc_r -o appname <sources> <ROOT_DIR>/gs.o  
/<ROOT_DIR>/lib/libha_mt.a -bexpall  
-blibpath: /<ROOT_DIR>/lib:/usr/lib:/usr/lib/threads:.
```

3) Link with the Runtime C run-time libraries (recommended)

- a) For programs that use `mmap()`, `shmat()`, `munmap()`, `shmdt()` or store pointers in memory spaces acquired by these functions, link with the Runtime C run-time libraries `libc` or `libc_r` in `<ROOT_DIR>/lib` that were created at installation. To do this, add to your link line:

```
-L/<ROOT_DIR>/lib
```

- b) Use the following steps to confirm that your application is correctly using the Runtime C run-time library:

i) Load the application into the dbx debugger: `dbx <appname>` or `dump -Hv <appname>`

ii) Run the map command within dbx: `dbx> map`

iii) Check that `/<ROOT_DIR>/lib/libc.a` or `/<ROOT_DIR>/lib/libc_r.a` is listed somewhere in the map output.

- 4) Set `GS_LICENSE_FILE` to the location of your text license. See “The Geodesic Runtime Product License” on page 9 for more information.



Using `-bilibpath` is an alternative to using the `LIBPATH` environment variable. This way, multiple programs can have different libpaths.

If the `LIBPATH` environment variable is used, it must be used at run-time. `<ROOT_DIR>/lib` directory must be specified in it before the `/usr/lib` directory.

If this is not done, it may result in a loader error at run-time.



If the application is multi-threaded, the environment variable `AIXTHREAD_SCOPE` must be set to `S`.

If this is not done, it may result in multi-threaded applications hanging.



If you are deploying your application with Runtime and have linked with the Runtime C run-time libraries, you must also redistribute the files in `<ROOT_DIR>/etc/intercept` and execute the `makeintercept` script upon installation of your software.



Footprint reduction will only work if `GS_USE_MMAP` is set to 1. At this time, this will reduce the speed of your application.

LINUX

Libraries and Object Files

Below is a guide to the distributed library files. The object file is `gs.o` and must be linked in addition to the library.

The libraries have the threading type embedded in the name. Each library name, before the extension, will be postfixed with `_st` (single-threaded), `_mt` (multi-threaded), or `_smp` (SMP tuned) designating the threading type.

Product	Single-Threaded	Multi-Threaded	SMP
Ha_runtime_up			
Static	<code>libha_st.a</code>	<code>libha_mt.a</code>	
Shared	<code>libha_st.so</code>	<code>libha_mt.so</code>	
Ha_runtime_smp			
Static			<code>libha_smp.a</code>
Shared			<code>libha_smp.so</code>

Linking Runtime

Runtime should only be linked into the finished application, not into individual libraries that go into your executable.

- 1) Open the Makefile for your application
- 2) Locate your Makefile's linking directive
- 3) Add the Runtime Files:
 - a) Add the license file, `gs.o`, prior to the Runtime library in the link line.
 - b) Add the appropriate Runtime library to the link line, after user- and third-party libraries and before the C-runtime library. If you are not sure which libraries are available, see "Libraries and Object Files" on page 16.



We recommend you use the Runtime shared library.



Since UNIX uses a POSIX standard, a 1-pass linker, our library needs to be placed before the C-runtime library in the link line. As long as it stays before the C-runtime library, it should be placed as late in the library list in the link line as possible. This is so that user and third party libraries may compute their links without the interference of our library.

- c) The `-lpthread` option must also be specified to the linker for multi-threaded and SMP libraries.
- d) Add `-ldl` to your command line.

For example, if you want to link your program with the multi-threaded, shared version of the High Availability Runtime library, the link stage of your compilation should look something like this:

```
% g++ -o my_app -lpthread [other flags] [object files] gs.o
gslicense.o [other libraries] <ROOT_DIR>/lib/libha_mt.so
-ldl
```

- 4) You may verify that the application has been linked properly by using one of the following utilities.
 - a) If using shared libraries, use the `ldd` utility. For example:

```
% ldd my_app
```

 - i) If see the Runtime library among the shared library dependencies, application is linked properly.
 - ii) If you don't see the Runtime library among the shared library dependencies, application is not linked properly.
 - b) If using static libraries, use the `nm` utility. For example:

```
% nm my_app | grep gsMalloc
```

 - i) Output results: application is linked properly.
 - ii) No output results: application is not linked properly.
- 5) Set `GS_LICENSE_FILE` to the location of your text license. See “The Geodesic Runtime Product License” on page 9 for more information.

MICROSOFT WINDOWS

Libraries and Object Files

Below is a guide to the distributed library files. The object file is `gs.obj` and must be linked in addition to the library.

The libraries have the threading type embedded in the name. Each library name, before the extension, will be postfixed with `_st` (single-threaded), `_mt` (multi-threaded), or `_smp` (SMP tuned) designating the threading type.

(shared libraries only)

Product	Single-Threaded	Multi-Threaded
Ha_runtime_up MSVC6.DLL	ha<VERSION>6s.dll libha_st.lib	ha<VERSION>6t.dll libha_mt.lib
		SMP
Ha_runtime_smp MSVC6.DLL		ha<VERSION>6p.dll libha_smp.lib

<VERSION> = Current released version of Runtime, for example, 6.0.1.5.

Linking Runtime

Runtime should only be linked into the finished application, not into individual libraries that go into your executable.

- 1) Open your project in Microsoft Visual C++
- 2) Add Runtime files to your project
 - a) Add files by right clicking on the name of your project in the project pane and then selecting **Add Files to Project** from the menu. This brings you to the **Insert Files into Project** dialog.

To find the correct Runtime libraries for your application, refer to “Libraries and Object Files” on page 18.

For example, if you installed High Availability Runtime and your application is multi-threaded and built under Visual C++ 6, you would choose

`<ROOT_DIR>\lib\libha_mt.lib`.

- b) To use Runtime, first type `*.*` in the File Name field (so you can see all available files), and then use the CTRL key to place the following files, in order, at the top of your list:

`gs.obj`

Runtime library as per the table in “Libraries and Object Files” on page 18.



The Runtime library and object file must be the first items in your list of linked files. In Project, Settings, Link, Object/Library Modules the files `gs.obj` and the appropriate Runtime library should be the first in the list, in that order.

- 3) Click OK.
- 4) To see if your program was linked with Runtime, you can use the `dumpbin` utility provided with Microsoft Visual Studio. Check the import table of your executable to see if it is importing Runtime symbols:

```
C:> dumpbin /imports myExe > imports.txt
```

In the file, there should be an import entry for the Runtime library; it should list symbols such as `malloc()`, `free()`, `operator new`, or `operator delete`.

- 5) Set `GS_LICENSE_FILE` to the location of your text license. See “The Geodesic Runtime Product License” on page 9 for more information.

SUN SOLARIS

Libraries and Object Files

Below is a guide to the distributed library files. The object file is `gs.o` and must be linked in addition to the library.

The libraries have the threading type embedded in the name. Each library name, before the extension, will be postfixed with `_st` (single-threaded), `_mt` (multi-threaded), or `_smp` (SMP tuned) designating the threading type.

Product	Single-Threaded	Multi-Threaded	SMP
Ha_runtime_up			
Static	<code>libha_st.a</code>	<code>libha_mt.a</code>	
Shared	<code>libha_st.so</code>	<code>libha_mt.so</code>	
Ha_runtime_smp			
Static			<code>libha_smp.a</code>
Shared			<code>libha_smp.so</code>

Linking Runtime

Runtime should only be linked into the finished application, not into individual libraries that go into your executable.

- 1) Open the Makefile for your application
- 2) Locate your Makefile's linking directive
- 3) Add the Runtime Files:
 - a) Add the license file, `gs.o`, prior to the Runtime library in the link line.
 - b) Add the appropriate Runtime library to the link line, after user- and third-party libraries and before the C-runtime library. If you are not sure which libraries are available, see "Libraries and Object Files" on page 20.



We recommend you use the Runtime shared library.



Since UNIX uses a POSIX standard, a 1-pass linker, our library needs to be placed before the C-runtime library in the link line. As long as it stays before the C-runtime library, it should be placed as late in the library list in the link line as possible. This is so that user and third party libraries may compute their links without the interference of our library.

- c) The `-lpthread` option must also be specified to the linker for multi-threaded and SMP libraries.
- d) Add `-ldl` to your command line.

For example, if you want to link your program with the multi-threaded, shared version of the High Availability Runtime library, the link stage of your compilation should look something like this:

```
% CC -o my_app -lpthread [other flags] [object files] gs.o
gslicense.o [other libraries] <ROOT_DIR>/lib/libha_mt.so
-ldl
```

- 4) You may verify that the application has been linked properly by using the `nm` utility, for example:

```
% nm my_app | grep gsMalloc
```

- a) Output results: application is linked properly.
 - b) No output results: application is not linked properly.
- 5) Set `GS_LICENSE_FILE` to the location of your text license. See “The Geodesic Runtime Product License” on page 9 for more information.



By default, Solaris uses lazy linking. Use `LD_BIND_NOW` to force non-lazy binding. See the troubleshooting item “I’m getting errors such as `ld.so.1: ld: fatal: relocation error: symbol not found:`” on page 80 for details.

Tuning Runtime's Performance

Full information regarding tuning Runtime's performance can be found in the user's guide. This section contains additional information for those people working with the code directly.

- “When to Tune Runtime's Performance” on page 22
- “When Not to Tune Runtime's Performance” on page 23
- “How to Tune Runtime's Performance” on page 23
- “Controlling When Garbage Collection Occurs” on page 24
- “Telling Runtime Which Memory to Scan for Pointers” on page 25
- “Tuning Runtime's Virtual Memory Performance” on page 26
- “Ignoring Interior Pointers Beyond First Page” on page 27
- “Multi-Threading” on page 28
- “Mixing Automatic and Manual Memory Management” on page 29

WHEN TO TUNE RUNTIME'S PERFORMANCE

If you have converted a pre-existing program to use Runtime's automatic memory management, you can optimize its operation by removing the old code for managing memory from the program. Although old memory management logic is harmless, it simply wastes time when you are using automatic memory management.



Optimizing Runtime's operation must be done accurately. The internal operation of Runtime is already highly optimized. Consequently, the main opportunity for optimizing Runtime's performance is to instruct it about the particulars of your program's use of memory. By forcing you to manually describe specific aspects of your program's memory usage, these optimizations introduce some of the disadvantages normally associated with manual memory management, such as increased opportunity for programmer error or loss of encapsulation and reuse.



By default, Runtime does not call an object's destructor when automatically reclaiming its memory. For this reason, don't remove existing calls to `delete` from your programs until you have checked for required destructors. Many destructors simply release memory and don't need to be called when Runtime is automatically managing memory. If a heap object needs its destructor called, manually delete the object.

WHEN NOT TO TUNE RUNTIME'S PERFORMANCE

- If your program is not yet operating correctly, don't spend time optimizing it. Optimization can only make a program run more efficiently; it can't fix a program's behavior.
- If you haven't yet fully optimized your program's logic and data structures, you should optimize them before attempting to optimize Runtime's operation. Because Runtime is rarely the limiting factor in a program's performance, you can almost always achieve much greater efficiency improvements to your program by focusing optimization effort on your application's design, data structures, and algorithms.

HOW TO TUNE RUNTIME'S PERFORMANCE

In addition to tuning Runtime's performance by controlling when garbage collection occurs, you can tell Runtime which memory to scan for pointers.



Do not attempt to make Runtime faster by mixing automatic and manual memory management. Runtime takes the same amount of time to operate whether it is managing all of your data or just some of it. This is because Runtime needs to scan all data, even manually managed data, for pointers to automatically managed data. The one exception is `gsMallocLeaf()`. See "Telling Runtime Which Memory to Scan for Pointers" on page 25 for details.

CONTROLLING WHEN GARBAGE COLLECTION OCCURS

Identifying points where your program can spare time for Runtime to analyze its data structures can provide significant improvements in Runtime's performance. For example, if your program displays a series of results in the screen, the user will probably spend several seconds viewing them. Calling for a garbage collection immediately after displaying the results would allow Runtime to perform garbage collection when your program is most likely idle.

Tell Runtime to perform a complete garbage collection cycle. Instruct Runtime to immediately perform a complete garbage collection cycle by calling `gsCollect()`. Call `gsCollect()` when your program can best spare time for Runtime to perform its analysis. Runtime analyzes 3 or 4 megabytes of memory per second on a low-end PC or workstation. Thus, if your program only uses a few megabytes of memory, `gsCollect()` will usually take less than one second to garbage collect your program. If your program uses more memory, the time spent in a collection cycle will be proportionally longer.



Tell Runtime how frequently to perform garbage collection cycles. In addition to the information in the user's guide, you can explicitly request garbage collection by calling `gsCollect()`.

TELLING RUNTIME WHICH MEMORY TO SCAN FOR POINTERS

To enhance performance, use `gsMallocLeaf()` to allocate memory that you know contains no pointers. For example, use `gsMallocLeaf()` to allocate string data. By allocating memory in this fashion, you inform Runtime that it does not need to scan the memory for pointers to other data.

Using `gsMallocLeaf()` to allocate memory that does not contain pointers makes programs more efficient in two ways:

- 1) Runtime does not need to scan this memory for pointers, reducing the time required for a garbage collection cycle. For programs that primarily allocate pointer-free memory, the improvement to Runtime's performance can be quite significant. For example, if 75% of a program's data consists of strings, Runtime will garbage collect roughly four times as fast if the strings are allocated with `gsMallocLeaf()`.
- 2) Allocating pointer-free memory with `gsMallocLeaf()` dramatically improves Runtime's virtual memory performance as described in the heading "Tuning Runtime's Virtual Memory Performance" below by reducing the amount of memory to be scanned and collected.



Only use `gsMallocLeaf()` to allocate memory that does not contain pointers. If you inadvertently use `gsMallocLeaf()` to allocate memory that contains pointers, these pointers will be hidden from Runtime. As a result, Runtime may erroneously free data used by this memory. More precisely, if you use `gsMallocLeaf()` to allocate memory that contains pointers, any automatically managed data accessible from those pointers must also be protected by pointers that are not hidden from Runtime.



Programming Suggestion:

If you have a class that contains no pointers, declare a class-specific operator `new()` that uses `gsMallocLeaf()` to allocate memory. While Runtime does not permit you to use third-party allocators to allocate automatically managed memory (see "Geodesic Runtime Restrictions" on page 43), a class-specific operator `new()` that uses `gsMallocLeaf()` does not violate this restriction because `gsMallocLeaf()` is built into Runtime.

TUNING RUNTIME'S VIRTUAL MEMORY PERFORMANCE

Tell Runtime which memory it needs to scan. See “Telling Runtime Which Memory to Scan for Pointers” on page 25 for information on how to do this. This optimization provides by far the greatest improvement to Runtime’s virtual memory performance.

IGNORING INTERIOR POINTERS BEYOND FIRST PAGE

Very large objects (over 64 KB) may be unnecessarily retained by Runtime. If there is any word of live memory that points into the object, Runtime will retain the object because that word may later be cast into a pointer, even though this is usually unlikely. This is not an issue for objects smaller than about 64 KB because Runtime takes steps to minimize collisions.

You can optimize Runtime's space usage by allocating very large objects with `gsMallocIgnoreOffPage()`. Such memory will only be retained if there is a live pointer that points into the first logical page (8192 bytes, which may be larger than a physical page on your platform) of the object. As this is nearly universally true in practice, Runtime will be able to aggressively reclaim the object when it is no longer in use.

If the function `gsSetIgnoreOffPagePointers()` has been set to `GS_ON` or the environment variable `GS_IGNORE_OFF_PAGE_POINTERS=1`, then `malloc()` and `new` will automatically use `gsMallocIgnoreOffPage()` to allocate objects larger than 64 KB. Allocation of smaller objects is not affected. You can customize the threshold value (from 64kb) for switching to `gsMallocIgnoreOffPage()` by calling the function `gsSetVeryLargeAllocationSize()`.

MULTI-THREADING

When to Use Multi-Threading

Runtime's automatic memory management makes memory management easier in multi-threaded programs:

- Use Runtime to manage shared data. Heap data that is shared among threads is traditionally very difficult to manage by hand. Because threads are scheduled by the operating system, it is often impossible to know at compile time which thread will be the last to use the data. Runtime will automatically manage the shared data correctly.
- Using Runtime to manage shared data is a major improvement over the traditional approach of reference counting the shared data. Compared to Runtime's automatic memory management, reference counting is awkward to use, yields poor performance, and has inherent bugs and limited semantics.

Tuning Multi-Threading

Because Runtime needs to acquire a lock for each allocation in a multi-threaded program, memory allocation is slower than in single-threaded operation. The following advanced technique reduces lock acquisition overhead:



Allocate many objects at once. Use `gsMallocMany(s)` to allocate a pool of objects of size `s`. This lets a thread reduce Runtime's overhead by allocating a number of objects at once under a single lock. Your program can then extract objects from the pool when it needs them. Detailed instructions and an example of using `gsMallocMany()` are described in the programmer's manual.

MIXING AUTOMATIC AND MANUAL MEMORY MANAGEMENT



While it is possible to mix automatic and manual memory management, we strongly discourage you from doing so. If, however, you feel your program needs it, this section will help you.

When using the High Availability library, all your program's memory is automatically managed. This section explains how to use Runtime to mix automatic and manual memory management. Runtime lets you use as much or as little automatic memory management as you'd like.

This flexibility respects the C and C++ philosophy of never forcing you to use some "feature" whether or not it is appropriate to your situation.

When to Mix Automatic and Manual Memory Management

- Runtime does not make you fix what isn't broken. If you have a program that generally manages memory well but has some data that is awkward to manage correctly, you may tell Runtime to manage only that data.
- If your program allocates memory that does not contain pointers, you can use `gsMallocLeaf()` to allocate the memory. This tells Runtime to not scan that memory for pointers. See "Telling Runtime Which Memory to Scan for Pointers" on page 25 for detailed instructions.
- Runtime does not allow you to supply your own memory allocator for automatically managed data. Runtime almost always makes outside allocators unnecessary because its built-in allocator offers extremely high performance by using fixed-size allocators for small objects and a sophisticated cache locality strategy. However, if you choose to use your own allocator for some of your data, you must manually manage this data.
- You may have a data structure that violates the correct use rules for Runtime given in "Geodesic Runtime Restrictions" on page 43. This is extremely rare, but if it happens, you must either make that type conform to Runtime's restrictions or manually manage that type.
- As much as we hate to admit it, you may find bugs in Runtime, or Runtime may expose bugs in your compiler. (If you suspect you have found such a bug, please contact us.) When this happens, you may need to manually manage the data that gives Runtime trouble. The ability to mix automatic and manual memory management keeps workarounds as non-intrusive as possible.
- You may also have your own uses for combining garbage collection with manual memory management.

When Not to Mix Automatic and Manual Memory Management

Runtime's automatic memory management is reliable, efficient, and easy. Automatically managing only some of a program's data usually is *not* more efficient than automatically managing all its data because Runtime scans the entire non-collected heap by default for pointers to garbage collected objects.



Bypassing Runtime's scan of the non-collected heap by using your own allocator as described in the entries `malloc()`, `calloc()`, and `realloc()` in the programmer's manual is possible but can be error-prone.

How to Mix Automatic and Manual Memory Management

You can tell Runtime to automatically manage only a part of your program's memory while continuing to use traditional memory management on the rest of your program's data.

Follow these steps to mix automatic and manual memory management:

- 1) include `gst.h` in your application.
- 2) Allocate manually managed memory with `gsMallocManual()`. Manually managed memory must be released by explicitly calling `free()` or `delete`.

Programming Suggestion—Allocate data with a class-specific operator `new` or a “`new_function`”

More likely than not, you will identify particular types that you want to be manually managed.

C++ Programs

You can create a class-specific operator `new` for allocating that type that calls `gsMallocManual()`:

```
class S {
    void *operator new(size_t s) {
        return gsMallocManual(s);
    }
};

// Manually managed object.
// Be sure to delete when done.
S *sp = new S;
```

C Programs

You can create a “new_function” for allocating a particular type.

For example, if you wanted Runtime to manually manage objects of type `struct S`, you would allocate objects of this type with `gsMallocManual()`. You could do this explicitly every time you allocate a `struct S`, but it would be easy to forget and accidentally sometimes call `malloc()`. Instead, create a function `new_S()` and use it for allocating `struct S` objects as follows:

```
struct S *new_S() {
    return (struct S *)gsMallocManual(sizeof struct S);
}
...
/* Manually managed object. */
/* Be sure to free when done. */
struct S *sp = new_S();
```

Tuning Runtime's Reliability

Full information regarding tuning Runtime's reliability can be found in the user's guide. This section contains additional information for those people working with the code directly.

“Automatically Fixing Premature Frees” on page 32

“Intercepting the C Run-Time DLLs and the Microsoft Windows API” on page 33

AUTOMATICALLY FIXING PREMATURE FREES

When Not to Automatically Eliminate Premature Frees

You may want to free some objects manually to ensure they are released as quickly as possible. You may wish to free them with `gsFree()` to continue to benefit from protection against premature freeing of other objects. However, this feature should be used carefully because you will not be protected against premature calls to `gsFree()`.

How Automatically Fixing Premature Frees Works



If `realloc()` returns a new pointer, the memory of the old object will not be freed until Runtime ensures that it is safe to do so.

How To Automatically Fix Premature Frees



After setting `GS_DISABLE_FREE=1`, delete no longer frees memory, but it still invokes destructors. Runtime protects you from premature freeing of memory but will not necessarily protect you from premature invocation of destructors.

INTERCEPTING THE C RUN-TIME DLLS AND THE MICROSOFT WINDOWS API



This section applies to Microsoft Windows users only.

Runtime automatically intercepts calls to memory management functions in the C run-time DLLs and the Microsoft Windows API to properly scan their memory.

Intercepting the C Run-Time DLLs

Runtime automatically intercepts any memory management calls to the C run-time DLLs made by any module of the running program and replaces them with the corresponding Runtime functions. This feature is how Runtime fixes memory deallocation errors in DLLs used by your program. See “Using Geodesic Runtime with Third-Party Tools and Libraries” on page 82 for more on how DLLs interact with Runtime.

The Windows APIs

Runtime automatically scans memory allocated directly by the Windows API through such functions as `GlobalAlloc()`, `LocalAlloc()`, `SetWindowLong()`, `TlsAlloc()`, `HeapAlloc()`, `VirtualAlloc()`, and the Win32 Heap API. This memory is scanned for pointers but not garbage collected, so you must still call the appropriate API function to release the memory.

Benefits of Intercepting the C Run-Time DLLs and Windows API

By default, Runtime intercepts memory management calls to the C run-time DLLs and Windows API, providing you with a number of benefits:

- Runtime fixes memory deallocation errors in DLLs that your program uses. See “Using Geodesic Runtime with Third-Party Tools and Libraries” on page 82 for more information.
- Runtime’s awareness of the Windows API allows you to place pointers to automatically managed data in Windows data words and other places that bypass the Runtime allocator.

You may see a performance improvement with the Runtime allocator depending on your program.

Programmer's Guide

CONFIGURING GEODESIC RUNTIME

This section explains how to configure Runtime at program initialization.

Runtime has a number of functions that change its settings, such as `gsSetFixPrematureFrees()`, `gsSetScanAlignment()`, `gsSetZeroAllocatedObject()`, and `gsSetPrintStats()`. See “Geodesic Runtime Functions” on page 47 for a complete list.

There are two ways to configure Runtime:

- Configure Runtime with environment variables. See the user's guide for a full listing.
- Configure Runtime programmatically.

When to Configure Runtime

- `gsSetScanAlignment()` and other collector settings that should remain in effect throughout the entire operation of the program should be set at initialization.
- Some functions must be set in `gsInitialize()`. These are as follows:

<code>gsSetAddBytes()</code>	<code>gsSetAllowUserStacks()</code>
<code>gsSetDesktopInteraction()</code>	<code>gsSetFastMode()</code>
<code>gsSetLogAllLeaks()</code>	<code>gsSetLogFileName()</code>
<code>gsSetLogFilePath()</code>	<code>gsSetLogWithPID()</code>
<code>gsSetNumberOfHeaps()</code>	<code>gsSetPrintStats()</code>
<code>gsSetReportLeaksSwitch()</code>	<code>gsSetScanAlignment()</code>
<code>gsSetIgnoreOffPagePointers()</code>	<code>gsSetStopSignal()</code>
<code>gsSetVeryLargeAllocationSize()</code>	<code>gsSetZeroAllocatedObject()</code>

- The `gsInitialize()` function is a good default place to set any Runtime variables and call any Runtime functions that your program has no obvious best place to initialize, for example, `gsSetFixPrematureFrees()` and `gsSetCollectAtEnd()`. Runtime will find the `gsInitialize()` function in your executable or in any DLL, and it will always invoke it before doing any work.



If several `gsInitialize()` functions are defined, they will all be run in an undefined order; verify if this is what you want.

When Not to Use Runtime Configuration

- The default settings of Runtime are designed to be optimal for most programs; changing them may have subtle or unexpected consequences. Unless you are in an unusual situation, you may be better off letting Runtime make its own configuration decisions.
- If you want to change a setting while your program is running, you should make the change at that time. For example, you might tweak the collection policy only when your program is interacting with the user.
- Be cautious if you put your own initialization code commands in `gsInitialize()`, since it is intended specifically for Runtime settings. The C run-time library and memory allocation are not yet enabled at that point, so any code that allocates memory or calls the C or C++ standard libraries will give unpredictable results.

Configuring Runtime with Environment Variables

Many of Runtime's variables can be initialized with values taken from environment variables. Just define these variables in your environment before running the program. See "Geodesic Runtime Environment Variables" in the user's guide for a list of environment variables that Runtime recognizes.



If the program also sets the values programmatically, the environment variable is overridden by the code in the program.

Configuring Runtime Programmatically

It is often best, and sometimes essential, to make programmatic settings before Runtime initializes, so that Runtime respects them from the moment it starts running.

To initialize Runtime with particular settings:

- 1) Include `gst.h` in your program

2) Define a function named `gsInitialize()` with the expression

```
gsInitFunction
gsInitialize()
{
    /* Put initialization code here */
}
```



The function `gsInitialize()` is executed before Runtime is enabled and before the C run-time libraries are initialized. Do not put any code that calls C run-time library or memory allocation functions in `gsInitialize()`.



`gsInitFunction()` is a macro that simplifies the declaration of `gsInitialize()`. It is not a real type. Define `gsInitialize()` exactly as shown above. Do not return a value from `gsInitialize()`.

Example

Suppose you choose to add 10 bytes to the allocation requests. Define `gsInitialize()` in your program by:

```
gsInitFunction
gsInitialize()
{
    int status = gsSetAddBytes(10);
    assert (status == GS_SUCCESS);
    /*...*/
}
```

GEODESIC RUNTIME OBJECT FILE

Runtime comes with a single object file, `gs.obj` or `gs.o`. Users must link with this file to use Runtime. See “Libraries and Object Files” in the appendix for your operating system for platform- and compiler-specific details.

`gs.obj` OR `gs.o`

This is the Runtime object file. Whenever you link with Runtime, you will link with `gs.obj` in Microsoft Windows or `gs.o` in Unix as well as the Runtime library.

The object file should be specified on the link line before the Runtime library. The object file is not a replacement for the Runtime library.

THE GEODESIC RUNTIME LIBRARY

This section describes the Runtime libraries. You can refer to “Geodesic Runtime and the Standard Memory Management Primitives” on page 40 for detailed information about how `malloc()`, `calloc()`, `realloc()`, `free()`, `new`, `delete`, and `mallinfo()` are redefined by Runtime under these libraries.

The libraries have the threading type embedded in the name. Each library name, before the extension, will be postfixed with `_st` (single-threaded), `_mt` (multi-threaded), or `_smp` (SMP tuned) designating the threading type.

Basic use of the Runtime library is described in “Enabling Runtime” in the platform specific section for your operating system.



With each library, Runtime also includes a corresponding DLL. See “Libraries and Object Files” on page 18 for the name of the DLL corresponding to each library. You must distribute the DLL together with your application.

libha

The `libha` library, which ships with the High Availability Runtime product, will automatically manage all of a program’s memory. Using the appropriate `libha` library automatically and permanently fixes a program’s memory leaks. When you use the appropriate `libha` library with your application, you can program without calling `free()` or `delete` while otherwise programming normally. The High Availability Runtime product provides thread safety, high performance, and automatic memory management that is optimized for uniprocessor systems. The High Availability Runtime SMP product additionally provides thread efficiency for multi-processor systems.

GEODESIC RUNTIME HEADER FILE

Runtime comes with a header file, `gst.h`. All functions, classes, and other symbols used by Runtime are declared in `gst.h`.

`gst.h`

This is the Runtime header file. It declares all functions, classes, and other symbols used by Runtime. You do not need to include `gst.h` if you only use the standard C and C++ memory management primitives. You will only need to include `gst.h` if you use any Runtime functions, classes, or preprocessor directives. All the Runtime symbols explained in “Geodesic Runtime Functions” starting on page 47 and “Geodesic Runtime Macros” on page 46 are declared in `gst.h`.



Runtime comes with other headers `gst-*.h`. These are included by `gst.h` as appropriate. You do not need to include them in your program.

GEODESIC RUNTIME AND THE STANDARD MEMORY MANAGEMENT PRIMITIVES

This section explains how the standard C and C++ memory management functions `calloc()`, `delete`, `free()`, `malloc()`, `mallinfo()`, `new`, and `realloc()` are redefined by Runtime.

`calloc()`

Under the Runtime library, `calloc()` has its normal behavior. Memory allocated with `calloc()` will be initialized to zero. This memory is automatically managed and will be automatically freed by Runtime when no longer in use.

`delete`



Under the Runtime library, `delete` has its normal C++ behavior. `delete` calls its argument's destructor (if any) and frees its memory with `gsFree()`. If you call the function `gsSetFixPrematureFrees(GS_ON)` or set the environment variable `GS_DISABLE_FREE`, `delete` will call destructors but not release memory. This protects you from errors arising from freeing memory that is still in use and lets you reuse existing code easily but may increase space requirements. See the entry on `gsSetFixPrematureFrees()` in "Geodesic Runtime Functions" starting on page 47 for more information.

`free()`

Under the Runtime library, `free()` has its normal behavior and immediately releases any memory allocated by Runtime by default. If you call `gsSetFixPrematureFrees(GS_ON)` or set the environment variable `GS_DISABLE_FREE`, Runtime will ignore your calls to `free()`. This protects you from inadvertently freeing memory that is still in use and lets you reuse existing code easily but may increase space requirements. See the entry on `gsSetFixPrematureFrees()` in "Geodesic Runtime Functions" starting on page 47 for more information.

`malloc()`

Under the Runtime library, `malloc()` has its normal behavior. Memory allocated with `malloc()` is automatically managed and will be automatically freed by Runtime when no longer in use.

If `gsSetZeroAllocatedObject()` has been set to `GS_ON` or the environment variable `GS_ZERO_ALLOCATED_OBJECT` is set, then `malloc()` will return zeroed memory.

If the function `gsSetIgnoreOffPagePointers()` has been set to `GS_ON`, then `malloc()` will use `gsMallocIgnoreOffPage()` to perform allocations larger than 64 KB. You can change this size limit by redefining `gsSetVeryLargeAllocationSize()`.

`mallinfo()`



Unix only

Provides instrumentation describing space usage. It returns the `mallinfo` structure (defined in `malloc.h`) with the following members:

Member	Description
<code>arena</code>	The total memory allocated to the program by Runtime in bytes.
<code>ordblks</code>	The number of blocks of memory allocated to non-small objects
<code>smlblks</code>	The number of blocks of memory allocated to small object.
<code>hblks</code>	The number of block headers. A block header refers to space used to describe one of more blocks of memory.
<code>hblkhd</code>	The amount of memory allocated to block headers in bytes
<code>usmlblks</code>	The amount of memory in use by small object in bytes.
<code>fsmlblks</code>	The amount of memory allocated to, but unused by small objects in bytes.
<code>uordblks</code>	The amount of memory in use by non-small objects in bytes.
<code>fordblks</code>	The amount of memory allocated to, but unused by non-small objects in bytes.
<code>keepcost</code>	This value is always 0.

`new`



Under the Runtime library, `new` has its normal behavior, using the Runtime allocator. Memory allocated with `new` is automatically managed and will be automatically released by Runtime when no longer in use.

If the function `gsSetIgnoreOffPagePointers()` is set to `GS_ON`, then `new` will use `gsMallocIgnoreOffPage()` to perform allocations larger than 64 KB. You can change this size limit by redefining `gsSetVeryLargeAllocationSize()`.



If you redefine a global or class-specific operator `new()` under the Runtime library, you must perform the actual memory allocation with one of Runtime's memory allocation functions (functions whose names begin with `gsMalloc`). Do not use third-party allocators with the Runtime library. Although the Runtime library is designed to coexist with many third-party allocators so it can be used with object files of unknown origin, the Runtime library *always* performs best with its own allocator.

`realloc()`

Under the Runtime library, `realloc()` has its normal behavior. Memory allocated with `realloc()` is automatically managed and will be automatically freed by Runtime when no longer in use.

If you call the function `gsSetFixPrematureFrees(GS_ON)` or set the environment variable `GS_DISABLE_FREE`, `delete` will call destructors but not release memory. If `realloc()` returns a new pointer, the memory of the old object will not be freed until REMDI ensures it is safe to do so. This protects you from errors arising from freeing memory that is still in use and lets you reuse existing code easily but may increase space requirements. See the entry on `gsSetFixPrematureFrees()` in "Geodesic Runtime Functions" starting on page 47 for more information.

GEODESIC RUNTIME RESTRICTIONS

In order to correctly identify what memory is in use, Runtime is subject to the following restrictions:

- For Runtime to recognize that an object is in use, there must exist a word in memory that points inside the object. It doesn't matter if the pointer is explicitly stored as a C++ pointer; it could be stored in a union or even in an integer (although that is usually bad practice). However, Runtime will not recognize that an object is in use if the only pointer to it has been XOR'ed with another pointer, or if it is broken into separate bytes that are then rearranged. These activities create a situation where there is no word in memory that is a pointer to the object, so Runtime will not be able to determine that the object is in use.
- Do not use third-party allocators to allocate automatically managed memory. Runtime goes to great lengths to accommodate third-party allocators because it is designed to manage object code of unknown origin. However, Runtime will not automatically manage memory allocated by third-party allocators.
- If you use your own allocator for non-garbage collected memory, Runtime may not search that memory for pointers to garbage collected memory.
- Because Runtime treats every word of accessible memory as a potential pointer, it may fail to reclaim an object because some memory that is not being used as a pointer happens to contain the object's address by coincidence. Such a misidentification never results in reclamation of objects that are still in use. At worst, it can cause the unnecessary retention of a few objects, which will not affect your program's operation. An example of this is an object that contains a phone number. To the system, this looks similar to a pointer and may indeed resolve to another object's location.



This retention does not result in the accumulation of leaked memory because the number of unnecessarily retained objects is bounded over time.

- Runtime requires special treatment to garbage collect some libraries. See “Using Geodesic Runtime with Third-Party Tools and Libraries” on page 82 for more information.
- If you are using Runtime on code for which source is not available or for other legacy code, Runtime may not work correctly because it is possible that the code is violating some of the above Runtime restrictions.

Important Note for Scientific Programmers:

To simulate Fortran-style unit offset arrays (running from 1 to M, instead of 0 to M-1), some scientific programmers declare a normal array and then subtract 1 from it, to get a pointer to

an array whose “first” element has an index of 1. This workaround is endorsed in section 1.2 of *Numerical Recipes in C*¹:

“One problem is that many algorithms naturally like to go from 1 to M, not from 0 to M-1. Sure, you can always convert them, but they then often acquire a baggage of additional arithmetic in array indices that is, at best, distracting. It is better to use the power of the C language, in a consistent way, to make the problem disappear.

Consider

```
float b[4], *bb;  
bb=b-1;
```

The pointer `bb` now points one location before `b`. An immediate consequence is that the array elements `bb[1] . . . bb[4]` all exist. We will refer to `bb` as a *unit-offset* vector.”

Unfortunately, **this is illegal ANSI C, and it may break under Runtime** (or any modern allocator). You are pointing off the beginning of the array, and (unlike the perfectly legal situation at the *end* of an array, where we need only one extra byte to accommodate the off-array pointer) we cannot guarantee that your pointer will be identified with the object it “nearly” points to. Your array may be left with *no* valid pointer, and summarily reclaimed. Fortunately, this is easy to fix when you have source code: declare every unit-offset array one element bigger than you need it, and simply do not use the zeroth element.

¹Press, Flannery, Teukolsky, and Vetterling, *Numerical Recipes in C*, 1ed, Cambridge, 1988.

API Reference

GEODESIC RUNTIME TYPES

The following section contains types we define in our interface. In C/C++, a user-defined type is just shorthand for expressing a built in type, or some combination of built-in types. We define these types so we can use them as arguments to or returns from our API functions.

gsWord

```
typedef unsigned long gsWord;
```

An unsigned integral type used to represent a word.

gsSignedWord

```
typedef long gsSignedWord;
```

A signed integral type used to represent a word.

gsPutFunction

```
typedef void (* gsPutFunction)(const char *msg);
```

A function pointer type for puts-like function.

gsConstCharStar

```
typedef const char * gsConstCharStar;
```

A string type.


GEODESIC RUNTIME MACROS

A macro is a construct that is expanded by the C/C++ preprocessor.



You must include the Runtime header file `gst.h`, to use any of the Runtime macros.



A macro being listed as  denotes that we recommend the user have in-depth knowledge of their program and the command before using it.

GS_FAILURE

This macro is used to return the failure of setting a function. This equates to a value of -1.

GS_NEXT(ptr)



This macro is used with memory allocated by `gsMallocMany()`. If `ptr` points to one of the allocated objects, then `GS_NEXT(ptr)` is the address of the next one. If there are no more objects, it returns 0.


GS_SUCCESS

This macro is used to return the success of setting a function. This equates to a value of 0.

GEODESIC RUNTIME FUNCTIONS

This section lists the Runtime functions. Functions always override environment variables.



A function being listed as  denotes that we recommend the user have in-depth knowledge of their program and the command before using it.

FunctionName ()

Prototype: Parameters of the function call.

Default: Default value of a setter function if not called.

Description: Description of what the function does.

Function Return: Results of calling the function.

Restrictions: Restrictions for the use of the function.

Related APIs: Corresponding environment variable. The function will override any setting of the environment variable. May also list other related APIs.

Comments and Warnings: Listing of any comments or warnings.

gsCalloc ()

Prototype: `void* gsCalloc (size_t size, size_t number);`

Default: n/a

Description: `gsCalloc()` behaves like `calloc()`, returning an array of zeroed memory. Memory allocated with `gsCalloc()` is automatically managed and will be automatically freed by Runtime when no longer in use.

Function Return: None.

Restrictions: None.

Related APIs: `calloc()`,

Comments and Warnings: None.

gsCollect ()

Prototype: `void gsCollect (void);`

Default: n/a

Description: Tells Runtime to perform a complete collection cycle. While Runtime automatically schedules garbage collection cycles according to your program's needs, `gsCollect()` lets you force a collection cycle at any time. For example, you may

know that after displaying a calculation, your program will be doing nothing else for a while, making this an ideal time to do some garbage collecting. See “Controlling When Garbage Collection Occurs” on page 24 for usage information.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: None.

gsFootPrintReduce ()



Prototype: `void gsFootPrintReduce(void);`

Default: n/a

Description: Calling `gsFootPrintReduce()` will shrink the heap if possible. Ordinarily, `gsFootPrintReduce()` is called as a side effect of every collection. You need to call it explicitly only if your program has a predictable pattern of varying heap use, and you have a good reason to reduce its footprint during ebb times; for example: to release memory for another process.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: None.

gsFree ()

Prototype: `void gsFree (void * ptr);`

Default: n/a

Description: Frees memory allocated by any of Runtime’s memory allocation functions to `ptr`.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: None.

gsGetAddBytes ()

Prototype: `int gsGetAddBytes (void);`

Default: n/a

Description: Returns the number of bytes an allocation is increased by, rounded up to the nearest word.

Function Return: int

Restrictions: None.

Related APIs: Can be modified with `gsSetAddBytes()` or `GS_ADD_BYTES`.

Comments and Warnings: None.

`gsGetAllowUserStacks()`

Prototype: int `gsGetAllowUserStacks (void);`

Default: n/a

Description: Returns the status Runtime allowing user-level libraries.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: Can be set with `gsSetAllowUserStacks()`.

Comments and Warnings: None.

`gsGetAutomaticGarbageCollection()`



Prototype: int `gsGetAutomaticGarbageCollection (void);`

Default: n/a

Description: This function returns the status of Runtime's garbage collection schedule.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: This can be set with `gsSetAutomaticGarbageCollection()`.

Comments and Warnings: None.

`gsGetCollectAtEnd()`

Prototype: int `gsGetCollectAtEnd (void);`

Default: n/a

Description: Returns the status of Runtime doing an additional collection at program exit.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: Can be set with `gsSetCollectAtEnd()`.

Comments and Warnings: None.

`gsGetDesktopInteraction()`



Prototype: `int gsGetDesktopInteraction (void);`

Default: n/a

Description: This function returns the status of Runtime using dialog boxes to display errors.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: Microsoft Windows Only

Related APIs: Can be set with `gsSetDesktopInteraction()`.

Comments and Warnings: None.

`gsGetFastMode()`

Prototype: `int gsGetFastMode (void);`

Default: n/a

Description: This function returns a value of `GS_ON` if fast mode is currently enabled.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: This can be set with `gsSetFastMode()`.

Comments and Warnings: None.

`gsGetFixPrematureFrees()`

Prototype: `int gsGetFixPrematureFrees (void);`

Default: n/a

Description: This function returns `GS_ON` if premature frees are already being fixed.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: This can be set with `gsSetFixPrematureFrees()`.

Comments and Warnings: None.

gsGetFootPrintReduce ()

Prototype: `int gsGetFootPrintReduce (void);`

Default: n/a

Description: This function returns the current status of footprint reduction.

Function Return: GS_ON or GS_OFF

Restrictions: None.

Related APIs: This can be set with `gsSetFootPrintReduce()`.

Comments and Warnings: None.

gsGetFullLogFileName ()

Prototype: `const char* gsGetFullLogFileName (void);`

Default: n/a

Description: This function returns the fully qualified name of the log file, including any settings of the name, path, logging with PID, or Hostname.

Function Return: char

Restrictions: None.

Related APIs: This is modified by `gsLogFileName()`, `gsLogFilePath()`, `gsLogWithPID()`, and `gsLogWithHostname()`.

Comments and Warnings: None.

gsGetGCPriority ()



Prototype: `int gsGetGCPriority (void);`

Default: n/a

Description: This function returns the percentage of total live memory that will be allowed to accumulate garbage between collections.

Function Return: int

Restrictions: None.

Related APIs: This can be set with `gsSetGCPriority()`.

Comments and Warnings: None.

gsGetLogAllLeaks ()

Prototype: `int gsGetLogAllLeaks (void);`

Default: n/a

Description: Returns the status of Runtime reporting all leaks.

Function Return: GS_ON or GS_OFF

Restrictions: None.

Related APIs: Can be set with `gsSetLogAllLeaks()`.

Comments and Warnings: None.

gsGetLogFileName()

Prototype: `const char* gsGetLogFileName (void);`

Default: n/a

Description: Returns the base file name to which Runtime prints statistics, warnings, errors, and leak reports.

Function Return: char

Restrictions: None.

Related APIs: This can be set with `gsSetLogFileName()`.

Comments and Warnings:

gsGetLogFilePath()

Prototype: `char* gsGetLogFilePath (void);`

Default: n/a

Description: Returns the path, including drive, for the log file. Returns "." if set to the default of /tmp on Unix or \$TEMP or \$TMP on Windows.

Function Return: char

Restrictions: None.

Related APIs: Can be set with `gsSetLogFilePath()`.

Comments and Warnings: None.

gsGetLogWithHostname()

Prototype: `int gsGetLogWithHostname (void);`

Default: n/a

Description: This function returns the status of Runtime using the `Hostname` in the log file name.

Function Return: GS_ON or GS_OFF

Restrictions: UNIX only.

Related APIs: Can be set with `gsSetLogWithHostname()`.

Comments and Warnings: None.

gsGetLogWithPID()

Prototype: `int gsGetLogWithPID (void);`

Default: n/a

Description: Returns a value of GS_ON if a unique log file name containing its process ID is being used.

Function Return: GS_ON or GS_OFF

Restrictions: None.

Related APIs: Can be set with `gsSetLogWithPID()`.

Comments and Warnings: None.

gsGetMaxMemFreedForFootPrintReduce()



Prototype: `unsigned long gsGetMaxMemFreedForFootPrintReduce (void);`

Default: n/a

Description: This function returns the regularity in bytes of Runtime returning memory to the operating system.

Function Return: int

Restrictions: None.

Related APIs: Can be set with `gsSetMaxMemFreedForFootPrintReduce()`.

Comments and Warnings: None.

gsGetNumberOfHeaps()

Prototype: `int gsGetNumberOfHeaps (void);`

Default: n/a

Description: Returns the number of heaps being used by Runtime.

Function Return: int

Restrictions: SMP Product Only

Related APIs: This can be set with `gsSetNumberOfHeaps()` or GS_MULTI_HEAP.

Comments and Warnings: None.

gsGetPrintStats()

Prototype: `int gsGetPrintStats (void);`

Default: n/a

Description: This function returns the status of Runtime logging statistics on its operation.

Function Return: GS_ON or GS_OFF

Restrictions: None.

Related APIs: This can be set with `gsSetPrintStats()`.

Comments and Warnings: None.

gsGetPutsFunction()



Prototype: `gsPutFunction gsGetPutsFunction (void);`

Default: n/a

Description: This function returns a pointer to the function that Runtime uses to output strings.

Function Return: pointer

Restrictions: None.

Related APIs: This can be set with `gsSetPutsFunction()`.

Comments and Warnings: None.

gsGetReportLeakFunction()



Prototype: `void (*gsGetReportLeakFunction) (const char *block);`

Default: n/a

Description: This function returns a pointer to the function that Runtime uses to report leaks.

Function Return: pointer

Restrictions: None.

Related APIs: This can be set with `gsSetReportLeakFunction()`.

Comments and Warnings: None.

gsGetReportLeaksSwitch()

Prototype: `int gsGetReportLeaksSwitch (void);`

Default: n/a

Description: This function returns the status of Runtime creating a log file.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: This can be set with `gsSetReportLeaksSwitch()`.

Comments and Warnings: None.

gsGetScanAlignment()



Prototype: `int gsGetScanAlignment (void);`

Default: n/a

Description: This function returns the alignment of application pointers.

Function Return: `int`

Restrictions: None.

Related APIs: Can be set with `gsSetScanAlignment()`.

Comments and Warnings: None.

gsGetIgnoreOffPagePointers()

Prototype: `int gsGetIgnoreOffPagePointers (void);`

Default: n/a

Description: Returns the status of Runtime considering very large objects leaked if no pointer exists in the first page.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: Can be set with `gsSetIgnoreOffPagePointers()`.

Comments and Warnings: None.

gsGetStopSignal()



Prototype: `int gsGetStopSignal (void);`

Default: n/a

Description: This function returns the signal Runtime is using to put threads to sleep during a collection cycle.

Function Return: `int`

Restrictions: HP-UX, IBM AIX, and Linux (multi-threaded and SMP) Only

Related APIs: Can be set with `gsSetStopSignal()`.

Comments and Warnings: None.

gsGetVeryLargeAllocationSize()

Prototype: `unsigned long gsGetVeryLargeAllocationSize (void);`

Default: n/a

Description: This function returns the number of bytes an allocation has to be in order to be treated as a very large allocation.

Function Return: `int`

Restrictions: None.

Related APIs: This can be set with `gsSetVeryLargeAllocationSize()`.

Comments and Warnings: None.

gsGetZeroAllocatedObject()

Prototype: `int gsGetZeroAllocatedObject (void);`

Default: n/a

Description: This function returns a value of `GS_ON` if all non-leaf objects that Runtime allocates are filled with zero bytes.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: This can be set `gsSetZeroAllocatedObject()`.

Comments and Warnings: None.

gsInitialize()

Prototype:

```
gsInitFunction gsInitialize ( void ) {  
    /* . . . API calls . . . */  
}
```

Default: n/a

Description: Define this function to make Runtime settings at Runtime initialization. See “Configuring Geodesic Runtime” on page 34 for examples and more information.

Function Return: Does not return a value.

Restrictions: None.

Related APIs: For a list of functions that must be called in `gsInitialize()`, see “When to Configure Runtime” on page 34.

Comments and Warnings: The function `gsInitialize()` is executed before Runtime is enabled and before the C run-time libraries are initialized. Do not put any code that calls C run-time library or memory allocation functions in `gsInitialize()`.

gsMalloc()

Prototype: `void* gsMalloc (size_t size);`

Default: n/a

Description: This function allocates an automatically managed object of size `size`. If `gsSetZeroAllocatedObject()` is set to `GS_ON` or the environment variable `GS_ZERO_ALLOCATED_OBJECT` is set, the returned memory is zeroed. Like `malloc()`, `gsMalloc()` will return `NULL` if it is unable to allocate the space.

Function Return: None.

Restrictions: None.

Related APIs: `gsSetZeroAllocatedObject()`,
`GS_ZERO_ALLOCATED_OBJECT`

Comments and Warnings: None.

gsMallocIgnoreOffPage()

Prototype: `void* gsMallocIgnoreOffPage (size_t size);`

Default: n/a

Description: This function behaves like `gsMalloc()` with the following exception: the memory allocated will only be considered as in use if there is a pointer that is pointing somewhere within the first logical page of the memory block. Runtime uses an internal logical page size of 8192 bytes, which may be larger than the physical page size on your platform. If the function `gsSetIgnoreOffPagePointers()` has been

set to `GS_ON`, then `malloc()` will call `gsMallocIgnoreOffPage()` to allocate memory. See the heading “Ignoring Interior Pointers Beyond First Page” on page 27 for an explanation of when to use `gsMallocIgnoreOffPage()`.

Function Return: `GS_ON` or `GS_OFF`

Restrictions: None.

Related APIs: `gsMalloc()`

Comments and Warnings: None.

`gsMallocLeaf()`



Prototype: `void* gsMallocLeaf (size_t size);`

Default: n/a

Description: You can optionally use `gsMallocLeaf()` to allocate automatically managed memory that you know does not contain pointers. The memory returned is not zeroed. Using `gsMallocLeaf()` improves Runtime’s efficiency but can lead to bugs if `gsMallocLeaf()` is used to allocate memory that contains pointers. See the heading “Telling Runtime Which Memory to Scan for Pointers” on page 25 for more information on using `gsMallocLeaf()` to tune Runtime’s performance.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: Any object allocated with `gsMallocLeaf()` will not be scanned for pointers by Runtime. Consequently, if an automatically managed object were only accessible through this object, it might be reclaimed by Runtime while still in use. By only using `gsMallocLeaf()` to allocate objects without pointers, you avoid any risk of reclaiming an object that is still in use. However, there are exceptional situations when `gsMallocLeaf()` may be used to allocate objects containing pointers. These exceptions are very advanced and dangerous and should be considered for experts only:

If you know for a fact that Runtime does not need to scan a particular object to find everything it uses, you may be able to allocate that object with `gsMallocLeaf()`. For example, suppose you have a type `T` that contains a pointer to an element of an automatically managed array. If you know for certain that there also will always be a global pointer aimed at the array, then you may be able to allocate objects of type `T` with `gsMallocLeaf(T)`.

You can use `gsMallocLeaf()` to provide “weak pointers”. A *weak pointer* is a pointer that is not meant to be traced by the collector. For example, suppose you have a

hash table containing objects that are used by data structures in your program. Normally, Runtime will not reclaim the object when your program is no longer using it, because it is still being “used” by the hash table, which has a pointer to it. If you allocate some of the hash table’s internal data structures with `gsMallocLeaf()`, Runtime will reclaim the object when the rest of the program is no longer using it.

gsMallocManual()

Prototype: `void* gsMallocManual (size_t size);`

Default: n/a

Description: This function allocates manually managed memory. Memory allocated by `gsMallocManual()` should be explicitly released by calling `free()` or `gsFree()`. Memory allocated by `gsMallocManual()` will be scanned by Runtime for pointers to protect any automatically managed memory it uses. See “How to Mix Automatic and Manual Memory Management” on page 30 for usage information.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: None.

gsMallocMany()



Prototype: `void *gsMallocMany (size_t size);`

Default: n/a

Description: This advanced function is meant for use in multi-threaded programs. `gsMallocMany(s)` allocates approximately a page worth of objects of size `size`, which are returned in a linked list. Using `gsMallocMany()` improves the efficiency of multi-threaded programs by allowing a thread to allocate a number of objects under a single lock.

The return value of `gsMallocMany()` is a pointer to the first object of the list. The first word of each item is a pointer to the next item and the remainder of the item is zeroed. For convenience, the macro `GS_NEXT(ptr)` returns the next item after `ptr`. When you use the item, initializing it with your data overwrites the pointer, automatically de-linking the item from the list. At that point, Runtime will see the item as independent of the rest of the list and will automatically reclaim it when no longer in use.

The following example illustrates the use of `gsMallocMany()`. This shows a function, possibly the main function of a thread, that allocates 100 objects of type `struct A` and places their address in an array.

```
thr_main()
{
/* memPool holds objects returned by gsMallocMany() until*/
/* needed by program */
struct A *memPool; /* Pool of pre-allocated objects */
struct A *array[100]; /* Where objects need to end up */
int i;
for(i = 0, memPool = 0; i < 100; i++) {
    if(memPool == 0) /* Any objects available to */
                    /* put into array? */
        memPool = gsMallocMany(sizeof struct A);
/* If not, grab some. */
/* Put first object in list into the array */
array[i] = memPool;
memPool = GS_NEXT(memPool);
/* Remove it from list of available objects */
}
}
```

See the heading “Tuning Multi-Threading” on page 28 for more on `gsMallocMany()`.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: None.

`gsRealloc()`

Prototype: `void* gsRealloc (void *ptr, size_t size);`

Default: n/a

Description: This behaves like `realloc()`, resizing the data pointed to by `ptr` to have size `size`. Returns a pointer to the resized memory, which may or may not be the same as `ptr`. The new memory shares all the same properties as the old memory (e.g. whether it is automatically or manually managed, whether it is a leaf, or whether it has debug information).

If you call the function `gsSetFixPrematureFrees(GS_ON)` or set the environment variable `GS_DISABLE_FREE`, `delete` will call destructors but not release memory. If `gsRealloc()` returns a new pointer, the memory of the old object will not be freed until `REMDI` ensures it is safe to do so. This protects you from errors arising from freeing memory that is still in use and lets you reuse existing code easily but may increase space requirements. See the entry on

`gsSetFixPrematureFrees()` in “Geodesic Runtime Functions” starting on page 47 for more information.

Function Return: None.

Restrictions: None.

Related APIs: None.

Comments and Warnings: None.

gsSetAddBytes()

Prototype: `int gsSetAddBytes(int n)`

Default: 0 (zero)

Description: `gsSetAddBytes()` protects your application from code that writes outside allocated memory. When you set `gsSetAddBytes()`, the size of any allocation request will be increased by the number of bytes specified, rounded up to the nearest word. See the user’s guide for more information regarding robust mode.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Works only in robust mode (default mode); will not work if `GS_FAST_MODE` is set or `gsSetFastMode(GS_ON)` has been called. Must be called in `gsInitialize()`.

Related APIs: The current setting can be obtained via `gsGetAddBytes()`. The setting of this function will override any setting of the environment variable `GS_ADD_BYTES` in your program.

Comments and Warnings: When using robust mode, all allocation requests are already increased by two words (8 bytes on a 32-bit machine). Any settings of less than 8 bytes will therefore be ignored.

gsSetAllowUserStacks()

Prototype: `int gsSetAllowUserStacks (int);`

Default: `GS_ON`

Description: Sometimes programs implement their own user-level thread libraries that change the location of the stack. By default, Runtime is set to correctly handle this situation.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_ALLOW_USER_STACKS` in your program. The current setting can be obtained via `gsGetAllowUserStacks()`.

Comments and Warnings: The collection frequency may be too high if user stacks are used and this function is set to `GS_OFF`.

gsSetAutomaticGarbageCollection()



Prototype: `int gsSetAutomaticGarbageCollection (int);`

Default: `GS_ON`

Description: This function tells Runtime whether to automatically schedule garbage collection. If `gsSetAutomaticGarbageCollection()` is set to the default value of `GS_ON`, Runtime will automatically schedule garbage collection cycles whenever appropriate. When `gsSetAutomaticGarbageCollection()` is set to `GS_OFF`, the garbage collector will not operate unless your program runs out of memory or you explicitly request garbage collection by calling `gsCollect()`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: None.

Related APIs: The setting of this function will override any setting of the environment variable `GS_DISABLE_AUTOMATIC_GARBAGE_COLLECTION` in your program. The current setting can be obtained via `gsGetAutomaticGarbageCollection()`.

Comments and Warnings: None.

gsSetCollectAtEnd()

Prototype: `int gsSetCollectAtEnd (int);`

Default: `GS_OFF`

Description: By default, Runtime does not run an additional collection cycle at program exit. To turn this on, set `gsSetCollectAtEnd()` to `GS_ON`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: None.

Related APIs: The setting of this function will override any setting of the environment variable `GS_COLLECT_AT_END` in your program. The current setting can be obtained via `gsGetCollectAtEnd()`.

Comments and Warnings: By our collector's definition, objects that are still alive and bound to valid pointers when the program exits are *not* memory leaks. This may be counterintuitive to former 16-bit Windows programmers, since in that programming model a process was responsible for cleaning up its own address space before exit. In a modern architecture, each process has its own protected address space, and reclaiming that space when the process dies is the responsibility of the operating system. Some memory debuggers, including NuMega's BoundsChecker, will continue to report these

objects as leaks. Programmers using Runtime in conjunction with other debugging tools should be aware of the issue, to avoid confusion.

Because `gsSetCollectAtEnd()` uses the standard C library function `atExit()` to force the final collection cycle, the final collection cycle will take place after all the local variables of `main()` have been destroyed. Global variables will still be in existence.

`gsSetDesktopInteraction()`



Prototype: `int gsSetDesktopInteraction (int);`

Default: `GS_ON`

Description: This function tells Runtime whether to use dialog boxes to display errors. The default value of `GS_ON` directs Runtime to use dialog boxes to display certain internal error messages. When this variable is set to `GS_OFF`, the errors are written to the log file, provided the log file is enabled with `GS_REPORT_LEAKS` being set. This variable is used when development is done using an NT service or a console application that cannot interact with the desktop.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Microsoft Windows Only. Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_DO_NOT_INTERACT_WITH_DESKTOP` in your program. Note that value of `GS_ON` for `gsSetDesktopInteraction()` equates with 0 for `GS_DO_NOT_INTERACT_WITH_DESKTOP`. The current setting can be obtained via `gsGetDesktopInteraction()`.

Comments and Warnings: None.

`gsSetFastMode()`

Prototype: `int gsSetFastMode (int mode);`

Default: `GS_OFF`

Description: Setting this to `GS_ON` enables fast mode, optimizing the allocation strategy for speed. See the user's guide for information regarding fast mode.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_FAST_MODE` in your program. The current setting can be obtained via `gsGetFastMode()`.

Comments and Warnings: None.

gsSetFixPrematureFrees ()

Prototype: `int gsSetFixPrematureFrees (int);`

Default: `GS_OFF`

Description: By setting this to `GS_ON`, tells Runtime to fix premature frees by preventing calls to `free()` and `delete` from reclaiming memory. Also optimizes collector tuning for fixing premature frees. See “Automatically Fixing Premature Frees” on page 32 for more information. To stop fixing premature frees, set this variable to `GS_OFF`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: None.

Related APIs: The setting of this function will override any setting of the environment variable `GS_DISABLE_FREE` in your program. The current setting can be obtained via `gsGetFixPrematureFrees()`.

Comments and Warnings: Fixing premature frees may increase space usage, since memory may be released more slowly after Runtime determines it is safe. This causes the system to potentially expand the heap if needed. A small amount of memory may not be released at all if Runtime cannot ensure that it is safe as described in “Geodesic Runtime Restrictions” on page 43. For most programs this is not significant, but it may be an issue in very large programs or programs with very strict memory requirements.

If `realloc()` returns a new pointer, the memory of the old pointer will not be freed until Runtime ensures it is safe to do so.

gsSetFootPrintReduce ()

Prototype: `int gsSetFootPrintReduce (int);`

Default: `GS_ON`

Description: Setting this function to `GS_OFF` turns off both automatic and explicit heap footprint reduction. This causes Runtime to retain unused memory to fulfill future memory requests instead of returning memory to the operating system.

Function Return: None.

Restrictions: `GS_SUCCESS` or `GS_FAILURE`

Related APIs: The setting of this function will override any setting of the environment variable `GS_DISABLE_FOOTPRINT_REDUCTION` in your program. The current setting can be obtained via `gsGetFootPrintReduce()`.

Comments and Warnings: If you wish to effectively disable automatic footprint reduction, set `GS_MEM_FREED_BEFORE_NEXT_FOOTPRINT_REDUCE` to 1000000000 (approximately 1 Gb). Manual footprint reduction will be available via `gsFootPrintReduce()`.

gsSetGCPriority()



Prototype: `int gsSetGCPriority (int);`

Default: 40

Description: This function controls the frequency of garbage collections. It represents the amount of garbage that will be allowed to accumulate between collections.

`gsSetGCPriority()` is a percentage of the total live memory:

$\text{Garbage between collections} = (\text{gsSetGCPriority}/100) * \text{live memory}$

(Live memory consists of the used heap)

By default, `gsSetGCPriority()` is set to 40. For example, if the live memory is 1 MB, the maximum garbage between collections is 0.4 MB. If `gsSetGCPriority()` is a larger value (`gsSetGCPriority=100`), there are going to be fewer collections but the size of the heap will grow. This is because fewer collections lead to expanding the heap more. On the other hand, if `gsSetGCPriority()` is a smaller value (`gsSetGCPriority=10`), the size of the heap will be smaller but there will be more collections. This setting trades speed for space.

Setting it to 10,000 (a factor of 100 times the live memory) or more effectively stops Runtime from garbage collecting unless your program runs out of memory or you explicitly request garbage collection by calling `gsCollect()`. A more efficient method of guaranteeing no collections is to set `gsSetAutomaticGarbageCollection()` to `GS_OFF`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: None.

Related APIs: The setting of this function will override any setting of the environment variable `GS_PRIORITY` in your program. The current setting can be obtained via `gsGetGCPriority()`.

Comments and Warnings: None.

gsSetLogAllLeaks()

Prototype: `int gsSetLogAllLeaks (int);`

Default: `GS_OFF`

Description: If this is set to `GS_ON`, then all leaks reported by the program are printed to the log. This overrides the default setting of 500 reported leaks.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`. This has no effect unless `gsSetReportLeaksSwitch()` is set to `GS_ON`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_LOG_ALL_LEAKS` in your program. The current setting can be obtained via `gsGetLogAllLeaks()`.

Comments and Warnings: None.

gsSetLogFileName()

Prototype: `int gsSetLogFileName (const char*);`

Default: `gs`

Description: This function sets the name of the file to which Runtime prints statistics, warnings, errors, and leak reports.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`. This has no effect unless `gsSetReportLeaksSwitch()` or `gsSetPrintStats()` is set to `GS_ON` or the environment variable `GS_REPORT_LEAKS` or `GS_PRINT_STATS` is set.

Related APIs: The setting of this function will override any setting of the environment variable `GS_LOG_FILE_NAME` in your program. The current setting can be obtained via `gsGetLogFileName()`.

Comments and Warnings: `.log` is automatically appended.

gsSetLogFilePath()

Prototype: `int gsSetLogFilePath (char*);`

Default: `/tmp` on Unix; the value of `$TEMP` or `$TMP` on Windows

Description: This function sets a path for the log file. Use `gsSetLogFileName()` to set the name of the log file. Under Windows, the drive can be set using either drive letters or UNC naming conventions.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`. This has no effect unless `gsSetReportLeaksSwitch()` or `gsSetPrintStats()` is set to `GS_ON` or the environment variable `GS_REPORT_LEAKS` or `GS_PRINT_STATS` is set.

Related APIs: The setting of this function will override any setting of the environment variable `GS_LOG_FILE_PATH` in your program. The current setting can be obtained via `gsGetLogFilePath()`.

Comments and Warnings: None.

gsSetLogWithHostname()

Prototype: `int gsSetLogWithHostname (int);`

Default: GS_OFF

Description: If `gsSetLogWithHostname` is set to GS_ON, the host name is added to the log file as it appears in HOSTNAME environment variable. This will vary from system to system. On some systems, HOSTNAME is set to the shortened host name (example: “duke”). On other systems, HOSTNAME is set to the full host name (example: “duke.company.com”).

Function Return: GS_SUCCESS or GS_FAILURE

Restrictions: UNIX only. Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable GS_LOG_WITH_HOSTNAME in your program. The current setting can be obtained via `gsGetLogWithHostname()`.

Comments and Warnings: Runtime only writes to the log file if `gsSetPrintStats()` or `gsSetReportLeaks()` is set to GS_ON or if the environment variable GS_PRINT_STATS or GS_REPORT_LEAKS is set. Because `gsSetLogWithHostname()` is irrelevant if no log file is written, it is generally used in conjunction with these calls.

`gsSetLogWithPID()`

Prototype: `int gsSetLogWithPID (int);`

Default: GS_OFF

Description: This function determines the name of log file to which subsequent debug info will be written. The default GS_OFF setting directs Runtime to `gs.log` in the directory where the executable resides and will overwrite the previous `gs.log` produced by the same executable. If `gsSetLogWithPID()` is set to GS_ON, every instance of the program will write to a unique log file name containing its process ID and will not overwrite earlier log files written by the same executable.

Function Return: GS_SUCCESS or GS_FAILURE

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable GS_DONT_LOG_WITH_PID in your program. The current setting can be obtained via `gsGetLogWithPID()`.

Comments and Warnings: Runtime only writes to the log file if `gsSetPrintStats()` or `gsSetReportLeaks()` is set to GS_ON or if the environment variable GS_PRINT_STATS or GS_REPORT_LEAKS is set. Because `gsSetLogWithPID()` is irrelevant if no log file is written, it is generally used in conjunction with these calls.

gsSetMaxMemFreedForFootPrintReduce ()



Prototype: `int gsSetMaxMemFreedForFootPrintReduce (unsigned long);`

Default: 1024000

Description: Runtime periodically returns memory to the operating system. The default regularity is 1024000 bytes (1 MB) of freed memory. You can change the frequency of footprint reduction by altering this value in bytes. This can be useful if your application has no other applications running on the same system. If you wish to effectively prevent automatic footprint reduction, set this to 1000000000 (approximately 1 Gb); manual footprint reduction will still be available via `gsFootPrintReduce()`.

Function Return: GS_SUCCESS or GS_FAILURE

Restrictions: None.

Related APIs: The setting of this function will override any setting of the environment variable `GS_MAX_MEM_FREED_BEFORE_NEXT_FOOTPRINT_REDUCE` in your program. The current setting can be obtained via `gsGetMaxMemFreedForFootPrintReduce()`.

Comments and Warnings: None.

gsSetNumberOfHeaps ()

Prototype: `int gsSetNumberOfHeaps (int numHeaps);`

Default: 2x the number of processors

Description: This function specifies the number of heaps for Runtime SMP to use.

Function Return: GS_SUCCESS or GS_FAILURE

Restrictions: SMP Product Only. Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_MULTI_HEAP` in your program. The current setting can be obtained via `gsGetNumberOfHeaps()`.

Comments and Warnings: See the user's guide for more information on selecting the right number of heaps.

gsSetPrintStats ()

Prototype: `int gsSetPrintStats (int);`

Default: GS_OFF

Description: This function tells Runtime whether or not to log statistics on its operation. Setting `gsSetPrintStats()` to `GS_ON` causes Runtime to periodically print statistics on its behavior and memory usage. These statistics will be output to `gs.log`, unless you have modified `gsSetLogFileName()`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_PRINT_STATS` in your program. The current setting can be obtained via `gsGetPrintStats()`.

Comments and Warnings: None.

`gsSetPutsFunction()`



Prototype: `void gsSetPutsFunction (gsPutFunction);`

Default: n/a

Description: This function sets a pointer to the function that Runtime uses to output strings. The default writes to the log file (if `gsSetReportLeaksSwitch()` or `GS_REPORT_LEAKS` is set to `GS_ON`). All Runtime reporting uses the function pointed to by `gsSetPutsFunction()` to write, so this function implements Runtime logging (unless you have redefined Runtime's reporting behavior to bypass it). To provide your own logging behavior, set `gsSetPutsFunction()` to the address of your logging function during collector initialization as described in "Configuring Geodesic Runtime" on page 34.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: None.

Related APIs: The current function can be obtained via `gsGetPutsFunction()`.

Comments and Warnings: This function should never go through the `malloc()` interface itself to allocate memory dynamically! The C run-time libraries often allocate heap memory, so it is usually unsafe to call them from within your `gsSetPutsFunction()`.

gsSetReportLeakFunction()



Prototype: `int (*gsSetReportLeakFunction) (const char *block);`

Default: n/a

Description: When Runtime reclaims a leaked object, it calls its default function with the address of the object. To provide your own leak reporting behavior, set `gsSetReportLeakFunction()` to the address of your leak reporting function.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: None.

Related APIs: The current setting can be obtained via `gsGetReportLeakFunction()`.

Comments and Warnings:

gsSetReportLeaksSwitch()

Prototype: `int gsSetReportLeaksSwitch (int);`

Default: `GS_OFF`

Description: When this is set to `GS_ON`, Runtime will create a log file with brief reports on memory leaks and other memory management errors.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_REPORT_LEAKS` in your program. The current setting can be obtained via `gsGetReportLeaksSwitch()`.

Comments and Warnings: None.

gsSetScanAlignment()



Prototype: `int gsSetScanAlignment (int alignment);`

Default: Depends upon `sizeof(void*)` for your operating system

Description: This function tells Runtime how pointers are stored in terms of bytes. For example, setting scan alignment to 4 says that pointers are all stored on 4 byte boundaries, although they may point to any address (aligned or not). The default value depends on the compiler. If you pack your data structures, you may need to change this value accordingly.

Function Return: GS_SUCCESS or GS_FAILURE

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_SET_SCAN_ALIGNMENT` in your program. The current setting can be obtained via `gsGetScanAlignment()`.

Comments and Warnings: See the notes in “How to Fix Memory Bugs in Third-party DLLs or Shared Libraries” on page 83 and “How to Fix Memory Bugs in Third-party Static Libraries” on page 84 for suggested usage.

Lowering it may substantially hurt performance, while setting it too high will cause Runtime to not recognize genuine pointers and erroneously free data that is still in use.

gsSetIgnoreOffPagePointers()

Prototype: `int gsSetIgnoreOffPagePointers (int);`

Default: `GS_OFF`

Description: If `gsSetIgnoreOffPagePointers()` is set to `GS_ON`, Runtime will consider a very large object as leaked if there is no pointer into its first page. This can help identify very large leaked objects even if there is a stray pointer into them. `malloc()` and `new` will use `gsMallocIgnoreOffPage()` to allocate objects larger than 65,535 bytes. You can change the threshold for very large objects by redefining `gsSetVeryLargeAllocationSize()`.

Function Return: GS_SUCCESS or GS_FAILURE

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_IGNORE_OFF_PAGE_POINTERS` in your program. The current setting can be obtained via `gsGetIgnoreOffPagePointers()`.

Comments and Warnings: See “Ignoring Interior Pointers Beyond First Page” on page 27 for usage information.

gsSetStopSignal()



Prototype: `int gsSetStopSignal (int newSignal);`

Default: see description below

Description: Runtime uses signals to put threads to sleep during a garbage collection cycle. Runtime searches for an available, unused signal at runtime. If you feel you need to, you can manually set the signal with `gsSetStopSignal()`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: HP-UX, IBM AIX, and Linux (multi-threaded and SMP) Only. Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_STOP_SIGNAL` in your program. The current setting can be obtained via `gsGetStopSignal()`.

Comments and Warnings: Be very careful with changing this as it can lead to thread problems.

`gsSetVeryLargeAllocationSize()`

Prototype: `int gsSetVeryLargeAllocationSize (unsigned long);`

Default: 65535

Description: This function tells Runtime how many bytes an allocation has to be in order to be treated as a very large allocation. This value is used by `malloc()` and `new` when `gsSetIgnoreOffPagePointers()` is set to `GS_ON`.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The current setting can be obtained via `gsGetVeryLargeAllocationSize()`.

Comments and Warnings: See the heading “Ignoring Interior Pointers Beyond First Page” on page 27 for an explanation of when to use `gsSetIgnoreOffPagePointers(GS_ON)`.

`gsSetZeroAllocatedObject()`

Prototype: `int gsSetZeroAllocatedObject (int);`

Default: `GS_OFF`

Description: Setting this to `GS_ON` causes all non-leaf objects that Runtime allocates to be filled with zero bytes. This can help the effectiveness of garbage collection, at a slight performance penalty, by helping Runtime to not keep things erroneously alive.

Function Return: `GS_SUCCESS` or `GS_FAILURE`

Restrictions: Must be set in `gsInitialize()`.

Related APIs: The setting of this function will override any setting of the environment variable `GS_ZERO_ALLOCATED_OBJECT` in your program. The current setting can be obtained via `gsGetZeroAllocatedObject()`.

Comments and Warnings: None.

DEPRECATION INFORMATION

Throughout the development of Runtime, certain functions have been changed or removed. If you were using a previous version of Runtime, the following tables may help you relate the functions and variables you used before with the new functions and variables.

Functions

REMIDI² V5.x	Runtime V6.0
<code>gsGetHeapAllocBehavior()</code>	Removed
<code>gsGetIgnoreOffPage()</code>	<code>gsGetIgnoreOffPagePointers()</code>
<code>gsSetHeapAllocBehavior()</code>	Removed
<code>gsSetIgnoreOffPage()</code>	<code>gsSetIgnoreOffPagePointers()</code>

² REMIDI is a product of Geodesic Systems, Inc., which is not affiliated in any way with Remy Corporation.

Troubleshooting Guide

This section addresses problems or issues you may run into. If your problem is not described here, please contact Geodesic Systems, Inc. Technical Support, and we'll work with you personally.

TROUBLESHOOTING INJECTING WITH GEODESIC RUNTIME

"Is there a way I can diagnose problems with the injector?"



Microsoft Windows only

To help diagnose problems with the injector, set the environment variable `GS_INJECTOR_DIAG=1`. This will produce a file called `gsInjDiag.log` in the temporary directory (generally, this is `c:\temp`, but may be set otherwise), containing information such as the control file used and error codes.

"I injected with Runtime but my program did not create a log file and no error message has appeared."



Microsoft Windows only

There are two MSVC compiler options that will cause later injection to become impossible:

- The `/ML` or `/MLd` option causes MSVC to statically link in the single-threaded C-
Runtime.
- The `/MT` or `/MTd` option causes MSVC to statically link in the multi-threaded C-
Runtime.

If either of these has been used, Runtime cannot be injected.

"I've been getting poor performance since I injected."




Microsoft Windows only

When injecting under Windows, if no performance improvement is noticed, check if the application is statically linked against the C Runtime library. Runtime injection will not work if this is true.

TROUBLESHOOTING LINKING WITH GEODESIC RUNTIME

“The linker reports an error when compiling a Runtime program.”

Possible causes of linking problems:

- Make sure your program is linking with the appropriate library, object file and license file. Microsoft Windows users must link their program to `gs.obj` and Unix users need to use `gs.o`. For more information, see “Linking Runtime” in the appropriate section for your operating system.
- Observe proper linking order. The object file should come first, then the Runtime library, and then any third-party libraries you may be using.
- You need to use your compiler’s C++ linker, even if your program was written in C.
- All Runtime functions are defined by including `gst.h`. If you only use Runtime through normal C and C++ memory allocation functions, such as `malloc()` or `new`, you do not need to include `gst.h`. However, if you explicitly use any Runtime specific functions, you must include `gst.h`.
- If the linker reports that some Runtime symbols are multiply defined, you may have included two different Runtime libraries in your project or makefile. You can only use one Runtime library for each application.
-  Microsoft Windows only: If you are using the link line in the “Project Settings” dialog, put double quotes around the object file and library paths, since blank spaces (e.g. “Program Files”) confuse the linker.

“The linker complains about an unresolved symbol `_dlopen`.”



Linux, and Sun Solaris

When linking with the Runtime library, it is necessary to link `libdl.so` by adding the flag “`-ldl`” in the linker command line. See “Linking Runtime” in the appropriate section for your operating system.

“The linker reported the following error:”

```
/usr/ccs/bin/ld: Unsatisfied symbols:  
  sem_post (code)  
  sem_destroy (code)  
  sem_wait (code)  
  sem_init (code)
```



HP-UX only

This error occurs when trying to build a multi-threaded application with Runtime using the static library. You must add `-lrt` to the `lpthread` library option (`-lpthread -lrt`) in order to get it to correctly compile and link with a static library.

“When I start my linked application, I got the following error:
‘This unlicensed Geodesic Systems product may only be used
with the sample programs...’.”

This probably means that the library was unable to find a valid license. Make sure that the license file is available, is not corrupted, and was included in the link command.

If this still does not correct your problem, verify that the license has not expired, is for the correct product, application (if application specific), and platform.

TROUBLESHOOTING RUNNING WITH GEODESIC RUNTIME

“Objects that are no longer in use are not reclaimed.”

This is usually not a sign of incorrect operation. Of the following possible causes for this problem, only the last requires any corrective action on the part of the programmer:

- If your program only uses a small amount of memory (i.e. less than about 100 KB), it may not trigger a garbage collection cycle. This is normal behavior.

By not actively performing garbage collection on programs with small heaps, Runtime allows small programs to run even faster than if they were manually managed because no time is spent managing memory. Runtime will automatically start garbage collecting a program when the program has allocated more than approximately 100 KB of memory.

If you want Runtime to immediately reclaim unused memory, call `gsCollect()` to force a garbage collection cycle. Also, calling the function `gsSetGCPriority()` to decrease its value will make Runtime garbage collect more frequently. See “Geodesic Runtime Functions” starting on page 47 for details on these functions.

- Runtime may fail to reclaim a small number of objects because the compiler may have left a pointer to one of them in a register, an integer may coincidentally contain the object’s address, or other similar reasons. Runtime only reclaims an object if it is truly inaccessible from the program. The number of objects that are unnecessarily retained in memory is almost always minuscule, even for programs using many megabytes of memory. This retention does not result in the accumulation of leaked memory because the number of such unnecessarily retained objects is bounded over time. Furthermore, since the retained objects *are* accessible, Runtime’s conservative behavior is the only safe option.
- Unnecessary retention is more likely to occur if you have called `gsSetFixPrematureFrees(GS_ON)`. See “Automatically Fixing Premature Frees” on page 32 for more information.
- If the log file has the message “Needed to allocate object at xxxxx despite bogus pointer”, follow the procedure given under the heading “Ignoring Interior Pointers Beyond First Page” on page 27.

“Objects are reclaimed while still in use.”

This usually results from incorrect use of Runtime or a violation of its restrictions. Likely causes include incorrect scan alignment, hiding a pointer, or not telling Runtime about custom allocators for non-collectible data. “Geodesic Runtime Restrictions” on page 43 describes Runtime’s restrictions together with suggested solutions.

“Runtime is not returning as much memory to the OS as I would like.”

You can address this either by calling the function `gsSetGCPriority()` and decreasing the value or by calling the function `gsFootPrintReduce()`. Also verify that `gsSetFootPrintReduce()` is set to `GS_ON` and the environment variable `GS_DISABLE_FOOTPRINT_REDUCTION` is set to 0. Note that `gsFootPrintReduce()` does not occur during every collection cycle.



Footprint reduction is not supported on HP-UX.

“Footprint reduction doesn't seem to be happening.”

In an application that has heavy memory activity interspersed with periods of complete memory allocator idleness, Runtime will not footprint reduce properly because garbage collection is not active. Resolve this by calling `gsFootPrintReduce()` at the beginning of idle periods to manually trigger footprint reduction.

“I'm getting errors such as `ld.so.1: ld: fatal: relocation error: symbol not found:`”



Sun Solaris only.

By default, Solaris uses lazy linking. This means that the dependencies for functions from shared libraries are resolved the first time the function is called. Non-lazy linking forces the dependencies to be resolved when the library is first loaded. `LD_BIND_NOW` is a Solaris standard environment variable that can be set to force non-lazy binding. Alternatively, the “`-z now`” flag may be specified in the link line to require non-lazy binding.

Using `LD_BIND_NOW` does not actually solve this problem. The real problem is missing definitions for functions. With `LD_BIND_NOW`, errors of this type will be seen immediately upon execution of the program. Without this, the errors may not be seen until much later and may make it all the way to the distribute software.

“I'm having difficulty using a third-party memory diagnostic tool.”

Runtime may not function correctly when used simultaneously with some memory diagnostic tools. Configure the tool so that it doesn't report leaks or premature frees because these will automatically be fixed (at least for all automatically managed data) when you remove the test tool. You may want to try temporarily removing any Runtime libraries while testing your program with the memory diagnostic tool. See “Using Geodesic Runtime with Third-Party Tools and Libraries” on page 82 for more information.

Additional Debugging Technique

Here is another procedure that frequently helps in troubleshooting Runtime programs:

Turning off inlining and compiler optimizations often eliminates the effect of compiler bugs. If your program is crashing mysteriously, try recompiling your program with function inlining or optimization turned off to determine if the bug is really yours or the compiler's. In fact, this is a good debugging practice in all situations.

Appendix

USING GEODESIC RUNTIME WITH THIRD-PARTY TOOLS AND LIBRARIES

Runtime can be used in conjunction with most third-party tools or libraries and will fix bugs in them as well. This section covers special considerations associated with using Runtime with such tools, including memory management tools, GUI libraries, and container libraries.

Libraries and DLLs

One of Runtime's most powerful features is that it can often fix memory management bugs in third-party libraries even without access to the source code. All you need is access to their object modules or library files — usually files with the extension “.obj”, “.lib”, or “.dll” in Windows or “.o”, “.a”, or “.so” in Unix.

Signs that a Program or Library has Memory Bugs

Here are some signs that a program, or a library it is using, has memory management bugs that may be fixed by Runtime:

- **Incorrect data results.** You get incorrect information, but your data still balances. An example could be that the billing information is not correct.
- **The program crashes with a Protection Violation or a Segmentation Fault.** Program crashes are frequently signs of loose pointers resulting from premature freeing of memory.
- **The amount of memory available on your system keeps decreasing as the program executes.** This is frequently caused by memory leaks.
- **A leak detector or run-time debugging tool indicates a memory error.** You can check for suspected memory bugs by using Geodesic Great Circle or another memory debugging tool.
- **You aren't certain it does *not* have memory bugs.** Unless you are certain that a program or library doesn't have memory bugs, it is safest to assume that it does. Because memory errors are often intermittent or obscure, it is hard to be confident that memory is being managed correctly. As a precaution, you can use Runtime to automatically fix many types of hidden memory bugs.

How to Fix Memory Bugs in Third-party DLLs or Shared Libraries

DLLs or shared libraries that link dynamically to the run-time libraries

Runtime automatically detects and garbage collects shared libraries and DLLs that use the DLL or shared library version of the C run-time libraries. In particular, Windows users do not need to modify the Microsoft-supplied MFC DLLs to use them with Runtime.

The usual and recommended procedure is to link Runtime into the application that calls the DLL or shared library. This will cause us to collect (1) the application itself, by link-time interception, and (2) the C run-time libraries, and any shared libraries or DLLs that call them, by run-time interception. Linking Runtime into the DLL or shared library itself will also work in most cases, and may be the only viable strategy if your library will be called from third-party code that you do not control. However, run-time interception can cause problems if triggered from a library that is loaded explicitly by `LoadLibrary()` after the parent application has already allocated memory. In general, you should link Runtime directly into a library only if that library is going to be loaded implicitly at application start-up.

DLLs or shared libraries that link statically to the run-time libraries

If a DLL or shared library was statically linked with the C run-time libraries, Runtime will not garbage collect the memory allocated by that DLL or shared library. In this case, Runtime will neither help nor hurt the DLL or shared library's memory management. If you want Runtime to manage the memory allocated by that DLL or shared library, you must rebuild the DLL or shared library in one of the following ways:

- Rebuild with its build settings changed so that it uses the run-time libraries in a DLL or shared library. See the appendix for your operating system for instructions specific to your compiler.
- Rebuild the DLL or shared library with Runtime linked in. Add `gs.obj` or `gs.o` and the Runtime library to its project or makefile.



While the above procedures automatically fix most memory deallocation bugs, because the program or library you are attempting to fix is an unknown quantity, it may not be fixed by this procedure. Here are some possible reasons why:

The DLL or shared library may have bugs that are not memory bugs. This procedure only fixes bugs related to releasing memory at the wrong time. Bugs in overall program logic will not automatically be fixed by Runtime.

The DLL or shared library may use its own memory allocator. Try putting the Runtime library at different locations in the program's library list. The best location for the Runtime library in the list depends on whether the linker can successfully substitute Runtime's allocator and whether the third-party library requires special functionality from its allocator. Even if the shared library or DLL uses its own allocator, Runtime will not cause any harm, but it will not be able to fix memory errors in that library. "Geodesic Runtime Restrictions" on page 43 provides detailed information on Runtime's behavior with third-party allocators.

The DLL or shared library may use non-aligned data structures. Try calling `gsSetScanAlignment()` as described in "Geodesic Runtime Functions" starting on page 47 and "Geodesic Runtime Restrictions" on page 43.

The DLL or shared library may violate any of the Runtime restrictions given in "Geodesic Runtime Restrictions" on page 43. Except for scan alignment and the library using its own allocator as discussed above, violations of these restrictions are quite rare.

How to Fix Memory Bugs in Third-party Static Libraries

Try to fix your program by following Runtime's instructions for basic use. These instructions, summarized here, are described in greater detail in "Linking Runtime" in the appropriate section for your operating system.

- 1) Put the Runtime library and object file in your program's project or makefile, as well as the `gslicense.c` file. If possible, put the Runtime library at the beginning of your program's library list.
- 2) Rebuild your program.

This procedure will automatically fix memory bugs in many programs and libraries.



Because the program or library you are attempting to fix is an unknown quantity, it may not be fixed by this procedure. Here are some possible reasons why:

The third-party library may have bugs that are not memory bugs.

This procedure only fixes bugs related to releasing memory at the wrong time. Bugs in overall program logic will not automatically be fixed by Runtime.

The third-party library may use its own memory allocator. Try putting the Runtime library at different locations in the program's library list. The best location for the Runtime library in the library list depends on whether the linker can successfully substitute Runtime's allocator and whether the third-party library requires special functionality from its allocator. Even if the shared library or DLL uses its own allocator, Runtime will not cause any harm, but it will not be able to fix memory errors in that library. "Geodesic Runtime Restrictions" on page 43 provides detailed information on Runtime's behavior with third-party allocators.

The static library may use non-aligned data structures. Try calling `gsSetScanAlignment()` as described in "Geodesic Runtime Functions" starting on page 47 and "Geodesic Runtime Restrictions" on page 43.

The library may violate any of the Runtime restrictions. Except for the library using its own allocator as discussed above, violations of these restrictions are quite rare. See "Geodesic Runtime Restrictions" on page 43 for further information.

Memory Management Tools

Third-party memory managers or memory debugging tools that use their own allocators cannot be used to allocate automatically managed memory while Runtime is linked or injected with your program. "Geodesic Runtime Restrictions" on page 43 provides more detailed information on Runtime's behavior with third-party allocators.

Some memory debuggers require you to use their allocators for all memory allocation. Runtime's robustness features will not be available in the debugger. It is also possible that the debugger will report errors that Runtime would automatically handle.

Third-party memory allocators are sometimes used to improve memory management performance. Although you cannot use such custom allocators with Runtime, you won't sacrifice high performance. Runtime uses an extremely high-performance allocator that combines fixed-size allocators for small objects with a sophisticated understanding of cache locality.

GUI Class Libraries



This section applies only to C++ programs that use Runtime to manage all their memory. If you are using a C GUI interface (rather than a C++ Class Library) no special considerations apply. For more information, see "Mixing Automatic and Manual Memory Management" on page 29.

Overview

GUI libraries are class libraries that aid in the development of GUI applications. Common GUI libraries include Microsoft's *MFC*, Borland's *OWL*, and Rogue Wave's *View.h++* and *zApp*. These libraries are sometimes also referred to as *application frameworks* because they often provide class encapsulations of many programming constructs used to support modern windowing applications. The concepts encapsulated by these libraries can include graphical interfaces, persistence, error reporting, and data exchange formats such as OLE. They may even wrap the entire Win32 API under Microsoft Windows.

How to Use GUI Class Libraries with Runtime

Explicitly delete C++ GUI objects when they are no longer in use. Explicitly delete objects that inherit from GUI classes when they are no longer in use as you would if you were not using automatic memory management. In particular, you should `delete` an object when you want its destructor to be called immediately, perhaps to close a window.

This is necessary to ensure that destructors of GUI objects are called correctly. This special treatment of destructors is necessary because GUI libraries use destructors both to return windowing resources and to force immediate actions. For example, classes that represent fonts or brushes release these resources in their destructors, or a class representing a dialog box may use its destructor to remove itself from the display.

Runtime does not call destructors when it automatically reclaims objects. Explicitly deleting GUI classes ensures that destructors are called properly.

Container Libraries

The container library supports a number of different ways to store data: sets, multisets, lists, hash tables, stacks, etc.

How to Use Container Libraries with Runtime

Container libraries work very well with Runtime. In fact, automatic memory management is often necessary to effectively use container libraries. This section explains how Runtime automatically makes container libraries easier to use.

You do not need to do anything special to use container libraries with Runtime. Simply use your container library without worrying about when memory is released.

The Memory Management Challenge Imposed by Containers

Container libraries traditionally present many memory management challenges, all of which are automatically addressed by Runtime. These challenges arise when a program component destroys a container or removes elements from a container. Because the elements of a container may have been inserted in the container at any time by any part of the program, the program component cannot easily determine whether or not it is the last user of these elements, leaving it unsure whether to delete them. Often the component needs to know about the rest of the application to answer these questions, making the component nearly

impossible to reuse in other applications. Often programs are simply forced to leak memory as the only way of being sure an object is not in use.

With Runtime, these difficulties automatically disappear. Runtime will automatically release memory at an appropriate time, so you can simply use containers without worrying about managing their memory.

A Comparison of Automatic and Traditional Management of Containers

Most container libraries include strategies for memory management. Although these strategies are the best that can be done without garbage collection, we will show why nothing short of Runtime's automatic memory management technology comes close to solving the problems of managing memory in the presence of container libraries.

Containers with Multiple Removal Functions

Some container libraries require you to choose among separate functions to remove an element from a container depending on whether or not you want the element's memory freed. These libraries often also have separate functions for destroying the container with and without freeing the elements. Rogue Wave's *tools.h++* provides an example of this approach. This strategy simply transfers the memory management challenge entirely to the programmer, who has to figure out which function to call.

These libraries automatically work with Runtime. Simply remove elements from containers without worrying about when they will be freed. Their memory will be automatically freed when appropriate. Because Runtime can disable `free()` on garbage-collected memory entirely by calling `gsSetFixPrematureFrees(GS_ON)`, you can easily protect yourself from premature freeing as well as memory leaks.

Containers with Switches

Some container libraries have a switch on containers telling them whether to free their elements' memory when they are destroyed. Borland's class libraries are typical of this approach. Unfortunately, if the switch is set, the program cannot have more than one pointer aimed at a container element without risking loose pointers. If the switch is not set, the programmer has to implement all the correct and complex freeing logic without help from the container library.

These libraries automatically work with Runtime. If it is convenient, you may prefer to set the container's switch to not free its elements' memory, eliminating finalization concerns. Their memory will be automatically freed when appropriate. Because Runtime can disable `free()` on garbage-collected memory entirely by calling `gsSetFixPrematureFrees(GS_ON)`, you can easily protect yourself from premature freeing as well as memory leaks.

Value Based Containers

Some container libraries use *value semantics*. This means that they store copies of the elements put in them, rather than storing pointers to the elements. A good example of this approach is the Standard Template Libraries, which recently became part of the new C++ standard.

Value semantics clarify object ownership because each container has its own copy of the data, but making the copies exacts a big cost in space and time. More importantly, in order to get polymorphic containers and sharing of data, programmers frequently have to use containers whose elements are pointers and must manage the referenced memory entirely by hand.

These libraries automatically work with Runtime. Simply use these libraries without explicitly releasing memory.

If you are choosing a container strategy, you may want to avoid a strategy that involves passing large objects by value because passing data by value wastes time and space and Runtime eliminates the memory difficulties that often cause programmers to resort to value based containers.

Reference Counted Containers

Some container libraries use reference counting. This means that every element contains a count of how many pointers are pointing to it. In this case, the programmer usually uses smart pointers (i.e. classes that act like pointers) to reference the objects. When the reference count goes down to 0, the element is destroyed. Reference counting comes the closest of these approaches to automatically managing memory and is often described as a “poor man’s garbage collector”. Unfortunately, reference counting will not reclaim circular data structures, such as circular or doubly-linked linked lists. Reference counting implementations almost never allow polymorphism, multiple inheritance, interior pointers, non-heap objects, arrays, and composition. With Runtime, all these cases automatically work correctly, even without wrapping your pointers.

While reference counted container libraries automatically work with Runtime, you can gain extra efficiency by removing the now unnecessary reference counting logic and data from the library. This often automatically gives the container library new capabilities, such as polymorphism and the ability to store array elements.

Notices

PROPRIETARY RIGHTS

Portions of the Geodesic software include modifications to code which were released publicly by Xerox Corporation, subject to the requirement that the following notice be retained and included in the modified code: Copyright 1988, 1989 Hans-J. Boehm, Alan J. Demers. Copyright (C) 1991-1995 by Xerox Corporation. All rights reserved. THIS MATERIAL IS PROVIDED, AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESS OR IMPLIED. ANY USE IS AT YOUR OWN RISK.

Copyright (c) 2001, Geodesic Systems, Inc. All rights reserved.

Index

A

allocators	
custom.....	85
automatic and manual memory management	
mixing	29

C

C run-time DLLs	
intercepting.....	33
calloc().....	40
configuration.....	34
container libraries.....	86
custom allocators	85

D

delete.....	40
deprecation.....	74
DLL	
Microsoft Windows	18

E

export control classification number	6
environment variables	
configuring Runtime.....	35

F

footprint reduction	
API.....	48, 64
free().....	40
functions.....	47

G

garbage collection	
frequency.....	65
scheduling.....	24
gs.log.....	66
gs.o.....	37
gs.obj	37
GS_FAILURE.....	46
GS_NEXT	46, 59
GS_SUCCESS	46
gsCalloc()	47
gsCollect()	47
gsFootPrintReduce()	48
gsFree()	48
gsGetAddBytes()	48
gsGetAllowUserStacks().....	49
gsGetAutomaticGarbageCollection()	49
gsGetCollectAtEnd()	49
gsGetDesktopInteraction()	50
gsGetFastMode()	50
gsGetFixPrematureFrees()	50
gsGetFootPrintReduce()	51
gsGetFullLogFileName()	51
gsGetGCPriority()	51
gsGetIgnoreOffPagePointers()	55
gsGetLogAllLeaks()	51
gsGetLogFileName()	52
gsGetLogFilePath()	52
gsGetLogWithHostname().....	52
gsGetLogWithPID()	53
gsGetMaxMemFreedForFootPrintReduce ()	53
gsGetNumberOfHeaps()	53
gsGetPrintStats()	54
gsGetPutsFunction()	54
gsGetReportLeakFunction()	54, 70

gsGetReportLeaksSwitch()	55
gsGetScanAlignment()	55
gsGetStopSignal()	56
gsGetVeryLargeAllocationSize()	56
gsGetZeroAllocatedObject()	56
gsInitialize	36
gsInitialize()	57
gsMalloc	57
gsMalloc()	57
gsMallocIgnoreOffPage()	57
gsMallocLeaf()	58
gsMallocManual()	59
gsMallocMany()	59
gsRealloc()	60
gsSetAddBytes()	61
gsSetAllowUserStacks()	61
gsSetAutomaticGarbageCollection()	62
gsSetCollectAtEnd()	62
gsSetDesktopInteraction()	63
gsSetFastMode()	63
gsSetFixPrematureFrees()	64
gsSetFootPrintReduce()	64
gsSetGCPriority()	65
gsSetIgnoreOffPagePointers()	71
gsSetLogAllLeaks()	65
gsSetLogFileName()	66
gsSetLogFilePath()	66
gsSetLogWithHostname()	66
gsSetLogWithPID()	67
gsSetMaxMemFreedForFootPrintReduce())	68
gsSetNumberOfHeaps()	68
gsSetPrintStats()	68
gsSetPutsFunction()	69
gsSetReportLeaksSwitch()	70
gsSetScanAlignment()	70
gsSetStopSignal()	71
gsSetVeryLargeAllocationSize()	72
gsSetZeroAllocatedObject()	72
gst.h	39
GUI libraries	86

H

header file	39
-------------	----

I

icons	3
include files	39
initialization	<i>See</i> gsInitialize()
injection	
comparison to linking	7

L

libha	38
libraries	38
HP-UX	11
IBM AIX	13
libha	38
Linux	16
Microsoft Windows	18
Sun Solaris	20
third-party	82
license	
c source	9
comparison	9
text	9
linking	8
comparison to injection	7
HP-UX	11
IBM AIX	13
Linux	16
Microsoft Windows	18
Sun Solaris	20
troubleshooting	77
log file	
changing name	66
setting path	66

M

macros	46
mallinfo()	41
malloc()	40
manual and automatic memory	
management	
mixing	29
memory tools	
third-party	82
MFC libraries	83
Microsoft Windows API	33

multiple objects
 allocating..... *See* `gsMallocMany()`
 multi-threaded
 tuning..... 28

N

new 41
 notices..... 89

O

object file..... 37
 obsolete 74
 outside libraries..... 82

P

pointers
 ignoring beyond first page 27
 which memory to scan 25
 preprocessor macros..... 46
 priority settings..... 65
 proprietary rights..... 89

R

`realloc()` 42, 60

reference counting..... 88
 restrictions 43
 running
 troubleshooting..... 79

S

startup *See* `gsInitialize()`

T

third-party libraries..... 82
 third-party memory tools 82
 troubleshooting
 linking..... 77
 running..... 79
 typeface 3
 types..... 45

W

weak pointer..... 58

Z

zeroing non-leaf objects 72