Contents

1 Installation 1

1.1 UNIX ......................................................... 1

1.1.1 Installing the Files ........................................ 1

1.1.2 Configuring the Environment .............................. 2

1.1.3 Licensing .................................................. 2

1.1.4 Testing the Installation ................................. 3

1.1.5 Swap Space ............................................... 3

1.1.6 GAUSS Run-Time Engine ............................... 3

1.2 Windows 95/98/NT ......................................... 3

1.2.1 Installing the Files ...................................... 3

1.2.2 Configuring the Environment ............................ 4

1.2.3 Licensing ................................................ 4

1.2.4 POSIX Threads .......................................... 4

1.2.5 Testing the Installation ................................ 4

1.2.6 Swap Space .............................................. 4

1.2.7 GAUSS Run-Time Engine .............................. 5
2 Sample Programs

2.1 UNIX ......................................................... 7
2.2 Windows 98/ME/NT/2000 ................................. 8

3 Using the GAUSS Engine .............................. 9

3.1 Setup and Initialization ............................. 10
  3.1.1 Logging .............................................. 10
  3.1.2 Home Directory .................................. 10
  3.1.3 I/O Callback Functions ......................... 10
  3.1.4 Initialize Engine ................................. 11

3.2 Computation ............................................ 11
  3.2.1 Workspaces ....................................... 11
  3.2.2 Programs .......................................... 11
  3.2.3 GAUSS Engine Data Structures ................. 12
  3.2.4 Copying and Moving Data to a Workspace .... 14
  3.2.5 Getting Data From a Workspace ............... 15
  3.2.6 Calling Procedures ............................. 16

3.3 Shutdown ................................................. 16

4 Multi-threaded Applications .................. 17

4.1 Locks ................................................. 17
4.2 Compiling and Executing GAUSS Programs .... 18
  4.2.1 Assuring Concurrency ......................... 18
4.3 Calling GAUSS Procedures ....................... 19
  4.3.1 Assuring Concurrency ......................... 19
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5.1</td>
<td>Viewing Graphics</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>Interactive Commands</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>5.2.1</td>
<td>quit</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>5.2.2</td>
<td>ed</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>5.2.3</td>
<td>compile</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>5.2.4</td>
<td>run</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>5.2.5</td>
<td>browse</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>5.2.6</td>
<td>config</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>Debugging</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>The GC Compiler</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>7.1</td>
<td>Functions</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>7.1.1</td>
<td>Pre-initialization setup</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>7.1.2</td>
<td>Initialization and Shutdown</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>7.1.3</td>
<td>Compiling and Executing GAUSS programs</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>7.1.4</td>
<td>Calling Procedures</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>7.1.5</td>
<td>Creating and Freeing GAUSS Format Data</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>7.1.6</td>
<td>Moving Data Between GAUSS and Your Application</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>7.1.7</td>
<td>GAUSS Enterprise Engine Error Handling</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>7.2</td>
<td>Include Files</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>C API: Reference</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Structure Reference</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Index</td>
<td>177</td>
</tr>
</tbody>
</table>
Chapter 1

1.1 UNIX

1.1.1 Installing the Files

From CD or download, copy the `.tar.gz` file to `/tmp`.

Unzip the file using `gunzip`.

Create a directory to install the GAUSS Engine to. We’ll assume `/usr/local/mteng40`.

```
mkdir /usr/local/mteng40
```

Go to that directory.

```
cd /usr/local/mteng40
```

Extract the files from the tar file.

```
tar xvf /tmp/tar_file_name
```

The GAUSS Engine files are now in place.
1.1.2 Configuring the Environment

You need to set an environment variable called MTENGHOME that points to the installation directory.

C shell

```bash
setenv MTENGHOME /usr/local/mteng40
```

Korn, Bourne shell

```bash
MTENGHOME=/usr/local/mteng40
export MTENGHOME
```

The engine looks in $MTENGHOME for its configuration file, `gauss.cfg`. Anyone who will be running the engine needs to have at least read access to this file. The name of the environment variable can be changed to something other than MTENGHOME by calling `GAUSS_SetHomeVar`.

By default the engine creates temporary files in /tmp. You can change this by editing `gauss.cfg`—look for the `tmp_path` configuration variable. If you change it, anyone who uses the engine will need read/write/execute access to the directory you specify.

1.1.3 Licensing

AIX

The engine runs under a license server. Follow the instructions in `README.ls` in the `LServ` subdirectory for obtaining a license and running the license server.

You can verify the operation of the license server with the `lsmon` utility, also located in the `LServ` subdirectory.

```bash
LServ/lsmon
```

`lsmon` will list out the licenses the server is currently administering.

Solaris, OSF1, HP-UX, Linux

Execute `lmhostid` in the `FLEXlm` directory to get the hostid of the machine that will run the GAUSS Engine, or in the case of floating licenses, the machine that will run the license server daemon. Email the output to `license@Aptech.com`. You will be sent a license and instructions for its installation.
1. INSTALLATION

1.1.4 Testing the Installation

After completing the above steps, you can build some of the sample programs to verify the correctness of the installation. See section 2.1 for details.

1.1.5 Swap Space

The GAUSS Engine uses malloc and the normal system swap space. This system is dynamic and requires no workspace size setting. Make sure your system has enough swap space to handle the size and number of matrices you will be needing simultaneously. Each matrix takes $8 \times \text{rows} \times \text{columns}$ bytes.

1.1.6 GAUSS Run-Time Engine

If you have purchased the GAUSS Run-Time Engine (GRTE), you will see the shared library libmtengrt. To use it, use -lmtengrt instead of -lmteng in your Makefile. The GRTE will not create globals. It is to be used with compiled .gcg files that have been compiled with the GAUSS Engine.

To create compiled files, use the compile command from the command line interface, engauss or the gc executable. Your application can call GAUSS_LoadCompiledFile to load the program contained in the .gcg file.

Any global variables that are assigned within a GAUSS program or using the API assignment functions must be initialized in the .gcg file. GAUSS_CompileString can be used with the GRTE as long as it does not create new globals.

1.2 Windows 95/98/NT

1.2.1 Installing the Files

From CD

Insert the CD into a CD drive. We’ll assume e:. Go to the taskbar, click on Start, then Run..., then run e:\setup. setup will prompt you for registration information and a directory to install to, and copy the GAUSS Engine files to your hard disk.

From Download

Save the .zip file on your hard drive and unzip it into a temporary directory. We’ll assume c:\tmp. Go to the taskbar, click on Start, then Run..., then run c:\tmp\setup. setup will prompt you for registration information and a directory to install to, and copy the GAUSS Engine files to your hard disk.
1.2.2 Configuring the Environment

The engine requires an environment variable called MTENGHOME that points to the installation directory. setup initializes MTENGHOME for you, but you need to reboot (Windows 95/98) or log out and in again (Windows NT) for the change to take effect. If you didn’t do so when setup originally queried you, please do so now.

1.2.3 Licensing

Execute lmhost.exe in the flexlm directory to get the hostid of the machine that will run the GAUSS Engine, or in the case of floating licenses, the machine that will run the license server. Email the output to license@Aptech.com. You will be sent a license and instructions for its installation.

1.2.4 POSIX Threads

The GAUSS Engine is implemented using POSIX threads for Win32. you can obtain the Pthreads library from:

http://sources.redhat.com/pthreads-win32/

The GAUSS Engine was linked using pthreadVC.dll and pthreadVC.lib. You need both the .dll and the .lib file to link with the GAUSS Engine.

You will also need:

- pthread.h
- semaphore.h
- sched.h

1.2.5 Testing the Installation

After completing the above steps, you can build some of the sample programs to verify the installation. See section 2.2 for details.

1.2.6 Swap Space

The GAUSS Engine now uses malloc and the normal system swap space. This system is dynamic and requires no workspace size setting. Make sure your system has enough swap space to handle the size and number of matrices you will be needing simultaneously. Each matrix takes $8 \times \text{rows} \times \text{columns}$ bytes.
1. INSTALLATION

1.2.7 GAUSS Run-Time Engine

If you have purchased the **GAUSS Run-Time Engine**, you will find mtengrt.dll and mtengrt.lib. To use it, link with these instead of mteng.dll and mteng.lib in your Makefile. The **GRTE** will not create globals. It is to be used with compiled .gcg files that have been compiled with the **GAUSS Engine**.

To create compiled files, use the `compile` command from the command line interface, `engauss` or the `gc` executable. Your application can call `GAUSS_LoadCompiledFile` to load the program contained in the .gcg file.

Any global variables that are assigned within a **GAUSS** program or using the API assignment functions must be initialized in the .gcg file. `GAUSS_CompileString` can be used with the **GRTE** as long as it does not create new globals.
1. INSTALLATION
At least six sample programs are provided, *eng2d.c*, *mtexpr.c*, *mtcall.c*, *grte01.c*, *grte02.c* and *grte03.c*.

The examples that start with *grte* will run with the **GAUSS Run-Time Engine**. The makefile is set to link these examples with the **GAUSS Run-Time Engine**. You will need to modify the makefile to link them with the **GAUSS Engine**. See the source code for these examples for further instructions.

### 2.1 UNIX

The engine is shipped with several sample C programs that incorporate the engine, and a Makefile for building them. First, go to the directory you installed the engine to.

```
cd /usr/local/mteng40
```

**eng2d**

Run **make** to build **eng2d**.

```
make eng2d
```

**eng2d** sets some global variables, runs a program that uses them, then extracts the result from the workspace. Try running it.

```
./eng2d
```

You can see that the computation printed out by the **GAUSS** program and the data extracted by **GAUSS_GetMatrix** are the same.
2.2 **Windows 98/ME/NT/2000**

The engine is shipped with several sample C programs that incorporate the engine, and a Makefile for building them. (Note: The Makefile is written for Microsoft Visual C/C++ 5.0. If you are using a different compiler, you will have to manually compile the sample programs).

Open a Command Prompt (DOS) window and go to the directory you installed the engine to. We’ll assume \texttt{c:\mteng40}.

```
c:
   cd \mteng40
```

\texttt{eng2d}

Run \texttt{nmake} to build \texttt{eng2d}.

```
nmake eng2d
```

\texttt{eng2d} sets some global variables, runs a program that uses them, then extracts the result from the workspace. Try running it.

```
eng2d
```

You can see that the computation printed out by the \texttt{GAUSS} program and the data extracted by \texttt{GAUSS\_GetMatrix} are the same.

See the Makefile for other targets; there may have been additions after the manual was printed.
Chapter 3

Using the GAUSS Engine

This chapter covers the general guidelines for creating an application that uses the GAUSS Engine. Specific multi-threading issues are covered in Chapter 4.

The use of the GAUSS Engine can be broken up into the following steps:

- Setup and Initialization
  - Set up logging
  - Set home directory
  - Hook I/O callback functions
  - Initialize Engine

- Computation
  - Create workspaces
  - Copy or move data
  - Compile or load GAUSS code
  - Execute GAUSS code
  - Free workspaces

- Shutdown
3. Setup and Initialization

3.1 Logging

General GAUSS Engine system errors are sent to a file and/or a stream pointer. Default values are provided for each. You can change the default values or turn off logging altogether with GAUSS_SetLogFile and GAUSS_SetLogStream. This should be done before calling any other GAUSS Engine functions.

3.1.2 Home Directory

The GAUSS Engine home directory location is usually set to the same directory as the main executable of the calling application. It is used to locate the configuration file, Run-Time Library files, etc. used by the GAUSS Engine.

Use GAUSS_SetHome to set the home directory, prior to calling GAUSS_Initialize. An alternate method is to use GAUSS_SetHomeVar to set the name of an environment variable that contains the home directory location.

3.1.3 I/O Callback Functions

The GAUSS Engine calls user defined functions for program output from print statements and for error messages. Default functions are provided for the main thread in console applications.

Normal program output: stdout
Program error output: stderr
Program input: stdin

To change the default behavior, you can supply callback functions of your own and use the following functions to hook them:

Normal program output: GAUSS_HookProgramOutput
Program error output: GAUSS_HookProgramErrorOutput
Program input: GAUSS_HookProgramInputString

The functions GAUSS_HookProgramInputChar, GAUSS_HookProgramInputCharBlocking and GAUSS_HookProgramInputCheck are also supported, but no default behaviour is defined.

All I/O callback functions are thread specific and must be explicitly hooked in each thread that uses them, except for the three above that are hooked by default for the main thread.
3. **USING THE GAUSS ENGINE**

Use the hook functions to specify the input functions that the **GAUSS Engine** calls as follows:

<table>
<thead>
<tr>
<th>Functions Hooked By</th>
<th>Are Called By</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS__HookProgramInputChar</td>
<td>key</td>
</tr>
<tr>
<td>GAUSS__HookProgramInputCharBlocking</td>
<td>keyw, show</td>
</tr>
<tr>
<td>GAUSS__HookProgramInputCheck</td>
<td>keyav</td>
</tr>
<tr>
<td>GAUSS__HookProgramInputString</td>
<td>con, cons</td>
</tr>
</tbody>
</table>

There are two hook functions that are used to control output from **GAUSS** programs. Use **GAUSS__HookProgramOutput** to hook a function that **GAUSS** will call to display all normal program output. Use **GAUSS__HookProgramErrorOutput** to hook a function that **GAUSS** will call to display all program error output.

### 3.1.4 Initialize Engine

Call **GAUSS__Initialize** after the previous steps are completed. The **GAUSS Engine** is now ready for use.

### 3.2 Computation

#### 3.2.1 Workspaces

All computation in the **GAUSS Engine** is done in a *workspace*. Workspaces are independent from one another and each workspace contains its own global data and procedures. Workspaces are created with **GAUSS__CreateWorkspace**, which returns a *workspace handle*.

Workspaces are freed with **GAUSS__FreeWorkspace**. The contents of a workspace can be saved to disk with **GAUSS__SaveWorkspace**.

#### 3.2.2 Programs

Two functions are provided in order to execute **GAUSS** program code. Each requires a *program handle*.

- **GAUSS__Execute**  
  *Executes a **GAUSS** program*

- **GAUSS__ExecuteExpression**  
  *Executes a right-hand side expression*

Six functions are provided to create program handles. A program handle contains compiled **GAUSS** program code.
3. USING THE GAUSS ENGINE

Gauss_CompileExpression : Compiles a right-hand side expression
Gauss_CompileFile : Compiles a GAUSS program file
Gauss_CompileString : Compiles GAUSS commands in a character string
Gauss_CompileStringAsFile : Compiles GAUSS commands in a character string
Gauss_LoadCompiledFile : Loads a compiled program from disk
Gauss_LoadCompiledBuffer : Loads a compiled program from memory

The following code illustrates a simple program that creates a random matrix and computes its inverse.

```gauss
WorkspaceHandle_t *w1;
ProgramHandle_t *ph;
int rv;

w1 = Gauss_CreateWorkspace("Workspace 1");
ph = Gauss_CompileString(w1, "x = rndu( 10, 10 ); xi = inv( x );", 0, 0);
rv = Gauss_Execute(ph);
```

When this program is finished executing, the workspace will contain two global matrices. \( x \) is a \( 10 \times 10 \) matrix of random numbers and \( xi \) is its inverse.

The following code retrieves \( xi \) from the workspace to the calling application.

```gauss
Matrix_t *mat;
mat = Gauss_GetMatrix(w1, "xi");
```

The following code copies the retrieved matrix to another workspace as \( xinv \).

```gauss
WorkspaceHandle_t *w2;

w2 = Gauss_CreateWorkspace("Workspace 2");
rv = Gauss_CopyMatrixToGlobal(w2, mat, "xinv");
```

The copy can also be done directly from one workspace to another.

```gauss
WorkspaceHandle_t *w2;

w2 = Gauss_CreateWorkspace("Workspace 2");
rv = Gauss_CopyGlobal(w2, "xinv", w1, "xi");
```

3.2.3 GAUSS Engine Data Structures

The following data structures are used for moving data between the application and the GAUSS Engine. See Chapter 9 for detailed information on the structures.
3. USING THE GAUSS ENGINE

- **Matrix_t**: 2 dimensional matrix, real or complex
- **String_t**: character string
- **StringArray_t**: string array
- **StringElement_t**: string array element

Use the GAUSS Engine API calls to create and free this data. You can create copies of the data or aliases to the data.

If you have a lot of data, you will want to minimize the amount of memory used and the number of times a block of data is copied from one location in memory to another.

Use **GAUSS_Matrix** to create a **Matrix_t** structure. The following code creates a copy of the matrix `x`.

```c
WorkspaceHandle_t *w1;
Matrix_t *mat;
double x[100][20];

w1 = GAUSS_CreateWorkspace( "Workspace 1" );
mat = GAUSS_Matrix( w1, 100, 20, x );
```

The call to **GAUSS_Matrix** calls **malloc** once for the **Matrix_t** structure and once for the matrix data. It then copies the matrix into the newly allocated block.

The following code creates an alias for the matrix `x`.

```c
Matrix_t *matalias;
matalias = GAUSS_MatrixAlias( w1, 100, 20, x );
```

The call to **GAUSS_MatrixAlias** calls **malloc** once for the **Matrix_t** structure. It then sets the data pointer in the **Matrix_t** structure to the address of `x`. No copy is necessary.

The following code frees both `mat` and `matalias`.

```c
GAUSS_FreeMatrix( mat );
GAUSS_FreeMatrix( matalias );
```

The first call above frees both the data block (which is a **malloc**’d copy of `x`) and the **Matrix_t** structure for `mat`. The second call frees only the **Matrix_t** structure for `matalias` because that **Matrix_t** structure contained only an alias to data that the user is left responsible for freeing if necessary.
3. USING THE GAUSS ENGINE

3.2.4 Copying and Moving Data to a Workspace

Use the GAUSS Engine API calls to pass the data between a GAUSS Engine workspace and your application. There are two versions of many of these API calls. One makes a copy of the data (malloc's a new data block) and the other moves the data (gives the data pointer away without any calls to malloc and frees the original structure). The functions are named accordingly.

The following code uses GAUSS_CopyMatrixToGlobal to copy a matrix to the GAUSS Engine. The matrix will be called xm in the workspace.

```gauss
WorkspaceHandle_t *w1;
Matrix_t *mat;
double x[100][20];
int rv;

w1 = GAUSS_CreateWorkspace( "Workspace 1" );
mat = GAUSS_Matrix( w1, 100, 20, x );
rv = GAUSS_CopyMatrixToGlobal( w1, mat, "xm" );
```

The following code uses GAUSS_MoveMatrixToGlobal to move a matrix to the GAUSS Engine and free the Matrix_t structure. The matrix will be called xm in the workspace. The original malloc'd block held by the double pointer x is left intact.

```gauss
WorkspaceHandle_t *w1;
Matrix_t *mat;
double *x;
int r, c;
int rv;

r = 1000;
c = 10;
x = (double *) malloc( r*c*sizeof(double) );
memset( x, 0, r*c*sizeof(double) );
w1 = GAUSS_CreateWorkspace( "Workspace 1" );
mat = GAUSS_Matrix( w1, 100, 20, x );
rv = GAUSS_MoveMatrixToGlobal( w1, mat, "xm" );
```

This can also be accomplished with a nested call, eliminating the need for the intermediate structure. Again, the original malloc'd block held by the double pointer x is left intact.

```gauss
WorkspaceHandle_t *w1;
double *x;
int r, c;
```
3. USING THE GAUSS ENGINE

```c
int rv;

r = 1000;
c = 10;
x = (double *) malloc( r*c*sizeof(double) );
memset( x, 0, r*c*sizeof(double) );
w1 = GAUSS_CreateWorkspace( "Workspace 1" );
rv = GAUSS_MoveMatrixToGlobal( w1, GAUSS_Matrix( w1, r, c, x ), "xm" );
```

A very large `malloc`'d matrix can be given to a workspace without any additional `malloc`'s or copying with `GAUSS_AssignFreeableMatrix`. In the code below, a $1000000 \times 100$ real matrix is created and placed in a workspace.

```c
WorkspaceHandle_t *w1;
double *x;
int r, c;
int rv;

r = 1000000;
c = 100;
x = (double *) malloc( r*c*sizeof(double) );
memset( x, 0, r*c*sizeof(double) );
w1 = GAUSS_CreateWorkspace( "Workspace 1" );
rv = GAUSS_AssignFreeableMatrix( w1, r, c, 0, x, "largex" );
```

After the call to `GAUSS_AssignFreeableMatrix`, the block of memory pointed to by the double pointer `x` is owned by the `GAUSS Engine`. An attempt by the user to free it will cause a fatal error. The `GAUSS Engine` will free the block when necessary.

### 3.2.5 Getting Data From a Workspace

The following code retrieves the matrix `xi` from the workspace to the calling application.

```c
Matrix_t *mat;
mat = GAUSS_GetMatrix( w1, "xi" );
```

The following code checks the type of the symbol `xi` and retrieves it from the workspace to the calling application.

```c
Matrix_t *mat;
StringArray_t *sa;
String_t *st;
int type;
```
mat = NULL;
sa = NULL;
st = NULL;

type = GAUSS_GetSymbolType( w1, "xi" );

switch( type )
{
    case GAUSS_MATRIX:
        mat = GAUSS_GetMatrix( w1, "xi" );
        break;
    case GAUSS_STRING_ARRAY:
        sa = GAUSS_GetStringArray( w1, "xi" );
        break;
    case GAUSS_STRING:
        st = GAUSS_GetString( w1, "xi" );
        break;
    default:
        fprintf( stderr, "Invalid type (%d)\n", type);
        break;
}

3.2.6 Calling Procedures

Two functions are provided to call GAUSS procedures, passing the arguments directly to the calling application and receiving the returns back directly, without the use of globals. Each requires an empty program handle. An empty program handle can be created with GAUSS_CreateProgram.

**GAUSS_CallProc** Calls a GAUSS procedure

**GAUSS_CallProcFreeArgs** Calls a GAUSS procedure and frees the arguments

3.3 Shutdown

When your application has completed using the GAUSS Engine you should call GAUSS_Shutdown before exiting the application.

It is possible to restart the GAUSS Engine by calling GAUSS_Initiaize again after calling GAUSS_Shutdown.
The GAUSS Engine can be used in multi-threaded applications. To achieve the maximum amount of concurrency, you need to structure your application correctly.

The setup and initialization functions should be called from the main thread once at the beginning of the application. The functions that create the matrix, string and string array structures have no associated threading issues. The functions that compile, execute and move data between the application and the GAUSS Engine are discussed below.

If each thread is using a different workspace, there are no associated concurrency issues. The GAUSS Engine API is thread-safe across different workspaces for all functions as long as each workspace has only one associated thread. GAUSS__CopyGlobal will read lock the source workspace and write lock the target workspace as it copies.

There are rules that you can follow to achieve nearly 100% concurrency for multiple threads in a single workspace. Those rules are also discussed below.

### 4.1 Locks

A workspace can have multiple read locks or one write lock. If a thread has a write lock on a workspace, all other threads are blocked until the thread releases the write lock. If a workspace is read locked by one or more threads, any threads requesting write locks are blocked until all the read locks are released.

Two flags are used with the compile functions to guarantee that the program compiled is thread-safe. These are readonlyC and readonlyE for “read only compile” and “read
4. MULTI-THREADED APPLICATIONS

only execute”, respectively. They control workspace locking for compiling and execution of GAUSS code and are used during compiles to trap for code that is not thread-safe. The value of readonlyE is passed to the execute functions, via the program handle.

Be aware that this information is not kept across multiple compiles in the same workspace. Only the values from the compile that created the program handle are passed to the executer. It is therefore possible to make multiple compiles in a workspace and do a readonly compile that succeeds erroneously. The reason for this is that procedures that assign to globals may be resident in the workspace from a previous compile and will not get recompiled each time. If an already resident procedure that assigns to globals is called in a subsequent compile, the global assignment will not be detected.

In practice, this does not usually matter. These arguments are to be used as an aid during development to verify that your code is or is not assigning to globals. They will not prevent you from creating code that is not thread-safe. When your compile fails, it shows you the line of code that violated the rules you specified with the arguments.

4.2 Compiling and Executing GAUSS Programs

GAUSS_CompileFile, GAUSS_CompileString and GAUSS_CompileExpression read lock the workspace when the readonlyC argument is true (non-zero) and write lock the workspace when it is false. When readonlyC is true, the compile will fail if it tries to create or redefine any globals, including procedure definitions. When the readonlyE argument is true, the compile will fail if the program assigns to any globals. The value of readonlyE is passed to the executer, via the program handle.

GAUSS_Execute and GAUSS_ExecuteExpression read lock the workspace if the program was compiled with the readonlyE argument set to true and write lock the workspace otherwise.

4.2.1 Assuring Concurrency

To assure concurrent compilation and execution of multiple threads in a single workspace, design your code so it can be compiled with readonlyC and readonlyE both true for any compiles and executes that you intend to run concurrently in the same workspace.

In practice this usually means you have an initialization cycle (compile and execute) with both flags false to compile and execute the code necessary to define and initialize any global data for a workspace. You then have a second initialization cycle (compile only) with readonlyE true to compile the procedures you need. This data and these procedures can then used in a thread-safe fashion (both flags true) in subsequent compiles and executes in the same workspace.
4. MULTI-THREADED APPLICATIONS

4.3 Calling GAUSS Procedures

The functions \texttt{GAUSS\_CallProc} and \texttt{GAUSS\_CallProcFreeArgs} provide a way to call GAUSS procedures with no globals used for either the arguments or the returns of the procedure. Arguments are passed directly from the application to the procedure via a C structure array and the returns are handled the same way. No globals are necessary in the workspace.

The program handle used with these functions can be created with \texttt{GAUSS\_CompileFile}, \texttt{GAUSS\_CompileString} or \texttt{GAUSS\_CreateProgram}. If the program handle is created with \texttt{readonlyE} true, then \texttt{GAUSS\_CallProc} and \texttt{GAUSS\_CallProcFreeArgs} read lock the workspace, otherwise they use a write lock.

4.3.1 Assuring Concurrency

To assure concurrent execution of multiple threads in a single workspace, design your procedures so they can be compiled with \texttt{readonlyE} true. Assuming a procedure that is listed in a library, the following code illustrates this:

```c
ProgramHandle_t *ph;
char cmd[100];
int readonlyC, readonlyE;

strcpy( cmd, "library mylib; external proc proc1, proc2;" );
readonlyC = 0;
readonlyE = 1;
ph = GAUSS\_CompileString( wh, cmd, readonlyC, readonlyE );
```

If this compile succeeds, you can call the procedures multiple times simultaneously in separate threads and they will execute concurrently. The compile will fail if the procedures contain code that assigns to global variables.
4. MULTI-THREADED APPLICATIONS
Chapter 5

Using the Command Line Interface

ENGAUSS is the command line version of GAUSS, which comes with the GAUSS Engine. The executable file, engauss, is located in the GAUSS Engine installation directory.

The format for using ENGAUSS is:

```
engauss flag(s) program program...
```

- `-b` Execute file in batch mode and then exit. You can execute multiple files by separating file names with spaces.
- `-l logfile` Set the name of batch mode log file when using the `-b` argument. The default is `wksp/gauss.log.###`, where `###` is the pid.
- `-e expression` Executes a GAUSS expression. This command is not logged when GAUSS is in batch mode.
- `-o` Suppresses the sign-on banner (output only).
- `-T` Turns the dataloop translator on.
- `-t` Turns the dataloop translator off.

5.1 Viewing Graphics

GAUSS generates `.tkf` files for graphical output. The default output for graphics is `graphic.tkf`. Two functions are available to convert `.tkf` files to PostScript for printing and viewing with external viewers: the `tkf2ps` function will convert `.tkf` files to PostScript (.ps) files, and the `tkf2eps` function will convert `.tkf` files to encapsulated PostScript (.eps) files. For example, to convert the file `graphic.tkf` to a postscript file named `graphic.ps` use:

```
ret = tkf2ps ("filename.tkf", "filename.ps")
```

If the function is successful it returns 0.
5. USING THE COMMAND LINE INTERFACE

5.2 Interactive Commands

5.2.1 quit

The `quit` command will exit ENGAUSS.

The format for `quit` is:

```
quit
```

You can also use the `system` command to exit ENGAUSS from either the command line or a program (see `system` in the GAUSS Language Reference).

The format for `system` is:

```
system
```

5.2.2 ed

The `ed` command will open an input file in an external text editor, see `ed` in the GAUSS Language Reference.

The format for `ed` is:

```
ed filename
```

5.2.3 compile

The `compile` command will compile a GAUSS program file to a compiled code file.

The format for `compile` is:

```
compile source_file

compile source_file output_file
```

If you do not specify an `output_file`, GAUSS will append a `.gcg` extension to your `source_file` to create an `output_file`. Unlike the `gc` compiler, the `compile` command will not automatically replace a `.gau` extension with a `.gcg` extension. It will append a `.gcg` extension to `.gau` files.
5. **USING THE COMMAND LINE INTERFACE**

5.2.4  **run**

The `run` command will run a **GAUSS** program file or compiled code file.

The format for `run` is:

```
run filename
```

5.2.5  **browse**

The `browse` command allows you to search for specific symbols in a file and open the file in the default editor. You can use wildcards to extend search capabilities of the `browse` command.

The format for `browse` is:

```
browse symbol
```

5.2.6  **config**

The `config` command gives you access to the configuration menu allowing you to change the way **GAUSS** runs and compiles files.

The format for `config` is:

```
cfg
```

**Run Menu**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translator line</td>
<td>Toggles on/off the translation of a file using <strong>dataloop</strong>. The translator is not necessary for <strong>GAUSS</strong> program files not using <strong>dataloop</strong>.</td>
</tr>
<tr>
<td>number tracking</td>
<td>Toggles on/off execution time line number tracking of the original file before translation.</td>
</tr>
<tr>
<td>Line number tracking</td>
<td>Toggles on/off the execution time line number tracking. If the translator is on, the line numbers refer to the translated file.</td>
</tr>
</tbody>
</table>

**Compile Menu**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoload</td>
<td>Toggles on/off the autoloader.</td>
</tr>
<tr>
<td>Autodelete</td>
<td>Toggles on/off autodelete.</td>
</tr>
<tr>
<td><strong>GAUSS</strong> Library</td>
<td>Toggles on/off the <strong>GAUSS</strong> library functions.</td>
</tr>
<tr>
<td>User Library</td>
<td>Toggles on/off the user library functions.</td>
</tr>
</tbody>
</table>
5. USING THE COMMAND LINE INTERFACE

**Declare Warnings**
Toggles on/off the `declare` warning messages during compiling.

**Compiler Trace**
- **Off** Turns off the compiler trace function.
- **On** Turns on the compiler trace function.
- **Line** Traces compilation by line.
- **File** Creates a report of procedures and the local and global symbols they reference.

### 5.3 Debugging

The `debug` command runs a program under the source level debugger.

The format for `debug` is:

```
debug filename
```

**General Functions**

- **?** Displays a list of available commands.
- **q/Esc** Exits the debugger and returns to the GAUSS command line.
- **+/−** Enables/disables the last command repeat function.

**Listing Functions**

- **l number** Displays a specified number of lines of source code in the current file.
- **lc** Displays source code in the current file starting with the current line.
- **ll file line** Displays source code in the named file starting with the specified line.
- **ll file** Displays source code in the named file starting with the first line.
- **ll line** Displays source code starting with the specified line. File does not change.
- **ll** Displays the next page of source code.
- **lp** Displays the previous page of source code.

**Execution Functions**

- **s number** Executes the specified number of lines, stepping over procedures.
- **i number** Executes the specified number of lines, stepping into procedures.
- **x number** Executes code from the beginning of the program to the specified line count, or until a breakpoint is hit.
- **g [[args]]** Executes from the current line to the end of the program, stopping at breakpoints. The optional arguments specify other stopping points.

  The syntax for each optional argument is:
5. USING THE COMMAND LINE INTERFACE

- **filename line cycle**: The debugger will stop every cycle times it reaches the specified line in the named file.
- **filename line**: The debugger will stop when it reaches the specified line in the named file.
- **filename ,, cycle**: The debugger will stop every cycle times it reaches any line in the current file.
- **line cycle**: The debugger will stop every cycle times it reaches the specified line in the current file.
- **filename**: The debugger will stop at every line in the named file.
- **line**: The debugger will stop when it reaches the specified line in the current file.
- **procedure cycle**: The debugger will stop every cycle times it reaches the first line in a called procedure.
- **procedure**: The debugger will stop every time it reaches the first line in a called procedure.

**j [[args]]**: Executes code to a specified line, procedure, or cycle in the file without stopping at breakpoints. The optional arguments are the same as `g`, listed above.

**jx number**: Executes code to the execution count specified (number) without stopping at breakpoints.

**o**: Executes the remainder of the current procedure (or to a breakpoint) and stops at the next line in the calling procedure.

**View Commands**

- **v [[vars]]**: Searches for (a local variable, then a global variable) and displays the value of a specified variable.
- **v$ [[vars]]**: Searches for (a local variable, then a global variable) and displays the specified character matrix.

The display properties of matrices can be set using the following commands:

- **r**: Specifies the number of rows to be shown.
- **c**: Specifies the number of columns to be shown.
- **number, number**: Specifies the number of rows and columns to be shown.
- **w**: Specifies the width of the columns to be shown.
- **p**: Specifies the precision shown.
- **f**: Specifies the format of the numbers as decimal, scientific, or auto format.
- **q**: Quits the matrix viewer.
5. USING THE COMMAND LINE INTERFACE

Breakpoint Commands

**lb**  Shows all the breakpoints currently defined.

**b [[args]]**  Sets a breakpoint in the code. The syntax for each optional argument is:

- `filename line cycle`  The debugger will stop every `cycle` times it reaches the specified `line` in the named file.
- `filename line`  The debugger will stop when it reaches the specified `line` in the named file.
- `filename ,, cycle`  The debugger will stop every `cycle` times it reaches any line in the current file.
- `line cycle`  The debugger will stop every `cycle` times it reaches the specified `line` in the current file.
- `filename`  The debugger will stop at every line in the named file.
- `line`  The debugger will stop when it reaches the specified `line` in the current file.
- `procedure cycle`  The debugger will stop every `cycle` times it reaches the first line in a called procedure.
- `procedure`  The debugger will stop every time it reaches the first line in a called procedure.

**d [[args]]**  Removes a previously specified breakpoint. The optional arguments are the same arguments as **b**, listed above.
The GC compiler can be used in Makefiles or at a system command line to compile GAUSS programs. The syntax is as follows:

```
   gc [ -flags ] -o output_file source_file
   gc [ -flags ] [ -d output_directory ] source_file source_file...
```

The `-o` flag allows you to specify the name of the compiled file. If your `source_file` has a `.gau` extension, the default is to replace the `.gau` extension with `.gcg`. Otherwise, the default is to append `.gcg` to the name of your `source_file`. GAUSS will run compiled files only if they have a `.gcg` extension. Therefore, if you use the `-o` flag to specify an `output_file` name, you should give it a name with a `.gcg` extension.

The `-d` flag allows you to specify the directory in which the compiled files will reside. If you set the `-d` flag, all of the `source_files` you compile in that execution of `gc` will be placed in the specified directory. The default `output_directory` is the current working directory.

To specify a readonly compile or execute, use `-roc` or `-roe`, respectively.
6. *THE GC COMPILER*
7.1 Functions

7.1.1 Pre-initialization setup

These are the first functions called. Use these to set up logging, I/O, error handling and the home directory location.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS.GetHome</td>
<td>Gets the engine home path.</td>
</tr>
<tr>
<td>GAUSS.HookProgramErrorOutput</td>
<td>Sets the callback function for program error output.</td>
</tr>
<tr>
<td>GAUSS.HookProgramInputChar</td>
<td>Sets callback function for key function.</td>
</tr>
<tr>
<td>GAUSS.HookProgramInputCharBlocking</td>
<td>Sets callback function for keyw and show functions.</td>
</tr>
<tr>
<td>GAUSS.HookProgramInputCheck</td>
<td>Sets callback function for keyav function.</td>
</tr>
<tr>
<td>GAUSS.HookProgramInputString</td>
<td>Sets callback function for con and cons functions.</td>
</tr>
<tr>
<td>GAUSS.HookProgramOutput</td>
<td>Sets the callback function for normal program output.</td>
</tr>
<tr>
<td>GAUSS.SetHome</td>
<td>Sets the engine home path directly.</td>
</tr>
<tr>
<td>GAUSS.SetHomeVar</td>
<td>Sets the name of an environment variable containing the home path.</td>
</tr>
<tr>
<td>GAUSS.SetLogFile</td>
<td>Sets the file name and path for logging system errors.</td>
</tr>
<tr>
<td>GAUSS.SetLogStream</td>
<td>Sets the file pointer for logging system errors.</td>
</tr>
</tbody>
</table>
7. C API

7.1.2 Initialization and Shutdown

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS_Initialize</td>
<td>Initializes the engine. Call at the beginning of your application, after setup functions.</td>
</tr>
<tr>
<td>GAUSS_Shutdown</td>
<td>Shuts the engine down. Call prior to ending your application.</td>
</tr>
</tbody>
</table>

7.1.3 Compiling and Executing GAUSS programs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS_CompileExpression</td>
<td>Compiles a right-hand side expression.</td>
</tr>
<tr>
<td>GAUSS_CompileFile</td>
<td>Compiles a file containing GAUSS code.</td>
</tr>
<tr>
<td>GAUSS_CompileString</td>
<td>Compiles a character string containing GAUSS code.</td>
</tr>
<tr>
<td>GAUSS_CompileStringAsFile</td>
<td>Compiles a character string containing GAUSS code as a file.</td>
</tr>
<tr>
<td>GAUSS_CreateWorkspace</td>
<td>Creates a workspace handle.</td>
</tr>
<tr>
<td>GAUSS_Execute</td>
<td>Executes a program.</td>
</tr>
<tr>
<td>GAUSS_ExecuteExpression</td>
<td>Executes a right-hand side expression.</td>
</tr>
<tr>
<td>GAUSS_FreeProgram</td>
<td>Frees a program handle created in a compile.</td>
</tr>
<tr>
<td>GAUSS_FreeWorkspace</td>
<td>Frees a workspace handle.</td>
</tr>
<tr>
<td>GAUSS_LoadCompiledBuffer</td>
<td>Loads a compiled program from a buffer.</td>
</tr>
<tr>
<td>GAUSS_LoadCompiledFile</td>
<td>Loads a compiled program from a file.</td>
</tr>
<tr>
<td>GAUSS_LoadWorkspace</td>
<td>Loads workspace information saved in a file.</td>
</tr>
<tr>
<td>GAUSS_SaveProgram</td>
<td>Saves a compiled program as a file.</td>
</tr>
<tr>
<td>GAUSS_SaveWorkspace</td>
<td>Saves workspace information in a file.</td>
</tr>
<tr>
<td>GAUSS_TranslateDataloopFile</td>
<td>Translates a dataloop file.</td>
</tr>
</tbody>
</table>

7.1.4 Calling Procedures

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS_CallProc</td>
<td>Calls a procedure</td>
</tr>
<tr>
<td>GAUSS_CallProcFreeArgs</td>
<td>Calls a procedure and frees its arguments.</td>
</tr>
<tr>
<td>GAUSS_CopyArgToArg</td>
<td>Copies an argument from one argument list to another.</td>
</tr>
<tr>
<td>GAUSS_CopyArgToMatrix</td>
<td>Copies a matrix from an argument list descriptor to a matrix descriptor.</td>
</tr>
<tr>
<td>GAUSS_CopyArgToString</td>
<td>Copies a string from an argument list descriptor to a string descriptor.</td>
</tr>
<tr>
<td>GAUSS_CopyArgToStringArray</td>
<td>Copies a string array from an argument list descriptor to a string array descriptor.</td>
</tr>
</tbody>
</table>
### C API: Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS__CopyMatrixToArg</td>
<td>Copies a matrix to an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__CopyStringArrayToArg</td>
<td>Copies a string array to an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__CopyStringToArg</td>
<td>Copies a string to an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__CreateArgList</td>
<td>Creates an empty argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__CreateProgram</td>
<td>Creates a program handle to use when calling a procedure.</td>
</tr>
<tr>
<td>GAUSS__DeleteArg</td>
<td>Deletes an argument from an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__FreeArgList</td>
<td>Frees an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__GetArgType</td>
<td>Gets the type of an argument in an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__InsertArg</td>
<td>Inserts an argument in an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__MoveArgToArg</td>
<td>Moves an argument from one argument list to another.</td>
</tr>
<tr>
<td>GAUSS__MoveArgToMatrix</td>
<td>Moves a matrix from an argument list descriptor to a matrix descriptor.</td>
</tr>
<tr>
<td>GAUSS__MoveArgToString</td>
<td>Moves a string from an argument list descriptor to a string descriptor.</td>
</tr>
<tr>
<td>GAUSS__MoveArgToStringArray</td>
<td>Moves a string array from an argument list descriptor to a string array descriptor.</td>
</tr>
<tr>
<td>GAUSS__MoveMatrixToArg</td>
<td>Moves a matrix to an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__MoveStringArrayToArg</td>
<td>Moves a string array to an argument list descriptor.</td>
</tr>
<tr>
<td>GAUSS__MoveStringToArg</td>
<td>Moves a string to an argument list descriptor.</td>
</tr>
</tbody>
</table>

### 7.1.5 Creating and Freeing GAUSS Format Data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS__ComplexMatrix</td>
<td>Creates a matrix descriptor for a complex matrix and copies the matrix.</td>
</tr>
<tr>
<td>GAUSS__ComplexMatrixAlias</td>
<td>Creates a matrix descriptor for a complex matrix.</td>
</tr>
<tr>
<td>GAUSS__FreeMatrix</td>
<td>Frees a matrix descriptor.</td>
</tr>
<tr>
<td>GAUSS__FreeString</td>
<td>Frees a string descriptor.</td>
</tr>
<tr>
<td>GAUSS__FreeStringArray</td>
<td>Frees a string array descriptor.</td>
</tr>
<tr>
<td>GAUSS__Matrix</td>
<td>Creates a matrix descriptor and copies matrix.</td>
</tr>
<tr>
<td>GAUSS__MatrixAlias</td>
<td>Creates a matrix descriptor.</td>
</tr>
<tr>
<td>GAUSS__String</td>
<td>Creates a string descriptor and copies the string.</td>
</tr>
<tr>
<td>GAUSS__StringAlias</td>
<td>Creates a string descriptor.</td>
</tr>
<tr>
<td>GAUSS__StringAliasL</td>
<td>Creates a string descriptor for a string of user-specified length.</td>
</tr>
<tr>
<td>GAUSS__StringArray</td>
<td>Creates a string array descriptor and copies the string array.</td>
</tr>
<tr>
<td>GAUSS__StringArrayL</td>
<td>Creates a string array descriptor for strings of user-specified length and copies the string array.</td>
</tr>
</tbody>
</table>
7. **C API**

**GAUSS\_StringL**

Creates a string descriptor for string of user-specified length and copies the string.

### 7.1.6 Moving Data Between GAUSS and Your Application

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS_AssignFreeableMatrix</td>
<td>Assigns malloc'd data to a global matrix.</td>
</tr>
<tr>
<td>GAUSS_CopyGlobal</td>
<td>Copies a symbol from one workspace to another.</td>
</tr>
<tr>
<td>GAUSS_CopyMatrixToGlobal</td>
<td>Copies a matrix to GAUSS.</td>
</tr>
<tr>
<td>GAUSS_CopyStringToGlobal</td>
<td>Copies a string to GAUSS.</td>
</tr>
<tr>
<td>GAUSS_CopyStringArrayToGlobal</td>
<td>Copies a string array to GAUSS.</td>
</tr>
<tr>
<td>GAUSS_GetDouble</td>
<td>Gets a double from a GAUSS global.</td>
</tr>
<tr>
<td>GAUSS_GetMatrix</td>
<td>Gets a matrix from a GAUSS global.</td>
</tr>
<tr>
<td>GAUSS_GetMatrixAndClear</td>
<td>Gets a matrix from a GAUSS global and clears the global.</td>
</tr>
<tr>
<td>GAUSS_GetMatrixInfo</td>
<td>Gets information for a matrix in a GAUSS global.</td>
</tr>
<tr>
<td>GAUSS_GetString</td>
<td>Gets a string from a GAUSS global.</td>
</tr>
<tr>
<td>GAUSS_GetStringArray</td>
<td>Gets a string array from a GAUSS global.</td>
</tr>
<tr>
<td>GAUSS_GetSymbolType</td>
<td>Gets the type of a symbol in a GAUSS global.</td>
</tr>
<tr>
<td>GAUSS_MoveMatrixToGlobal</td>
<td>Moves a matrix to GAUSS and frees the descriptor.</td>
</tr>
<tr>
<td>GAUSS_MoveStringToGlobal</td>
<td>Moves a string to GAUSS and frees the descriptor.</td>
</tr>
<tr>
<td>GAUSS_MoveStringArrayToGlobal</td>
<td>Moves a string array to GAUSS and frees the descriptor.</td>
</tr>
<tr>
<td>GAUSS_PutDouble</td>
<td>Puts a double into GAUSS.</td>
</tr>
</tbody>
</table>

### 7.1.7 GAUSS Enterprise Engine Error Handling

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAUSS_ErrorText</td>
<td>Gets the text for an error number.</td>
</tr>
<tr>
<td>GAUSS_GetError</td>
<td>Gets the stored error number.</td>
</tr>
<tr>
<td>GAUSS_GetLogFile</td>
<td>Gets the current error log file.</td>
</tr>
<tr>
<td>GAUSS_GetLogStream</td>
<td>Gets the current error log stream.</td>
</tr>
<tr>
<td>GAUSS_SetError</td>
<td>Sets the stored error number.</td>
</tr>
</tbody>
</table>

### 7.2 Include Files

`mteng.h` contains all the function declarations, structure definitions, etc. for the C API. Include it in any C file that references the engine.
Chapter 8

C API: Reference
GAUSS__AssignFreeableMatrix

Purpose

Assigns a malloc’d matrix to a GAUSS workspace.

Format

```c
int GAUSS__AssignFreeableMatrix( WorkspaceHandle_t *wh, unsigned int rows, unsigned int cols, int complex, double *address, char *name );
```

```c
ret = GAUSS__AssignFreeableMatrix( wh, rows, cols, complex, address, name );
```

Input

- `wh` pointer to a workspace handle.
- `rows` number of rows.
- `cols` number of columns.
- `complex` 0 if matrix is real, 1 if complex.
- `address` pointer to matrix.
- `name` pointer to name of matrix to assign to.

Output

- `ret` success flag, 0 if successful, otherwise:
  - 26 Too many symbols.
  - 91 Symbol name too long.
  - 481 GAUSS assignment failed.
  - 495 Workspace inactive or corrupt.

Remarks

GAUSS__AssignFreeableMatrix assigns a matrix that is created using malloc to a GAUSS workspace. GAUSS takes ownership of the matrix and frees it when necessary. The data are not moved or reallocated, making this the most efficient way to move a large matrix to a GAUSS workspace.

Do not attempt to free a matrix that has been assigned to GAUSS with GAUSS__AssignFreeableMatrix. The matrix data should be laid out in row-major order in memory. If the matrix is complex, it should be stored in memory with the entire real part first, followed by the imaginary part.

Call GAUSS__AssignFreeableMatrix with a WorkspaceHandle_t returned from GAUSS__CreateWorkspace.

Example

34
The function above uses `GAUSS_AssignFreeableMatrix` to create a matrix of zeros and assign it to a `GAUSS` workspace. The data are freed if `GAUSS_AssignFreeableMatrix` fails, otherwise `GAUSS` owns the matrix and will free it when necessary.

**See also**

`GAUSS_CopyMatrixToGlobal`, `GAUSS_MoveMatrixToGlobal`, `GAUSS_GetMatrix`, `GAUSS_GetMatrixInfo`
GAUSS_CallProc

- **Purpose**
  Calls a GAUSS procedure.

- **Format**
  
  ```c
  ArgList_t *GAUSS_CallProc( ProgramHandle_t *ph, char *procname, ArgList_t *args );
  
  rets = GAUSS_CallProc( ph, procname, args );
  ```

- **Input**
  
  - `ph` pointer to a program handle.
  - `procname` pointer to name of procedure to be called.
  - `args` pointer to an argument list structure containing the arguments for the procedure.

- **Output**
  
  - `rets` pointer to an argument list structure containing the returns of the procedure.

- **Remarks**
  
  **GAUSS_CallProc** calls a GAUSS procedure that is resident in memory. You pass the arguments to the procedure in an `ArgList_t` structure. Use **GAUSS_CreateArgList** to create an empty `ArgList_t` structure and the following functions to add arguments to it:

    - **GAUSS_CopyMatrixToArg**
    - **GAUSS_CopyStringArrayToArg**
    - **GAUSS_CopyStringToArg**
    - **GAUSS_MoveMatrixToArg**
    - **GAUSS_MoveStringArrayToArg**
    - **GAUSS_MoveStringToArg**

  **GAUSS_CallProc** creates an `ArgList_t` structure in which it puts the returns of the procedure. Use the following functions to move the returns of a procedure from an `ArgList_t` into descriptors for each respective data type:

    - **GAUSS_CopyArgToMatrix**
    - **GAUSS_CopyArgToString**
    - **GAUSS_CopyArgToStringArray**
    - **GAUSS_MoveArgToMatrix**
    - **GAUSS_MoveArgToString**
    - **GAUSS_MoveArgToStringArray**

36
8. C API: REFERENCE

Use GAUSS__GetArgType to get the type of an argument in the ArgList_t.

It is your responsibility to free both the ArgList_t returned from GAUSS__CallProc and the one passed in. They may be freed with GAUSS__FreeArgList.

Call GAUSS__CallProc with a ProgramHandle_t created with GAUSS__CreateProgram.

If GAUSS__CallProc fails, rets will be NULL. Use GAUSS__GetError to get the number of the error. GAUSS__CallProc may fail with any of the following errors:

- 30  Insufficient memory.
- 298 NULL program handle.
- 470 Symbol not found.
- 478 NULL procedure name.
- 493 Program execute failed.

Example

```c
ProgramHandle_t *ph;
ArgList_t *args, *rets;

ph = GAUSS_CreateProgram( wh, 0 );
args = GAUSS_CreateArgList();

if ( GAUSS_MoveStringToArg( args, GAUSS_String( "" ), 0 ) )
{
    char buff[100];

    printf( "MoveStringToArg failed: %s\n", 
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( args );
    return -1;
}

if ( GAUSS_MoveMatrixToArg( args, GAUSS_GetMatrix( wh, "a" ), 0 ) )
{
    char buff[100];

    printf( "MoveMatrixToArg failed: %s\n", 
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( args );
    return -1;
}
```
if ( GAUSS_MoveMatrixToArg( args, GAUSS_GetMatrix( wh, "b" ), 0 ) )
{
    char buff[100];

    printf( "MoveMatrixToArg failed: %s
", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( args );
    return -1;
}

if ( ( rets = GAUSS_CallProc( ph, "ols", args ) ) == NULL )
{
    char buff[100];

    printf( "CallProc failed: %s
", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( args );
    return -1;
}

This example assumes that \( wh \) is the pointer to a valid workspace handle and that \( a \) and \( b \) are both matrices that are already resident in \( wh \). It calls the procedure \( ols \) with the arguments contained in \( args \).

**See also**

- `GAUSS_CallProcFreeArgs`
- `GAUSS_CreateProgram`
- `GAUSS_CreateArgList`
- `GAUSS_FreeArgList`
- `GAUSS_GetArgType`
8. C API: REFERENCE

**Purpose**

Calls a GAUSS procedure and frees the argument list.

**Format**

\[
\text{ArgList}_t \times \text{GAUSS\_CallProcFreeArgs}( \text{ProgramHandle}_t \times ph, \text{char} * \text{procname}, \text{ArgList}_t \times args )
\]

\[
rets = \text{GAUSS\_CallProcFreeArgs}( ph, \text{procname}, args )
\]

**Input**

- \( ph \) pointer to a program handle.
- \( \text{procname} \) pointer to name of procedure to be called.
- \( \text{args} \) pointer to an argument list structure containing the arguments for the procedure.

**Output**

- \( rets \) pointer to the argument list structure containing the returns for the procedure.

**Remarks**

\text{GAUSS\_CallProcFreeArgs} is similar to \text{GAUSS\_CallProc}; however, the \text{ArgList}_t structure that you pass in will be rewritten with the returns from the procedure. This function saves both time and memory space.

Use \text{GAUSS\_CreateArgList} to create an empty \text{ArgList}_t structure and the following functions to add arguments to it:

- \text{GAUSS\_CopyMatrixToArg}
- \text{GAUSS\_CopyStringArrayToArg}
- \text{GAUSS\_CopyStringToArg}
- \text{GAUSS\_MoveMatrixToArg}
- \text{GAUSS\_MoveStringArrayToArg}
- \text{GAUSS\_MoveStringToArg}

\text{GAUSS\_CallProcFreeArgs} returns a pointer to \text{args}, which has been rewritten with the returns of the procedure. Use the following functions to move the returns of a procedure from an \text{ArgList}_t into descriptors for each respective data type:
Use `GAUSS_GetArgType` to get the type of an argument in the ArgList_t.

Call `GAUSS_CallProcFreeArgs` with a ProgramHandle_t created with `GAUSS_CreateProgram`.

If `GAUSS_CallProcFreeArgs` fails, rets will be NULL. Use `GAUSS_GetError` to get the number of the error. `GAUSS_CallProcFreeArgs` may fail with any of the following errors:

- 30 Insufficient memory.
- 298 NULL program handle.
- 470 Symbol not found.
- 478 NULL procedure name.
- 479 NULL argument list.
- 493 Program execute failed.

### Example

```c
ProgramHandle_t *ph
ArgList_t *args;

ph = GAUSS_CreateProgram( wh, 0 );
args = GAUSS_CreateArgList();

if ( GAUSS_MoveStringToArg( args, GAUSS_String(""), 0 ) )
{
    char buff[100];

    printf( "MoveStringToArg failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( args );
    return -1;
}

if ( GAUSS_MoveMatrixToArg( args, GAUSS_GetMatrix(wh, "x"), 0 ) )
{
    char buff[100];
```
printf( "MoveMatrixToArg failed: %s\n",
    GAUSS_ErrorText( buff, GAUSS_GetError() ) );
GAUSS_FreeProgram( ph );
GAUSS_FreeArgList( args );
return -1;
}

if ( ( args = GAUSS_CallProcFreeArgs( ph, "dstat", args ) ) == NULL )
{
    char buff[100];
    printf( "CallProcFreeArgs failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( args );
    return -1;
}

This example calls the procedure \texttt{dstat}, which is in the Run-Time Library. It assumes that \texttt{wh} is a pointer to a valid workspace handle. The example creates the \texttt{ArgList_t} \texttt{args}, and adds two arguments to it, assuming that \texttt{x} is already resident in \texttt{wh}. It then calls the procedure \texttt{dstat} with the arguments contained in \texttt{args}.

\textbf{See also}

\begin{itemize}
    \item \texttt{GAUSS\_CallProc}, \texttt{GAUSS\_CreateProgram}, \texttt{GAUSS\_CreateArgList},
    \texttt{GAUSS\_FreeArgList}, \texttt{GAUSS\_GetArgType}, \texttt{GAUSS\_InsertArg}, \texttt{GAUSS\_DeleteArg}
\end{itemize}
**GAUSS__CompileExpression**

<table>
<thead>
<tr>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiles an expression.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ProgramHandle_t *GAUSS__CompileExpression( WorkspaceHandle_t *wh, char *str, int readonlyC, int readonlyE );</code></td>
</tr>
</tbody>
</table>

```c
ph = GAUSS__CompileExpression( wh, str, readonlyC, readonlyE );
```

<table>
<thead>
<tr>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wh</code> pointer to a workspace handle.</td>
</tr>
<tr>
<td><code>str</code> pointer to string containing expression.</td>
</tr>
<tr>
<td><code>readonlyC</code> 1 or 0, if 1, the compile cannot create or redefine any global symbols. See Section 4.1.</td>
</tr>
<tr>
<td><code>readonlyE</code> 1 or 0, if 1, the program cannot assign to global symbols.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ph</code> pointer to a program handle.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>This function compiles an expression and creates a <code>ProgramHandle_t</code>. An expression is the right-hand side of an assignment statement without the assignment, for example:</td>
</tr>
<tr>
<td><code>x*y + z*inv( k );</code></td>
</tr>
<tr>
<td><code>diag( chol( x'x ) );</code></td>
</tr>
</tbody>
</table>

Follow `GAUSS__CompileExpression` by a call to `GAUSS__ExecuteExpression` to run the code just compiled. Use the program handle pointer returned from the compile as the input for the execute. `GAUSS__ExecuteExpression` returns an `ArgList_t`, which contains the returns from the expression.

If `GAUSS__CompileExpression` fails, `ph` will be NULL. Use `GAUSS__GetError` to get the number of the error. `GAUSS__CompileExpression` may fail with any of the following errors:

- 6 Statement too long.
- 30 Insufficient memory.
- 495 Workspace inactive or corrupt.
- 511 GAUSS compile error.
Example

```c
ProgramHandle_t *ph;
ArgList_t *ret;
Matrix_t *mat;

ph = GAUSS_CompileExpression( wh, "inv( x ) * x", 1, 1 );

if ( ph == NULL ) {
    char buff[100];
    printf( "Compile failed: %s
",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL ) {
    char buff[100];
    printf( "Execute failed: %s
",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}
```

The example code above assumes that \( x \) is already resident in the workspace \( \text{wh} \). \text{GAUSS__ExecuteExpression} creates the \text{ArgList_t}, \text{ret}, which contains the return from the executed expression.

See also

\text{GAUSS__CompileFile}, \text{GAUSS__CompileString}, \text{GAUSS__CompileStringAsFile}, \text{GAUSS__ExecuteExpression}
GAUSS__CompileFile

- **Purpose**
  Compiles a file, creating a program handle.

- **Format**
  
  ```
  ProgramHandle_t *GAUSS__CompileFile( WorkspaceHandle_t *wh, char *fn, int readonlyC, int readonlyE );
  
  ph = GAUSS__CompileFile( wh, fn, readonlyC, readonlyE );
  ```

- **Input**
  
  - `wh` pointer to a workspace handle.
  - `fn` pointer to file name.
  - `readonlyC` 1 or 0, if 1, the compile cannot create or redefine any global symbols. See Section 4.1.
  - `readonlyE` 1 or 0, if 1, the program cannot assign to global symbols.

- **Output**
  
  - `ph` pointer to a program handle.

- **Remarks**
  
  Follow `GAUSS__CompileFile` by a call to `GAUSS__Execute` to run the program just compiled. Use the program handle pointer returned from the compile as the input for the execute.

  Call `GAUSS__CompileFile` with a `WorkspaceHandle_t` pointer returned from `GAUSS__CreateWorkspace`.

  If `GAUSS__CompileFile` fails, `ph` will be NULL. Use `GAUSS__GetError` to get the number of the error. `GAUSS__CompileFile` may fail with either of the following errors:

  - 30 Insufficient memory.
  - 495 Workspace inactive or corrupt.
  - 511 GAUSS compile error.

- **Example**
The example code above runs the GAUSS example file `ols.e`. It assumes that `wh` is a valid workspace handle.

- **See also**

  GAUSS__CompileString, GAUSS__CompileStringAsFile, GAUSS__CompileExpression, GAUSS__Execute
GAUSS__CompileString

- **Purpose**
  Compiles a character string, returning a program handle.

- **Format**
  
  ```c
  ProgramHandle_t *GAUSS__CompileString( WorkspaceHandle_t *wh, char *str,
   int readonlyC, int readonlyE );
  
  ph = GAUSS__CompileString( wh, str, readonlyC, readonlyE );
  ```

- **Input**
  
  - `wh` pointer to a workspace handle.
  - `str` pointer to string to compile.
  - `readonlyC` 1 or 0, if 1, the compile cannot create or redefine any global symbols. See Section 4.1.
  - `readonlyE` 1 or 0, if 1, the program cannot assign to global symbols.

- **Output**
  
  - `ph` pointer to a program handle.

- **Remarks**
  Follow `GAUSS__CompileString` by a call to `GAUSS__Execute` to run the program just compiled. Use the program handle pointer returned from the compile as the input for the execute.

  Call `GAUSS__CompileString` with a `WorkspaceHandle_t` returned from `GAUSS__CreateWorkspace`.

  If `GAUSS__CompileString` fails, `ph` will be NULL. Use `GAUSS__GetErr` to get the number of the error. `GAUSS__CompileString` may fail with either of the following errors:

  - 30 Insufficient memory.
  - 495 Workspace inactive or corrupt.
  - 511 GAUSS compile error.

- **Example**

46
8. **C API: REFERENCE**

```c
ProgramHandle_t *ph;
int ret;

if ( ( ph = GAUSS_CompileString(
    wh,
    "a = rndn(3, 3); b = ones(3, 1); c = diagrv(a, b);",
    0,
    0
) ) == NULL )
{
    char buff[100];
    printf( "Compile failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];
    printf( "Execute failed: %s\n",
            GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}
```

The example above assumes that `wh` is a pointer to a valid workspace handle.

- **See also**
  - `GAUSS_CompileStringAsFile`, `GAUSS_CompileFile`, `GAUSS_CompileExpression`, `GAUSS_Execute`
GAUSS\_CompileStringAsFile

- **Purpose**
  Compiles a string as a file.

- **Format**

  ```c
  ProgramHandle\_t \*GAUSS\_CompileStringAsFile( WorkspaceHandle\_t \*wh, char \*fn, int readonlyC, int readonlyE);
  ```

  ```c
  ph = GAUSS\_CompileStringAsFile( wh, fn, readonlyC, readonlyE );
  ```

- **Input**
  - `wh` pointer to a workspace handle.
  - `fn` pointer to file name.
  - `readonlyC` 1 or 0, if 1, the compile cannot create or redefine any global symbols. See Section 4.1.
  - `readonlyE` 1 or 0, if 1, the program cannot assign to global symbols.

- **Output**
  - `ph` pointer to a program handle.

- **Remarks**

  This function compiles a string into memory by first writing it to a temporary file and then compiling the file. This is typically used to diagnose compile errors. The compiler will report line numbers. To make this really useful as a diagnostic tool, separate multiple statements in the string with linefeeds.

  Follow `GAUSS\_CompileStringAsFile` by a call to `GAUSS\_Execute` to run the program just compiled. Use the program handle pointer returned from the `Compile` as the input for the `Execute`.

  Call `GAUSS\_CompileStringAsFile` with a `WorkspaceHandle\_t` returned from `GAUSS\_CreateWorkspace`.

  If `GAUSS\_CompileStringAsFile` fails, `ph` will be NULL. Use `GAUSS\_GetError` to get the number of the error. `GAUSS\_CompileStringAsFile` may fail with any of the following errors:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Insufficient memory.</td>
</tr>
<tr>
<td>83</td>
<td>Error creating temporary file.</td>
</tr>
<tr>
<td>495</td>
<td>Workspace inactive or corrupt.</td>
</tr>
<tr>
<td>500</td>
<td>Cannot create temporary filename.</td>
</tr>
</tbody>
</table>
Example

```c
ProgramHandle_t *ph;
int ret;

if ( ( ph = GAUSS_CompileString(wh,
   "a = rndn(3, 3);\nb = ones(3, 1);\nc = diagrv(a, b);",
   0,
   0
 ) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];

    printf( "Execute failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}
```

The example above assumes that `wh` is a pointer to a valid workspace handle.

See also

- `GAUSS_CompileString`
- `GAUSS_CompileFile`
- `GAUSS_CompileExpression`
- `GAUSS_Execute`
GAUSS\_ComplexMatrix

- **Purpose**
  Creates a `Matrix\_t` for a complex matrix and copies the matrix data.

- **Format**
  ```c
  Matrix\_t *GAUSS\_ComplexMatrix( unsigned int rows, unsigned int cols, double *real, double *imag );
  ```
  ```c
  mat = GAUSS\_ComplexMatrix( rows, cols, real, imag );
  ```

- **Input**
  - `rows` number of rows.
  - `cols` number of columns.
  - `real` pointer to real part of matrix.
  - `imag` pointer to imaginary part of matrix.

- **Output**
  - `mat` pointer to a matrix descriptor.

- **Remarks**
  GAUSS\_ComplexMatrix `malloc`'s a `Matrix\_t` and fills it in with your input information. It makes a copy of the matrix and sets the `mdata` member of the `Matrix\_t` to point to the copy. GAUSS\_ComplexMatrix should be used only for complex matrices. To create a `Matrix\_t` for a real matrix, use GAUSS\_Matrix. To create a `Matrix\_t` for a complex matrix without making a copy of the matrix, use GAUSS\_ComplexMatrixAlias.

  Set `imag` to NULL if the matrix is stored in memory with each real entry followed by its corresponding imaginary entry. Otherwise, set `imag` to point to the block of memory that contains the imaginary part of the matrix.

  If `mat` is NULL, there was insufficient memory to `malloc` space for the matrix and its descriptor.

  Use this function to create a matrix descriptor that you can use in the following functions:
  ```c
  GAUSS\_CopyMatrixToArg
  GAUSS\_CopyMatrixToGlobal
  GAUSS\_MoveMatrixToArg
  GAUSS\_MoveMatrixToGlobal
  ```
  You can free the `Matrix\_t` with GAUSS\_FreeMatrix.

- **Example**

50
double mr[3][3] = {{3, -4, 6}, {1, 2, 3}, {4, 8, -2}};
double mi[3][3] = {{8, 0, -1}, {-13, 5, 2}, {6, 7, 4}};

if (GAUSS_MoveMatrixToGlobal(
    wh,
    GAUSS_ComplexMatrix(3, 3, &mr[0][0], &mi[0][0]),
    "a"
)) {
    char buff[100];
    printf("MoveMatrixToGlobal failed: %s\n",
           GAUSS_ErrorText(buff, GAUSS_GetError()));
    return -1;
}

The above example assumes that wh is a pointer to a valid workspace handle.

See also

GAUSS_ComplexMatrixAlias, GAUSS_Matrix, GAUSS_CopyMatrixToGlobal,
GAUSS_CopyMatrixToArg, GAUSS_MoveMatrixToGlobal,
GAUSS_MoveMatrixToArg, GAUSS_FreeMatrix
Purpose

Creates a Matrix_t for a complex matrix.

Format

Matrix_t *GAUSS_ComplexMatrixAlias( unsigned int rows, unsigned int cols, double *addr );

mat = GAUSS_ComplexMatrixAlias( rows, cols, addr );

Input

rows number of rows.

cols number of columns.

addr pointer to matrix.

Output

mat pointer to a matrix descriptor.

Remarks

GAUSS_ComplexMatrixAlias is similar to GAUSS_ComplexMatrix; however, it sets the mdata member of the Matrix_t to point to addr instead of making a copy of the matrix. GAUSS_ComplexMatrixAlias should be used only for complex matrices. The matrix data must be stored with the entire real part first, followed by the imaginary part. For real matrices, use GAUSS_Matrix.

If mat is NULL, there was insufficient memory to malloc space for the matrix descriptor.

Use this function to create a matrix descriptor that you can use in the following functions:

GAUSS_CopyMatrixToArg
GAUSS_CopyMatrixToGlobal
GAUSS_MoveMatrixToArg
GAUSS_MoveMatrixToGlobal

You can free the Matrix_t with GAUSS_FreeMatrix. It will not free the matrix data if the Matrix_t was created with GAUSS_ComplexMatrixAlias.

Example

52
Matrix_t *mat;
double *x;

x = (double *)malloc( 12*sizeof(double) );
memset( x, 0, 12*sizeof(double) );

if ( ( mat = GAUSS_ComplexMatrixAlias( 3, 2, &x ) ) == NULL )
{
    char buff[100];
    printf( "ComplexMatrixAlias failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( GAUSS_CopyMatrixToGlobal( wh, mat, "a" ) )
{
    char buff[100];
    printf( "CopyMatrixToGlobal failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeMatrix( mat );
    return -1;
}

This example malloc’s a matrix of zeroes and then uses that matrix data to create a Matrix_t for a complex matrix. It copies the matrix to wh, which it assumes to be a pointer to a valid workspace.

See also

GAUSS_ComplexMatrix, GAUSS_MatrixAlias, GAUSS_CopyMatrixToGlobal,
GAUSS_CopyMatrixToArg, GAUSS_MoveMatrixToGlobal,
GAUSS_MoveMatrixToArg, GAUSS_FreeMatrix
GAUSS__CopyArgToArg

■ Purpose
Copies an argument from one ArgList_t to another.

■ Format

```c
int GAUSS__CopyArgToArg( ArgList_t *targs, int targnum, ArgList_t *sargs, int sargnum );
```

```c
ret = GAUSS__CopyArgToArg( targs, targnum, sargs, sargnum );
```

■ Input

targs pointer to target argument list structure.
targnum number of argument in target argument list.
sargs pointer to source argument list structure.
sargnum number of argument in source argument list.

■ Output

ret success flag, 0 if successful, otherwise:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Insufficient workspace memory.</td>
</tr>
<tr>
<td>94</td>
<td>Argument out of range.</td>
</tr>
</tbody>
</table>

■ Remarks

GAUSS__CopyArgToArg copies the sargnum argument in sargs and assigns it to targs.

To add an argument to the end of an argument list or to an empty argument list, set targnum to 0. To replace an argument, set targnum to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call GAUSS__InsertArg and then set targnum to the number of the inserted argument. Arguments are numbered starting with 1.

The copy of the argument’s data will be freed when you call GAUSS__CallProcFreeArgs or GAUSS__FreeArgList later.

This function allows you to retain the argument in sargs. If you want to move the argument to targs, use GAUSS__MoveArgToArg instead.

■ Example

54
```c
ArgList_t *carg( WorkspaceHandle_t *wh, ArgList_t *args )
{
    ProgramHandle_t *ph;
    ArgList_t *ret;

    if ( ( ph = GAUSS_CompileExpression(wh,
                                           "sin( seqa( 0,.2*pi(),50 );",
                                           1,
                                           1
                                           ) ) == NULL )
    { char buff[100];
        printf( "Compile failed: %s\n",
                 GAUSS_ErrorText( buff, GAUSS_GetError() ) );
        return NULL;
    }

    if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
    { char buff[100];
        printf( "Execute failed: %s\n",
                 GAUSS_ErrorText( buff, GAUSS_GetError() ) );
        GAUSS_FreeProgram( ph );
        return NULL;
    }

    if ( GAUSS_CopyArgToArg( args, 3, ret, 1 ) )
    { char buff[100];
        printf( "CopyArgToArg failed: %s\n",
                 GAUSS_ErrorText( buff, GAUSS_GetError() ) );
        GAUSS_FreeProgram( ph );
        GAUSS_FreeArgList( ret );
        return NULL;
    }

    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( ret );

    return args;
}
```
This example compiles an expression in `wh`, which gives its return in an `ArgList_t`. It copies the return contained in `ret` into `args` as its third argument. It assumes that `args` contains at least three arguments, and it overwrites the third argument of `args`.

**See also**

`GAUSS_CopyArgToArg`, `GAUSS_CreateArgList`, `GAUSS_FreeArgList`, `GAUSS_InsertArg`, `GAUSS_DeleteArg`, `GAUSS_CallProc`, `GAUSS_CallProcFreeArgs`
Purpose
Copies a matrix from an ArgList_t to a Matrix_t structure.

Format
Matrix_t *GAUSS_CopyArgToMatrix( ArgList_t *args, int argnum );

mat = GAUSS_CopyArgToMatrix( args, argnum );

Input
args pointer to an argument list structure.
argnum argument number.

Output
mat pointer to a matrix descriptor.

Remarks
GAUSS_CopyArgToMatrix creates a matrix descriptor, mat, and copies a matrix contained in args into it. mat belongs to you. Free it with GAUSS_FreeMatrix.

Arguments in an ArgList_t are numbered starting with 1.

This function allows you to retain the matrix in the ArgList_t. If you want to move the matrix from the ArgList_t into a Matrix_t, use GAUSS_MoveArgToMatrix.

If GAUSS_CopyArgToMatrix fails, mat will be NULL. Use GAUSS_GetError to get the number of the error. GAUSS_CopyArgToMatrix may fail with any of the following errors:

30 Insufficient memory.
71 Type mismatch.
94 Argument out of range.

Example

ProgramHandle_t *ph;
ArgList_t *arg;
Matrix_t *mat;

if ( ( ph = GAUSS_CompileExpression( wh,
The above example copies the matrix returned from an executed expression into a `Matrix_t`. It assumes that `wh` is a pointer to a valid workspace handle. It retains `arg`, which should be freed later with `GAUSS_FreeArgList`.

**See also**

- `GAUSS_MoveMatrixToArg`
- `GAUSS_CallProc`
- `GAUSS_CallProcFreeArgs`
- `GAUSS_ExecuteExpression`
- `GAUSS_GetArgType`
- `GAUSS_FreeArgList`
8. C API: REFERENCE

GAUSS_CopyArgToString

- **Purpose**
  Copies a string from an ArgList_t to a String_t structure.

- **Format**
  ```c
  String_t *GAUSS_CopyArgToString( ArgList_t *args, int argnum );
  str = GAUSS_CopyArgToString( args, argnum );
  ```

- **Input**
  - `args` pointer to an argument list structure.
  - `argnum` number of argument in the argument list.

- **Output**
  - `str` pointer to a string descriptor.

- **Remarks**
  GAUSS_CopyArgToString creates a string descriptor, `str`, and copies a string contained in `args` into it. `str` belongs to you. Free it with GAUSS_FreeString.

  Arguments in an ArgList_t are numbered starting with 1.

  This function allows you to retain the string in the ArgList_t. If you want to move the string from the ArgList_t into a String_t, use GAUSS_MoveArgToString.

  If GAUSS_CopyArgToString fails, `str` will be NULL. Use GAUSS_GetError to get the number of the error. GAUSS_CopyArgToString may fail with any of the following errors:
  - 30 Insufficient memory.
  - 71 Type mismatch.
  - 94 Argument out of range.

- **Example**
  ```c
  ProgramHandle_t *ph;
  ArgList_t *arg;
  String_t *str;
  
  if ( ( ph = GAUSS_CompileExpression( 
    wh,
    ```
The above example code copies the string returned from an executed expression into a `String_t`. It assumes that `wh` is a pointer to a valid workspace handle. It retains `arg`, which should be freed later with `GAUSS_FreeArgList`.

**See also**

- `GAUSS_MoveArgToString`
- `GAUSS_CallProc`
- `GAUSS_CallProcFreeArgs`
- `GAUSS_ExecuteExpression`
- `GAUSS_GetArgType`
- `GAUSS_FreeArgList`
GAUSS_copyArgToStringArray

- **Purpose**
  Copies a string array from an `ArgList_t` to a `StringArray_t` structure.

- **Format**
  ```c
  StringArray_t *GAUSS_copyArgToStringArray( ArgList_t *args, int argnum );
  
  sa = GAUSS_copyArgToStringArray( args, argnum );
  ```

- **Input**
  - `args`: pointer to an argument list structure.
  - `argnum`: number of argument in the argument list.

- **Output**
  - `sa`: pointer to a string array descriptor.

- **Remarks**
  `GAUSS_copyArgToStringArray` creates a string array descriptor, `sa`, and copies a string array contained in `args` into it. `sa` belongs to you. Free it with `GAUSS_freeStringArray`.

  Arguments in an `ArgList_t` are numbered starting with 1.

  This function allows you to retain the string array in the `ArgList_t`. If you want to move the string array from the `ArgList_t` into a `StringArray_t`, use `GAUSS_moveArgToStringArray`.

  If `GAUSS_copyArgToStringArray` fails, `sa` will be NULL. Use `GAUSS_getError` to get the number of the error. `GAUSS_copyArgToStringArray` may fail with any of the following errors:

  - 30: Insufficient memory.
  - 71: Type mismatch.
  - 94: Argument out of range.

- **Example**
  ```c
  ProgramHandle_t *ph;
  ArgList_t *arg;
  StringArray_t *sa;
  ```
The above example copies the string array returned from an executed expression into a `StringArray_t`. It assumes that `wh` is a pointer to a valid workspace handle. It retains `arg`, which should be freed later with `GAUSS_FreeArgList`.

**See also**

- `GAUSS_MoveArgToStringArray`
- `GAUSS_CallProc`
- `GAUSS_CallProcFreeArgs`
- `GAUSS_ExecuteExpression`
- `GAUSS_GetArgType`
- `GAUSS_FreeArgList`
**Purpose**

Copies a global symbol from one **GAUSS** workspace to another.

**Format**

```c
int GAUSS_CopyGlobal( WorkspaceHandle_t *twh, char *tname, WorkspaceHandle_t *swh, char *sname );
```

```c
ret = GAUSS_CopyGlobal( twh, tname, swh, sname );
```

**Input**

- **twh**: pointer to target workspace handle.
- **tname**: pointer to name of target symbol.
- **swh**: pointer to source workspace handle.
- **sname**: pointer to name of source symbol.

**Output**

- **ret**: success flag, 0 if successful, otherwise:
  - 71: Type mismatch.
  - 91: Symbol too long.
  - 470: Symbol not found.
  - 495: Workspace inactive or corrupt.

**Remarks**

**GAUSS_CopyGlobal** can be used to copy a global symbol from one **GAUSS** workspace to another or to save a global symbol under a different name in the same **GAUSS** workspace.

Call **GAUSS_CopyGlobal** with a **WorkspaceHandle_t** pointer returned from **GAUSS_CreateWorkspace**.

**Example**

```c
int cpg( WorkspaceHandle_t *wh1, WorkspaceHandle_t *wh2 )
{
    ProgramHandle_t *ph;
    int ret;
```
if (( ph = GAUSS_CompileString(
    wh1, 
    "{ a, rs } = rndKMn( 3, 4, 31 );", 
    0, 
    0 
)) == NULL)
{
    char buff[100];
    
    printf( "Compile failed: %s\n", 
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];
    
    printf( "Execute failed: %s\n", 
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( ret = GAUSS_CopyGlobal( wh2, "rmat", wh1, "a" ) )
{
    char buff[100];
    
    printf( "Assign failed for rmat: %s\n", 
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1
}

return 0;

The above example copies the matrix \( \text{a} \) from \( \text{wh1} \) into \( \text{wh2} \) and calls the matrix copy \( \text{rmat} \).

- See also

GAUSS_GetMatrix, GAUSS_GetString, GAUSS_GetStringArray
8. C API: REFERENCE

GAUSS__CopyMatrixToArg

- **Purpose**
  Copies a matrix contained in a `Matrix_t` to an `ArgList_t`.

- **Format**
  ```c
  int GAUSS__CopyMatrixToArg( ArgList_t *args, Matrix_t *mat, int argnum );
  ret = GAUSS__CopyMatrixToArg( args, mat, argnum );
  ```

- **Input**
  - `args` pointer to an argument list structure.
  - `mat` pointer to a matrix descriptor.
  - `argnum` argument number.

- **Output**
  - `ret` success flag, 0 if successful, otherwise:
    - 30 Insufficient memory.
    - 494 Invalid argument number.

- **Remarks**
  GAUSS__CopyMatrixToArg `malloc`’s a copy of the matrix contained in the `mat` argument and assigns the copy to `args`.

  To add an argument to the end of an argument list or to an empty argument list, set `argnum` to 0. To replace an argument, set `argnum` to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call GAUSS__InsertArg and then set `argnum` to the number of the inserted argument. Arguments are numbered starting with 1.

  The matrix copy will be freed when you call GAUSS__CallProcFreeArgs or GAUSS__FreeArgList later.

  This function allows you to retain `mat`. If you want to move the matrix to the argument list and free the `Matrix_t`, use GAUSS__MoveMatrixToArg instead.

  Call GAUSS__CopyMatrixToArg with a `Matrix_t` that was returned from one of the following functions:

  - `GAUSS__ComplexMatrix`
  - `GAUSS__ComplexMatrixAlias`
  - `GAUSS__GetMatrix`
  - `GAUSS__Matrix`
  - `GAUSS__MatrixAlias`

- **Example**
The above example copies the matrix $\mathbf{md}$ into the `ArgList_t arg`. It retains `mat`, which should be freed later with `GAUSS_FreeMatrix`.

- **See also**
  
  `GAUSS_MoveMatrixToArg`, `GAUSS_Matrix`, `GAUSS_ComplexMatrix`,
  `GAUSS_CreateArgList`, `GAUSS_FreeArgList`, `GAUSS_InsertArg`,
  `GAUSS_CallProc`, `GAUSS_CallProcFreeArgs`
8. C API: REFERENCE

GAUSS_CopyMatrixToGlobal

- **Purpose**
  Copies a matrix contained in a Matrix_t into a GAUSS workspace.

- **Format**
  ```c
  int GAUSS_CopyMatrixToGlobal( WorkspaceHandle_t *wh, Matrix_t *mat, char *name );
  ```
  ```c
  ret = GAUSS_CopyMatrixToGlobal( wh, mat, name );
  ```

- **Input**
  - `wh` pointer to a workspace handle.
  - `mat` pointer to a matrix descriptor.
  - `name` pointer to name of matrix.

- **Output**
  - `ret` success flag, 0 if successful, otherwise:
    - 26 Too many symbols.
    - 30 Insufficient memory.
    - 91 Symbol too long.
    - 481 GAUSS assignment failed.

- **Remarks**
  GAUSS_CopyMatrixToGlobal malloc's a copy of the matrix contained in `mat` and assigns the copy to `wh`. GAUSS frees it when necessary. This function allows you to retain `mat`.

  If you want to move the matrix to `wh` and free the Matrix_t, use GAUSS_MoveMatrixToGlobal instead.

  Call GAUSS_CopyMatrixToGlobal with a Matrix_t returned from one of the following functions:
  ```c
  GAUSS_ComplexMatrix
  GAUSS_ComplexMatrixAlias
  GAUSS_GetMatrix
  GAUSS_Matrix
  GAUSS_MatrixAlias
  ```

  Input a WorkspaceHandle_t returned from GAUSS_CreateWorkspace.

- **Example**
Matrix_t *mat;
double md[2][3] = {{3, 4, 2}, {7, 9, 5}};
int ret;

mat = GAUSS_Matrix(2, 3, &md[0][0]);

if ( ret = GAUSS_CopyMatrixToGlobal(wh, mat, "a") ) {
    char buff[100];
    printf("CopyMatrixToGlobal failed: %s\n", GAUSS_ErrorText(buff, ret));
    GAUSS_FreeMatrix(mat);
    return -1;
}

The above example copies the matrix md into the GAUSS workspace indicated by wh. It assumes that wh is a pointer to a valid workspace handle. It retains mat, which should be freed later with GAUSS_FreeMatrix.

See also

GAUSS_MoveMatrixToGlobal, GAUSS_Matrix, GAUSS_ComplexMatrix, GAUSS_AssignFreeableMatrix, GAUSS_GetMatrix, GAUSS_PutDouble
8. C API: REFERENCE

GAUSS__CopyStringArrayToArg

- **Purpose**

  Copies a string array contained in a StringArray_t to an ArgList_t.

- **Format**

  ```c
  int GAUSS__CopyStringArrayToArg( ArgList_t *args, StringArray_t *sa, int argnum );
  ```

  ```c
  ret = GAUSS__CopyStringArrayToArg( args, sa, argnum );
  ```

- **Input**

  - **args**  
    pointer to an argument list structure.
  - **sa**  
    pointer to a string array descriptor.
  - **argnum**  
    number of argument.

- **Output**

  - **ret**  
    success flag, 0 if successful, otherwise:
    - 30  
      Insufficient memory.
    - 494  
      Invalid argument number.

- **Remarks**

  **GAUSS__CopyStringArrayToArg** malloc's a copy of the string array contained in the sa argument and assigns the copy to args.

  To add an argument to the end of an argument list or to an empty argument list, set argnum to 0. To replace an argument, set argnum to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call **GAUSS__InsertArg** and then set argnum to the number of the inserted argument. Arguments are numbered starting with 1.

  The string array copy will be freed when you call **GAUSS__CallProcFreeArgs** or **GAUSS__FreeArgList** later.

  This function allows you to retain sa. If you want to move the string array to the argument list and free the StringArray_t, use **GAUSS__MoveStringArrayToArg** instead.

  Create a StringArray_t with **GAUSS__StringArray** or **GAUSS__StringArrayL**, or use a StringArray_t returned from **GAUSS__GetStringArray**.

- **Example**
This example assumes that `wh` is a pointer to a valid workspace handle and that `stra` is a string array in that workspace. It gets the string array from the workspace and puts it into the `ArgList_t args`. It retains `sa`, which should be freed later with `GAUSS_FreeStringArray`.

**See also**

- `GAUSS_MoveStringArrayToArg`
- `GAUSS_StringArray`
- `GAUSS_StringArrayL`
- `GAUSS_CreateArgList`
- `GAUSS_FreeArgList`
- `GAUSS_InsertArg`
- `GAUSS_CallProc`
- `GAUSS_CallProcFreeArgs`
8. C API: REFERENCE

GAUSS__CopyStringArrayToGlobal

- **Purpose**
  
  Copies a string array contained in a `StringArray_t` into a GAUSS workspace.

- **Format**

  ```
  int GAUSS__CopyStringArrayToGlobal( WorkspaceHandle_t *wh, StringArray_t *sa, char *name );
  ```

  ```
  ret = GAUSS__CopyStringArrayToGlobal( wh, sa, name );
  ```

- **Input**

  - `wh` pointer to a workspace handle.
  - `sa` pointer to string array descriptor.
  - `name` pointer to name of string array.

- **Output**

  - `ret` success flag, 0 if successful, otherwise:
    
    | Code | Description              |
    |------|--------------------------|
    | 26   | Too many symbols.        |
    | 30   | Insufficient memory.     |
    | 91   | Symbol too long.         |
    | 481  | GAUSS assignment failed.  |
    | 495  | Workspace inactive or corrupt. |

- **Remarks**

  `GAUSS__CopyStringArrayToGlobal`'s a copy of the string array contained in `sa` and assigns the copy to `wh`. GAUSS frees it when necessary. This function allows you to retain `sa`.

  If you want to move the matrix to `wh` and free the `StringArray_t`, use `GAUSS__MoveStringArrayToGlobal` instead.

  Create a `StringArray_t` with `GAUSS__StringArray` or `GAUSS__StringArrayL`, and call `GAUSS__CopyStringArrayToGlobal` with a `WorkspaceHandle_t` returned from `GAUSS__CreateWorkspace`.

- **Example**
```c
int ret;
char *stra[2];
char str1[] = "cat";
char str2[] = "bird";

stra[0] = str1;
stra[1] = str2;

sa = GAUSS_StringArray( 2, 1, stra );

if ( ret = GAUSS_CopyStringArrayToGlobal( wh, sa, "st" ) )
{
    char buff[100];

    printf( "CopyStringArrayToGlobal failed: \%s\n",
            GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeStringArray( sa );
    return -1;
}
```

This example copies the string array `stra`, into the `GAUSS` workspace indicated by `wh`. It assumes that `wh` is a pointer to a valid workspace handle. It retains `sa`, which should be freed later with `GAUSS_FreeStringArray`.

**See also**

GAUSS_MoveStringArrayToGlobal, GAUSS_StringArray, GAUSS_StringArrayL, GAUSS_GetStringArray
Purpose
Copies a string contained in a String_t to an ArgList_t.

Format
```c
int GAUSS_CopyStringToArg( ArgList_t *args, String_t *str, int argnum );
```
```c
ret = GAUSS_CopyStringToArg( args, str, argnum );
```

Input
- `args` pointer to an argument list structure.
- `str` pointer to a string descriptor.
- `argnum` argument number.

Output
- `ret` success flag, 0 if successful, otherwise:
  - 30 Insufficient memory.
  - 494 Invalid argument number.

Remarks
The function `GAUSS_CopyStringToArg` `malloc`s a copy of the string contained in the `str` argument and assigns the copy to `args`.

To add an argument to the end of an argument list or to an empty argument list, set `argnum` to 0. To replace an argument, set `argnum` to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call `GAUSS_InsertArg` and then set `argnum` to the number of the inserted argument. Arguments are numbered starting with 1.

The string copy will be freed when you call `GAUSS_CallProcFreeArgs` or `GAUSS_FreeArgList` later.

This function allows you to retain `str`. If you want to move the string to the argument list and free the `String_t`, use `GAUSS_MoveStringToArg` instead.

Call `GAUSS_CopyStringToArg` with a `String_t` returned from one of the following functions:
- `GAUSS_GetString`
- `GAUSS_String`
- `GAUSS_StringAlias`
- `GAUSS_StringAliasL`
- `GAUSS_StringAlias`

Example
The above example copies the string `s` into the `ArgList_t` `args`. It retains `str`, which should be freed later with `GAUSS_FreeString`.

**See also**

`GAUSS_MoveStringToArg`, `GAUSS_String`, `GAUSS_StringL`, `GAUSS_CreateArgList`, `GAUSS_FreeArgList`, `GAUSS_InsertArg`, `GAUSS_CallProc`, `GAUSS_CallProcFreeArgs`
8. C API: REFERENCE

GAUSS_CopyStringToGlobal

- **Purpose**
  
  Copies a string contained in a `String_t` into a GAUSS workspace.

- **Format**
  
  ```c
  int GAUSS_CopyStringToGlobal( WorkspaceHandle_t *wh, String_t *str, char *name );
  ret = GAUSS_CopyStringToGlobal( wh, str, name );
  ```

- **Input**
  
  - `wh`: pointer to a workspace handle.
  - `str`: pointer to string descriptor.
  - `name`: pointer to name of string.

- **Output**
  
  - `ret`: success flag, 0 if successful, otherwise:
    - 26: Too many symbols.
    - 30: Insufficient memory.
    - 91: Symbol too long.
    - 481: GAUSS assignment failed.
    - 495: Workspace inactive or corrupt.

- **Remarks**
  
  - **GAUSS_CopyStringToGlobal** malloc's a copy of the string contained in `str` and assigns the copy to `wh`. GAUSS frees it when necessary. This function allows you to retain `str`.
  
  If you want to move the matrix to `wh` and free the `String_t`, use `GAUSS_MoveStringToGlobal` instead.

  Call `GAUSS_CopyStringToGlobal` with a `String_t` returned from one of the following functions:

  ```c
  GAUSS_getString
  GAUSS_string
  GAUSS_stringAlias
  GAUSS_stringAliasL
  GAUSS_stringL
  ```

  Input a `WorkspaceHandle_t` returned from `GAUSS_CreateWorkspace`.

- **Example**

  ```c
  ```
char e[] = "elephants";
int ret;

str = GAUSS_String( e );

if ( ret = GAUSS_CopyStringToGlobal( wh, str, "se" ) )
{
    char buff[100];

    printf( "CopyStringToArg failed: %s\n",
            GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeString( str );
    return -1;
}

The above example copies the string \texttt{e} into the \texttt{GAUSS} workspace indicated by \texttt{wh}. It assumes that \texttt{wh} is a pointer to a valid workspace handle. It retains \texttt{str}, which should be freed later with \texttt{GAUSS\_FreeString}.

- See also

\texttt{GAUSS\_MoveStringToGlobal}, \texttt{GAUSS\_String}, \texttt{GAUSS\_StringAlias}, \texttt{GAUSS\_GetString}
8. C API: REFERENCE

- **Purpose**
  Creates an empty argument list.

- **Format**
  ```c
  ArgList_t *GAUSS_CreateArgList( void );
  
  args = GAUSS_CreateArgList();
  ```

- **Output**
  ```c
  args   pointer to an argument list structure.
  ```

- **Remarks**
  **GAUSS_CreateArgList** creates an empty argument list structure. Add or replace arguments in it with the following commands:
  - **GAUSS_CopyMatrixToArg**
  - **GAUSS_CopyStringArrayToArg**
  - **GAUSS_CopyStringToArg**
  - **GAUSS_MoveMatrixToArg**
  - **GAUSS_MoveStringArrayToArg**
  - **GAUSS_MoveStringToArg**

  Use **GAUSS_InsertArg** to insert an argument into it.

  To copy or move an argument from one argument list structure to another, use **GAUSS_CopyArgToArg** or **GAUSS_MoveArgToArg**.

  Creating an **ArgList_t** structure allows you to use **GAUSS_CallProc** or **GAUSS_CallProcFreeArgs** to call a procedure without referencing any global variables.

  If **args** is NULL, there was insufficient memory to malloc space for the **ArgList_t**.

- **Example**
  ```c
  ArgList_t *args;

  if ( ( args = GAUSS_CreateArgList() ) == NULL )
  {
      char buff[100];

      printf( "CreateArgList failed: %s\n",
              GAUSS_ErrorText( buff, GAUSS_GetError() ) );
      return -1;
  }
  ```

  77
This example creates the argument list structure, `args`. Once arguments have been added to it, `args` may be used as an input for `GAUSS_CallProc` or `GAUSS_CallProcFreeArgs`.

- **See also**

`GAUSS_FreeArgList`, `GAUSS_InsertArg`, `GAUSS_DeleteArg`, `GAUSS_CallProc`, `GAUSS_CallProcFreeArgs`
8. C API: REFERENCE

**GAUSS_CreateProgram**

- **Purpose**
  Creates an empty program handle.

- **Format**
  
  ```c
  ProgramHandle_t *GAUSS_CreateProgram( WorkspaceHandle_t *wh, int readonlyE );
  
  ph = GAUSS_CreateProgram( wh, readonlyE );
  ```

- **Input**
  - `wh` pointer to a workspace handle.
  - `readonlyE` 1 or 0, if 1, the program cannot assign to global symbols.

- **Output**
  - `ph` pointer to a program handle.

- **Remarks**
  **GAUSS_CreateProgram** allows you to create an empty program handle that is associated with the workspace indicated by `wh`. This program handle pointer may then be passed into **GAUSS_CallProc** or **GAUSS_CallProcFreeArgs**.

  If **GAUSS_CreateProgram** fails, `ph` will be NULL. Use **GAUSS_GetError** to get the number of the error. **GAUSS_CreateProgram** may fail with either of the following errors:

  - 30 Insufficient memory.
  - 495 Workspace inactive or corrupt.

- **Example**
  ```c
  ProgramHandle_t *ph;
  int readonlyE = 1;

  if ( ( ph = GAUSS_createProgram( wh, readonlyE ) ) == NULL )
  {
      char buff[100];
      printf( "CreateProgram failed: %s\n",
              GAUSS_ErrorText( buff, GAUSS_GetError() ) );
      return -1;
  }
  ```

  The above example creates the program handle `ph`, which can be used in **GAUSS_CallProc** or **GAUSS_CallProcFreeArgs**. It assumes that `wh` is a pointer to a valid workspace handle.

- **See also**
  - **GAUSS_FreeProgram**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**
GAUSS__CreateWorkspace

- **Purpose**
  Initializes a workspace.

- **Format**
  
  ```c
  WorkspaceHandle_t *GAUSS__CreateWorkspace( char *name );
  ```

  ```c
  wh = GAUSS__CreateWorkspace( name );
  ```

- **Input**
  
  `name` pointer to name of workspace.

- **Output**
  
  `wh` pointer to a workspace handle.

- **Remarks**
  The workspace contains all of the global symbols. You can create as many workspaces as you want. Each workspace is isolated from all other workspaces.

  If `GAUSS__CreateWorkspace` fails, `wh` will be NULL. Use `GAUSS__GetErr` to get the number of the error. `GAUSS__CreateWorkspace` may fail with any of the following errors:

  - **28** Can’t open configuration file.
  - **29** Missing left parenthesis.
  - **497** Missing right parenthesis.
  - **498** Environment variable not found.
  - **499** Recursive definition of `GAUSSDIR`.

- **Example**

  ```c
  WorkspaceHandle_t *wh;
  
  if ( ( wh = GAUSS_CreateWorkspace( "wksp1" ) ) == NULL )
  {
    char buff[100];
    
    printf( "CreateWorkspace failed: %s\n",
      GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
  }
  ```

- **See also**

  `GAUSS__SaveWorkspace`, `GAUSS__LoadWorkspace`, `GAUSS__FreeWorkspace`, `GAUSS__GetError`
8. C API: Reference

**Purpose**

Deletes an argument from an ArgList_t.

**Format**

```c
int GAUSS_DeleteArg( ArgList_t *args, int argnum );
```

```c
ret = GAUSS_DeleteArg( args, argnum );
```

**Input**

- `args` pointer to an argument list descriptor.
- `argnum` argument number.

**Output**

- `ret` 0 if successful, otherwise 494 if the argument is out of range.

**Remarks**

Use `GAUSS_DeleteArg` to delete an argument from an ArgList_t so that you can reuse the ArgList_t for a different procedure call. To simply replace an argument in an ArgList_t, use one of the following functions:

- `GAUSS_CopyMatrixToArg`
- `GAUSS_CopyStringArrayToArg`
- `GAUSS_CopyStringToArg`
- `GAUSS_MoveMatrixToArg`
- `GAUSS_MoveStringArrayToArg`
- `GAUSS_MoveStringToArg`

**Example**

```c
ProgramHandle_t *ph;
ArgList_t *args;

if ( ( ph = GAUSS_CompileExpression( wh, 
    "rndKMi(200,4,31);",
    1,
    1
    ) ) == NULL )
{
```

```c
```
char buff[100];

printf( "Compile failed: %s\n",
    GAUSS_ErrorText( buff, GAUSS_GetError() ) );
return -1;
}

if ( ( args = GAUSS_ExecuteExpression( ph ) ) == NULL )
{
    char buff[100];

    printf( "Execute failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( GAUSS_DeleteArg( args, 2 ) )
{
    char buff[100];

    printf( "DeleteArg failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgs( args );
    return -1;
}

The above example assumes that wh is a pointer to a valid workspace handle. It executes an expression, which gives its returns in the ArgList_t, args. The example deletes the second argument from args so that the first argument may be used as the input for a later procedure call.

See also

GAUSS_CopyArgToArg, GAUSS_CreateArgList, GAUSS_FreeArgList, GAUSS_InsertArg, GAUSS_CallProc, GAUSS_CallProcFreeArgs
8. C API: REFERENCE

GAUSS_ErrorText

- **Purpose**
  Returns the error message that corresponds to a given error number.

- **Format**
  ```c
  char *GAUSS_ErrorText( char *buff, int errnum );
  
  cp = GAUSS_ErrorText( buff, errnum );
  ```

- **Input**
  - `buff` pointer to a character buffer.
  - `errnum` error number.

- **Output**
  - `cp` pointer to the character buffer containing the error message.

- **Remarks**
  `GAUSS_ErrorText` fills in the character buffer `buff` with the error message corresponding to `errnum`. It returns a pointer to that character buffer. This command allows you to get the error messages that correspond to error numbers returned from failed function calls or from `GAUSS_GetError`.

- **Example**
  ```c
  Matrix_t *mat;

  if ( ( mat = GAUSS_GetMatrix( wh, "a" ) ) == NULL )
  {
    char buff[100];

    printf( "GAUSS_GetMatrix failed: %s\n", 
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
  }
  ```

  This example prints the error message if `GAUSS_GetMatrix` fails. It assumes that `wh` is a pointer to a valid workspace handle and that `a` is already resident in `wh`.

- **See also**
  `GAUSS_GetError`
**GAUSS_Execute**

---

**Purpose**

Executes a program handle.

**Format**

```c
int GAUSS_Execute( ProgramHandle_t *ph );
```

```c
ret = GAUSS_Execute( ph );
```

**Input**

- `ph`: pointer to a program handle.

**Output**

- `ret`: success code, 0 if successful, otherwise:
  - 493: Program execute failed.
  - 495: Workspace inactive or corrupt.
  - 496: Program inactive or corrupt.

**Remarks**

`GAUSS_Execute` is called with a program handle pointer that was returned from one of the following commands:

- `GAUSS_CompileFile`
- `GAUSS_CompileString`
- `GAUSS_CompileStringAsFile`
- `GAUSS_LoadCompiledBuffer`
- `GAUSS_LoadCompiledFile`

**Example**

```c
ProgramHandle_t *ph;
int ret;

if ( ( ph = GAUSS_CompileFile( wh, "examples/ols.e", 0, 0 ) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}
```

---

84
if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];
    printf( "Execute failed: %s\n",
            GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

The example code above runs the GAUSS example file ols.e. It assumes that wh is a pointer to a valid workspace handle.

- **See also**

  GAUSS_CompileFile, GAUSS_CompileString, GAUSS_CompileStringAsFile,
  GAUSS_LoadCompiledBuffer, GAUSS_LoadCompiledFile
GAUSS_ExecuteExpression

■ Purpose

Executes an expression compiled into a program handle.

■ Format

```
ArgList_t *GAUSS_ExecuteExpression( ProgramHandle_t *ph );
```

```
rets = GAUSS_ExecuteExpression( ph );
```

■ Input

```
ph      pointer to a program handle.
```

■ Output

```
rets      pointer to argument list descriptor containing the returns of the expression.
```

■ Remarks

GAUSS_ExecuteExpression is called with a program handle pointer that was returned from GAUSS_CompileExpression.

GAUSS_ExecuteExpression creates an ArgList_t structure in which it puts the returns of the expression. Use the following functions to move the returns of an expression from an ArgList_t into descriptors for each respective data type:

```
GAUSS_CopyArgToMatrix
GAUSS_CopyArgToString
GAUSS_CopyArgToStringArray
GAUSS_MoveArgToMatrix
GAUSS_MoveArgToString
GAUSS_MoveArgToStringArray
```

Use GAUSS_GetArgType to get the type of an argument in an ArgList_t.

It is your responsibility to free the ArgList_t returned from GAUSS_CompileExpression. It may be freed with GAUSS_FreeArgList.

If GAUSS_ExecuteExpression fails, rets will be NULL. Use GAUSS_GetError to get the number of the error. GAUSS_ExecuteExpression may fail with any of the following errors:

```
  30  Insufficient memory.
  493 Program execute failed.
  495 Workspace inactive or corrupt.
  496 Program inactive or corrupt.
```
Example

```c
ProgramHandle_t *ph;
ArgList_t *ret;
Matrix_t *mat;

if ( ( ph = GAUSS_CompileExpression( wh, "inv( x ) * x", 1, 1 ) ) == NULL )
{
    char buff[100];
    printf( "Compile failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
{
    char buff[100];
    printf( "Execute failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( ( mat = GAUSS_MoveArgToMatrix( ret, 1 ) ) == NULL )
{
    char buff[100];
    printf( "MoveArgToMatrix: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( ret );
    return -1;
}
```

The example code above assumes that \( x \) is already resident in the workspace `wh`. `GAUSS_ExecuteExpression` creates the `ArgList_t`, `ret`, which contains the return from the executed expression.

See also

- `GAUSS_Execute`
- `GAUSS_CompileExpression`
- `GAUSS_GetError`
- `GAUSS_FreeArgList`
**GAUSS__FreeArgList**

- **Purpose**
  Frees an argument list.

- **Format**
  
  ```c
  void GAUSS__FreeArgList( ArgList_t *args );
  GAUSS__FreeArgList( args );
  ```

- **Input**
  
  `args`  pointer to an argument list structure.

- **Remarks**
  `GAUSS__FreeArgList` frees an `ArgList_t` structure and all of the arguments it contains.

- **Example**
  
  ```c
  ProgramHandle_t *ph;
  ArgList_t *ret;
  Matrix_t *mat;

  ph = GAUSS_CompileExpression( wh, "sumc(seqm(.2,1,50))", 1, 1 );

  if ( ph == NULL )
  {
      char buff[100];

      printf( "Compile failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
      return -1;
  }

  if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
  {
      char buff[100];

      printf( "Execute failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
      GAUSS_FreeProgram( ph );
      return -1;
  }
  ```

88
if ( ( mat = GAUSS_MoveArgToMatrix( ret, 1 ) ) == NULL )
{
    char buff[100];

    printf( "MoveArgToMatrix failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

GAUSS_FreeArgList( ret );

The example above assumes that `wh` is a pointer to a valid workspace handle.

- **See also**

  GAUSS__CreateArgList, GAUSS__CallProc, GAUSS__CallProcFreeArgs
Purpose

Frees a matrix descriptor and the data it contains.

Format

```c
void GAUSS_FreeMatrix( Matrix_t *mat );
```

```c
GAUSS_FreeMatrix( mat );
```

Input

```c
mat      pointer to a matrix descriptor.
```

Remarks

`GAUSS_FreeMatrix` frees a matrix descriptor and the matrix it points to.

Example

```c
Matrix_t *mat;
ArgList_t *args;
double x[4][2] = { {3,-4}, {6,9}, {-5,0}, {-1,-8} };

args = GAUSS_CreateArgList();

if ( ( mat = GAUSS_Matrix( 4, 2, &x[0][0] ) ) == NULL )
{
    char buff[100];
    printf( "Matrix failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( GAUSS_CopyMatrixToArg( args, mat, 0 ) )
{
    char buff[100];
    printf( "CopyMatrixToArg failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeMatrix( mat );
    return -1;
}

GAUSS_FreeMatrix( mat );
```
The above example creates a matrix descriptor, `mat`, and copies it to `args` as its first argument. It then frees `mat`.

- **See also**

  GAUSS__Matrix, GAUSS__MatrixAlias, GAUSS__ComplexMatrix, GAUSS__ComplexMatrixAlias, GAUSS__GetMatrix
**GAUSSFreeProgram**

- **Purpose**
  
  Frees a program handle.

- **Format**

  ```c
  void GAUSS_FreeProgram( ProgramHandle_t *ph );
  GAUSS_FreeProgram( ph );
  ```

- **Input**

  `ph`  
  pointer to a program handle.

- **Remarks**

  The `GAUSS_FreeProgram` frees a program handle that was created from one of the following commands:

  - `GAUSS.CompileExpression`
  - `GAUSS.CompileFile`
  - `GAUSS.CompileString`
  - `GAUSS.CompileStringAsFile`
  - `GAUSS.CreateProgram`
  - `GAUSS.LoadCompiledBuffer`
  - `GAUSS.LoadCompiledFile`

- **Example**

  ```c
  ProgramHandle_t *ph;
  int ret;

  if ( ( ph = GAUSS.CompileFile( wh, "examples/ols.e", 0, 0 ) ) == NULL )
  {
    char buff[100];
    printf( "Compile failed: %s\n",
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
  }

  if ( ret = GAUSS_Execute( ph ) )
  {
    char buff[100];
  }
  ```
printf( "Execute failed: %s
",
    GAUSS_ErrorText( buff, ret ) );
GAUSS_FreeProgram( ph );
return -1;
}

GAUSS_FreeProgram( ph );

The example code above runs the GAUSS example file ols.e. It assumes that wh is a valid workspace handle.

See also

GAUSS_CreateProgram, GAUSS_CompileExpression, GAUSS_CompileFile,
GAUSS_CompileString, GAUSS_CompileStringAsFile,
GAUSS_LoadCompiledBuffer, GAUSS_LoadCompiledFile
Purpose
Frees a string descriptor and the data it contains.

Format

void GAUSS_FreeString( String_t *str);
GAUSS_FreeString( str);

Input
str pointer to a string descriptor.

Remarks
GAUSS_FreeString frees a string descriptor and the string it points to.

Example

String_t *str;
char s[] = "tmp.out";

if ( ( str = GAUSS_String( s ) ) == NULL )
{
    char buff[100];
    printf( "String failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( GAUSS_CopyStringToGlobal( wh, str, "fname" ) )
{
    char buff[100];
    printf( "CopyStringToGlobal failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeString( str );
    return -1;
}

GAUSS_FreeString( str );

This example assumes that wh is a pointer to a valid workspace. It frees str after copying the string it contains to wh.

See also
GAUSS_String, GAUSS_StringAlias, GAUSS_StringL, GAUSS_StringAliasL, GAUSS_GetString
8. C API: REFERENCE

- **Purpose**
  Frees a string array descriptor and the data it contains.

- **Format**
  
  ```c
  void GAUSS_FreeStringArray( StringArray_t *sa );
  GAUSS_FreeStringArray( sa );
  ```

- **Input**
  
  `sa` pointer to a string array descriptor.

- **Remarks**
  
  `GAUSS_FreeStringArray` frees a string array descriptor and the string array it points to.

- **Example**
  
  ```c
  StringArray_t *sa;

  if ( ( sa = GAUSS_GetStringArray( wh, "names" ) ) == NULL )
  {
      char buff[100];

      printf( "GetStringArray failed: %s\n",
              GAUSS_ErrorText( buff, GAUSS_GetError() ) );
      return -1;
  }

  if ( sa->rows != 20 || sa->cols != 1 )
  {
      printf( "String array corrupt\n" );
      GAUSS_FreeStringArray( sa );
      return -1;
  }

  GAUSS_FreeStringArray( sa );
  ```

  This example assumes that `wh` is a pointer to a valid workspace and that the 20*1 string array `names` is already resident in that workspace. It gets `names` from `wh`, and puts it into a string array descriptor, `sa`. It checks the rows and columns of the string array and then frees `sa`.

- **See also**
  
  `GAUSS_StringArray`, `GAUSS_StringArrayL`, `GAUSS_GetStringArray`
GAUSS_FreeWorkspace

- **Purpose**
  Frees a workspace handle.

- **Format**
  
  ```c
  void GAUSS_FreeWorkspace( WorkspaceHandle_t *wh );
  ```

  ```c
  GAUSS_FreeWorkspace( wh );
  ```

- **Input**
  
  ```c
  wh   pointer to a workspace handle.
  ```

- **Remarks**
  
  GAUSS_FreeWorkspace frees a workspace handle that was created with GAUSS_CreateWorkspace.

- **Example**

  ```c
  WorkspaceHandle_t *wh;
  ProgramHandle_t *ph;

  wh = GAUSS_CreateWorkspace( "main" );

  if ( ( ph = GAUSS_CompileFile( wh, "examples/qnewton1.e", 0, 0 ) ) == NULL )
  {
    char buff[100];

    printf( "CompileFile failed: %s\n",
                GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeWorkspace( wh );
    return -1;
  }

  if ( GAUSS_Execute( ph ) )
  {
    char buff[100];

    printf( "Execute failed: %s\n",
                GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeWorkspace( wh );
    GAUSS_FreeProgram( ph );
    return -1;
  }

  GAUSS_FreeProgram( ph );
  GAUSS_FreeWorkspace( wh );
  ```
This example creates the workspace handle, `wh`, and runs the example file `qnewton1.e` in that workspace. At the end, it frees the program handle used to run the file as well as the workspace handle.

- **See also**

  `GAUSS__CreateWorkspace`, `GAUSS__SaveWorkspace`, `GAUSS__LoadWorkspace`
GAUSS__GetArgType

- **Purpose**
  
  Gets the type of a symbol in an ArgList_t.

- **Format**

  ```c
  int GAUSS__GetArgType( ArgList_t *args, int argnum );
  
  typ = GAUSS__GetArgType( args, argnum );
  ```

- **Input**

  - *args* pointer to an argument list descriptor.
  - *argnum* argument number.

- **Output**

  - *typ* type of symbol:
    
    GAUSS__MATRIX  
    GAUSS__STRING  
    GAUSS__STRING__ARRAY

- **Remarks**

  Use GAUSS__GetArgType to find the type of a symbol in an ArgList_t, so you can use the following functions to move the symbols to type-specific structures:

    GAUSS__CopyArgToMatrix  
    GAUSS__CopyArgToString  
    GAUSS__CopyArgToStringArray  
    GAUSS__MoveArgToMatrix  
    GAUSS__MoveArgToString  
    GAUSS__MoveArgToStringArray

  If GAUSS__GetArgType fails, *typ* will be -1. It will fail only if the argument is out of range.

- **Example**

  ```c
  ProgramHandle_t *ph;
  ArgList_t *ret;
  Matrix_t *mat;

  if ( ( ph = GAUSS__CompileExpression( 
```
This example assumes that \texttt{wh} is a pointer to a valid workspace handle. It executes an expression, which places its return in an \texttt{ArgList} \texttt{t}. The example checks to make sure that the return is of type \texttt{GAUSS\_MATRIX} before moving it to a matrix descriptor.
See also

GAUSS_CallProc, GAUSS_CallProcFreeArgs, GAUSS_ExecuteExpression
8. C API: REFERENCE

**Purpose**

Gets a global double from a GAUSS workspace.

**Format**

```c
int GAUSS_GetDouble( WorkspaceHandle_t *wh, double *d, char *name );
```

```c
ret = GAUSS_GetDouble( wh, d, name );
```

**Input**

- `wh` pointer to a workspace handle.
- `d` pointer to be set to double.
- `name` pointer to name of symbol.

**Output**

- `ret` success flag, 0 if successful, otherwise:
  - 41 Argument must be scalar.
  - 71 Type mismatch.
  - 91 Symbol too long.
  - 470 Symbol not found.
  - 495 Workspace inactive or corrupt.

**Remarks**

GAUSS_GetDouble finds a scalar in a GAUSS workspace and assigns the value of it to `d`. This gives you a safe copy of the data that you can work with without affecting the contents of the symbol table.

GAUSS_GetDouble must be called with a `WorkspaceHandle_t` returned from GAUSS_CreateWorkspace.

**Example**

```c
ProgramHandle_t *ph
double d;
int ret;

if ( ( ph = GAUSS_CompileString(
    wh,
    "\{ a, rs \} = rndKMn( 1, 1, 31 );",
```

---

101
The above example assumes that `wh` is a pointer to a valid workspace handle.

- **See also**

  `GAUSS_PutDouble, GAUSS_GetMatrix`
8. C API: REFERENCE

- **Purpose**
  Returns the stored error number.

- **Format**
  ```c
  int GAUSS_GetError( void );
  errnum = GAUSS_GetError();
  ```

- **Output**
  ```c
  errnum  error number.
  ```

- **Remarks**
  The **GAUSS Enterprise Engine** stores the error number of the most recently encountered error in a system variable. If a **GAUSS Enterprise Engine** command fails, it automatically resets this variable with the number of the error. However, the command does not clear the variable if it succeeds.

  Many **GAUSS Enterprise Engine** commands also return a success code. It is set to 0 if the command succeeds or to a specific error number if it fails. Most of the commands that do not return a success code will return a NULL pointer if they fail. Use **GAUSS_GetError** to check the errors from these commands. Since the variable does not get cleared, only call **GAUSS_GetError** if a function fails.

  The system variable is global to the current thread.

  Follow **GAUSS_GetError** with a call to **GAUSS_ErrorText** to get the error message that corresponds to `errnum`.

- **Example**
  ```c
  String_t *str;
  
  if ( ( str = GAUSS_GetString( wh, "s" ) ) == NULL )
  {
    char buff[100];
    
    printf( "GetString failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
  }
  ```

  This example prints the error message if **GAUSS_GetString** fails. It assumes that `wh` is a pointer to a valid workspace handle and that `s` is already resident in `wh`.

- **See also**
  **GAUSS_SetError**, **GAUSS_ErrorText**
GAUSS_GetHome

- **Purpose**
  Gets the current engine home path.

- **Format**
  ```
  char *GAUSS_GetHome( char *buff );
  ```
  ```
  path = GAUSS_GetHome( buff );
  ```

- **Input**
  ```
  buff       pointer to 1024 byte buffer to put path.
  ```

- **Output**
  ```
  path       pointer to buffer.
  ```

- **Remarks**
  **GAUSS_GetHome** fills *buff* with the current home path and returns a pointer to that buffer.

- **Example**
  ```
  char buff[1024];

  printf( "%s\n", GAUSS_GetHome( buff ) );
  ```

- **See also**
  **GAUSS_SetHome**
8. C API: REFERENCE

GAUSS\_GetLogFile

- **Purpose**
  
  Gets the name of the current log file.

- **Format**
  
  ```c
  char *GAUSS\_GetLogFile( char *buff );
  
  logfn = GAUSS\_GetLogFile( buff );
  ```

- **Input**
  
  ```c
  buff    pointer to buffer for log file name to be put in.
  ```

- **Output**
  
  ```c
  logfn    pointer to name of log file.
  ```

- **Remarks**
  
  The **GAUSS** Engine logs certain system level errors in 2 places: a file and an open file pointer. The default file is `/tmp/mteng.###.log` where `###` is the process ID number. The default file pointer is `stderr`.

  **GAUSS\_GetLogFile** fills `buff` with the name of the current log file and returns a pointer to that buffer.

- **Example**
  
  ```c
  char buff[40];
  
  printf( "%s\n", GAUSS\_GetLogFile( buff ) );
  ```

- **See also**
  
  `GAUSS\_SetLogFile`, `GAUSS\_GetLogStream`, `GAUSS\_SetLogStream`
GAUSS__GetLogStream

**Purpose**

Get the current log file pointer.

**Format**

```c
FILE *GAUSS__GetLogStream( void );
```

```
logfp = GAUSS__GetLogStream();
```

**Output**

`logfp` pointer to log file handle.

**Remarks**

The **GAUSS** Engine logs certain system level errors in 2 places: a file and an open file pointer. The default file is `/tmp/mteng.###.log` where `###` is the process ID number. The default file pointer is `stderr`.

**GAUSS__GetLogStream** returns the current log file pointer.

**See also**

**GAUSS__SetLogStream**, **GAUSS__GetLogFile**, **GAUSS__SetLogFile**
8. C API: REFERENCE

- **Purpose**
  Gets a global matrix from a GAUSS workspace.

- **Format**
  ```c
  Matrix_t *GAUSS_GetMatrix( WorkspaceHandle_t *wh, char *name );
  
  mat = GAUSS_GetMatrix( wh, name );
  ```

- **Input**
  - `wh` pointer to a workspace handle.
  - `name` pointer to name of matrix.

- **Output**
  - `mat` pointer to a matrix descriptor.

- **Remarks**
  `GAUSS_GetMatrix` finds a matrix in a GAUSS workspace and `malloc`'s a matrix descriptor, filling it in with the information for the matrix. It makes a copy of the matrix and sets the `mdata` member of the matrix descriptor to point to the copy. This gives you a safe copy of the matrix that you can work with without affecting the contents of the GAUSS symbol table. This copy of the matrix then belongs to you. Free it with `GAUSS_FreeMatrix`.

  If the matrix is complex, its copy will be stored in memory with the entire real part first, followed by the imaginary part.

  If the matrix is empty, the `rows` and `cols` members of the `Matrix_t` will be set to 0, and the `mdata` member will be NULL.

  Call `GAUSS_GetMatrix` with a `WorkspaceHandle_t` pointer returned from `GAUSS_CreateWorkspace`.

  If `GAUSS_GetMatrix` fails, `mat` will be NULL. Use `GAUSS_GetError` to get the number of the error. `GAUSS_GetMatrix` may fail with any of the following errors:

  - `30` Insufficient memory.
  - `71` Type mismatch.
  - `91` Symbol too long.
  - `470` Symbol not found.
  - `495` Workspace inactive or corrupt.
Example

ProgramHandle_t *ph;
Matrix_t *mat;
int ret;

if ( ( ph = GAUSS_CompileString(
    wh,
    "{ a, rs } = rndKMn( 4, 4, 31 ); b = inv( a );",
    0,
    0
  ) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];

    printf( "Execute failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( ( mat = GAUSS_GetMatrix( wh, "b" ) ) == NULL )
{
    char buff[100];

    printf( "GetMatrix failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

The example above assumes that wh is a pointer to a valid workspace handle.

See also

GAUSS_GetMatrixAndClear, GAUSS_GetMatrixInfo,
GAUSS_CopyMatrixToGlobal, GAUSS_MoveMatrixToGlobal, GAUSS_GetDouble
GAUSS__GetMatrixAndClear

- **Purpose**
  Gets a global matrix from a GAUSS workspace and clears the matrix in that workspace.

- **Format**
  ```c
  Matrix_t *GAUSS__GetMatrixAndClear( WorkspaceHandle_t *wh, char *name );
  ```
  ```c
  mat = GAUSS__GetMatrixAndClear( wh, name );
  ```

- **Input**
  - `wh` pointer to a workspace handle.
  - `name` pointer to name of matrix.

- **Output**
  - `mat` pointer to a matrix descriptor.

- **Remarks**
  **GAUSS__GetMatrixAndClear** finds a matrix in a GAUSS workspace and malloc's a matrix descriptor, filling it in with the information for the matrix. It sets the `mdata` member of the `Matrix_t` to point to the matrix and sets the matrix to a scalar 0 in the GAUSS symbol table. This allows you to get large matrices from a GAUSS workspace without using the time and memory space needed to copy the matrix. The matrix then belongs to you. Free it with **GAUSS__FreeMatrix**.

  If the matrix is complex, its copy will be stored in memory with the entire real part first, followed by the imaginary part.

  If the matrix is empty, the `rows` and `cols` members of the `Matrix_t` will be set to 0, and the `mdata` member will be NULL.

  Call **GAUSS__GetMatrixAndClear** with a `WorkspaceHandle_t` pointer returned from **GAUSS__CreateWorkspace**.

  If **GAUSS__GetMatrixAndClear** fails, `mat` will be NULL. Use **GAUSS__GetError** to get the number of the error. **GAUSS__GetMatrixAndClear** may fail with any of the following errors:

  - **30** Insufficient memory.
  - **71** Type mismatch.
  - **91** Symbol too long.
  - **470** Symbol not found.
  - **495** Workspace inactive or corrupt.
GAUSS_\_GetMatrixAndClear

Example

```c
ProgramHandle_t *ph;
Matrix_t *mat;
int ret;

if (( ph = GAUSS_\_CompileString(
wh,
"\{ a, rs \} = \text{rndKMu}( 10000, 1000, 31 );",
0,
0
) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_\_Execute( ph ) )
{
    char buff[100];

    printf( "Execute failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_\_FreeProgram( ph );
    return -1;
}

if (( mat = GAUSS_\_GetMatrixAndClear( wh, "a" ) ) == NULL )
{
    char buff[100];

    printf( "GetMatrixAndClear failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_\_FreeProgram( ph );
    return -1;
}
```

The example above assumes that `wh` is a pointer to a valid workspace handle. It gets a matrix of random numbers, `a`, and resets `a` in `wh` to a scalar 0.

See also

- GAUSS_\_GetMatrix
- GAUSS_\_GetMatrixInfo
- GAUSS_\_CopyMatrixToGlobal
- GAUSS_\_MoveMatrixToGlobal
- GAUSS_\_GetDouble
8. C API: REFERENCE

GAUSS__GetMatrixInfo

- **Purpose**
  
  Gets information for a matrix in a GAUSS workspace.

- **Format**

  ```c
  int GAUSS__GetMatrixInfo( WorkspaceHandle_t *wh, GAUSS_MatrixInfo_t *matinfo, char *name );
  ```

  ```c
  ret = GAUSS__GetMatrixInfo( wh, matinfo, name );
  ```

- **Input**

  - `wh` pointer to a workspace handle.
  - `matinfo` pointer to a matrix info descriptor.
  - `name` pointer to name of matrix.

- **Output**

  - `ret` success flag, 0 if successful, otherwise:
    - 71 Type mismatch.
    - 91 Symbol too long.
    - 470 Symbol not found.
    - 495 Workspace inactive or corrupt.

- **Remarks**

  GAUSS__GetMatrixInfo finds a matrix in a GAUSS workspace and fills in the matrix info descriptor with the information for the matrix. It sets the `maddr` member of the descriptor to point to the matrix. If the matrix is complex, it will be stored in memory with the entire real part first, followed by the imaginary part. Since GAUSS__GetMatrixInfo gives you a pointer to the data of the matrix contained in a GAUSS workspace, any changes you make to the data after getting it will be reflected in the symbol table. The matrix still belongs to GAUSS, and GAUSS will free it when necessary. You should not attempt to free a matrix that you get with GAUSS__GetMatrixInfo.

  Call GAUSS__GetMatrixInfo with a WorkspaceHandle_t pointer returned from GAUSS__CreateWorkspace.

- **Example**
ProgramHandle_t *ph;
GAUSS_MatrixInfo_t matinfo;
int ret;

if ( ( ph = GAUSS_CompileString( 
    wh, 
    "a = reshape( seqm( 2, .4, 25 ), 5, 5 );", 
    0, 
    0 
 ) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n", 
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];

    printf( "Execute failed: %s\n", 
            GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( ret = GAUSS_GetMatrixInfo( wh, &matinfo, "a" ) )
{
    char buff[100];

    printf( "GetMatrixInfo failed: %s\n", 
            GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

The example above assumes that \texttt{wh} is a pointer to a valid workspace handle.

\textbf{See also}

\texttt{GAUSS\_GetMatrix}, \texttt{GAUSS\_GetMatrixAndClear}, \texttt{GAUSS\_CopyMatrixToGlobal}, 
\texttt{GAUSS\_MoveMatrixToGlobal}, \texttt{GAUSS\_AssignFreeableMatrix}, \texttt{GAUSS\_GetDouble}
Purpose

Gets a global string from a GAUSS workspace.

Format

```c
String_t *GAUSS__GetString( WorkspaceHandle_t *wh, char *name );
```

```c
str = GAUSS__GetString( wh, name );
```

Input

- `wh` pointer to a workspace handle.
- `name` pointer to name of string.

Output

- `str` pointer to a string descriptor.

Remarks

GAUSS__GetString finds a string in a GAUSS workspace and malloc’s a string descriptor, filling it in with the information for the string. It makes a copy of the string’s data and sets the `stdata` member of the string descriptor to point to the copy. This gives you a safe copy of the data that you can work with without affecting the contents of the GAUSS symbol table. This copy of the data then belongs to you. Free it with GAUSS__FreeString.

Call GAUSS__GetString with a `WorkspaceHandle_t` pointer returned from GAUSS__CreateWorkspace.

If GAUSS__GetString fails, `str` will be NULL. Use GAUSS__GetError to get the number of the error. GAUSS__GetString may fail with any of the following errors:

- 30 Insufficient memory.
- 71 Type mismatch.
- 91 Symbol too long.
- 470 Symbol not found.
- 495 Workspace inactive or corrupt.

Example
The example above assumes that `wh` is a pointer to a valid workspace handle.

### See also

- `GAUSS_FreeString`
- `GAUSS_GetError`
- `GAUSS_CopyStringToGlobal`
- `GAUSS_MoveStringToGlobal`
- `GAUSS_GetStringArray`
8. C API: REFERENCE

GAUSSGetStringArray

- **Purpose**

  Gets a global string array from a GAUSS workspace.

- **Format**

  ```c
  StringArray_t *GAUSSGetStringArray( WorkspaceHandle_t *wh, char *name );
  ```

  ```c
  sa = GAUSSGetStringArray( wh, name );
  ```

- **Input**

  - *wh* pointer to a workspace handle.
  - *name* pointer to name of string array.

- **Output**

  - *sa* pointer to a string array descriptor.

- **Remarks**

  GAUSSGetStringArray finds a string array in a GAUSS workspace and malloc's a string array descriptor, filling it in with the information for the string array. It fills the `table` member of the descriptor with the address of an array of `rows*cols` string element descriptors. GAUSSGetStringArray makes copies of each string in the array and places the copies directly after the string element descriptors in memory. This gives you a safe copy of the string array that you can work with without affecting the contents of the GAUSS symbol table. This copy of the string array belongs to you. Free it with GAUSSFreeStringArray.

  Call GAUSSGetStringArray with a WorkspaceHandle_t pointer returned from GAUSSCreateWorkspace.

  If GAUSSGetStringArray fails, *sa* will be NULL. Use GAUSSGetError to get the number of the error. GAUSSGetStringArray may fail with any of the following errors:

  - 30 Insufficient memory.
  - 71 Type mismatch.
  - 91 Symbol too long.
  - 470 Symbol not found.
  - 495 Workspace inactive or corrupt.

- **Example**

  ```c
  ```

  115
```c
ProgramHandle_t *ph;
StringArray_t *stra;
int ret;

if ( ph = GAUSS_CompileString(
    wh,
    "string sa = { "cats" "dogs", "fish" "birds" };", 0, 0)
) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_Execute( ph ) )
{
    char buff[100];

    printf( "Execute failed: %s\n", GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( stra = GAUSS_GetStringArray( wh, "sa" ) )
{
    char buff[100];

    printf( "GetStringArray failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}
```

The example above assumes that `wh` is a pointer to a valid workspace handle.

- **See also**
  GAUSS_FreeStringArray, GAUSS_GetError, GAUSS_CopyStringArrayToGlobal, GAUSS_MoveStringArrayToGlobal, GAUSS_GetString
8. C API: REFERENCE

**GAUSS__GetSymbolType**

- **Purpose**
  
  Gets the type of a symbol in a GAUSS workspace.

- **Format**
  
  ```c
  int GAUSS__GetSymbolType( WorkspaceHandle_t *wh, char *name );
  ```

  ```c
  typ = GAUSS__GetSymbolType( wh, name );
  ```

- **Input**

  - `wh` pointer to a workspace handle.
  - `name` pointer to name of symbol.

- **Output**

  - `typ` type of symbol:
    
    - `GAUSS__MATRIX`
    - `GAUSS__STRING`
    - `GAUSS__STRING_ARRAY`
    - `GAUSS__PROC`
    - `GAUSS__OTHER`

- **Remarks**

  `GAUSS__GetSymbolType` returns the type of a symbol in a GAUSS workspace or 0 if it cannot find the symbol.

  Call `GAUSS__GetSymbolType` with a `WorkspaceHandle_t` returned from `GAUSS__CreateWorkspace`.

  If `GAUSS__GetSymbolType` fails, `typ` will be -1. Use `GAUSS__GetError` to get the number of the error. `GAUSS__GetSymbolType` may fail with either of the following errors:

    | Error Code | Description               |
    |------------|---------------------------|
    | 91         | Symbol too long.          |
    | 495        | Workspace inactive or corrupt. |

- **Example**

  ```c
  ProgramHandle_t *ph;
  Matrix_t *mat;
  int ret, typ;
  ```
The example above sets a character matrix, $b$, in a GAUSS workspace. It gets the type of $b$ to ensure that it is a matrix, and gets the matrix from the workspace. The example assumes that $wh$ is a pointer to a valid workspace handle.

**See also**

GAUSS__GetMatrix, GAUSS__GetString, GAUSS__GetStringArray
8. C API: REFERENCE

GAUSS__GetWorkspaceName

- **Purpose**

  Gets the name of a GAUSS workspace.

- **Format**

  ```c
  void GAUSS__GetWorkspaceName( WorkspaceHandle_t *wh, char *buff );
  GAUSS__GetWorkspaceName( wh, buff );
  ```

- **Input**

  - `wh` pointer to a workspace handle.
  - `buff` pointer to character buffer at least 64 bytes in length.

- **Remarks**

  `GAUSS__GetWorkspaceName` fills in the character buffer, `buff`, with the name of a GAUSS workspace indicated by a `WorkspaceHandle_t`.

- **See also**

  `GAUSS__CreateWorkspace`, `GAUSS__SetWorkspaceName`
GAUSS_HookFlushProgramOutput

**Purpose**

Specifies the function *GAUSS* calls to flush buffered output.

**Format**

```c
void GAUSS_HookFlushProgramOutput( void ( *flush_output_fn )( void ) );
```

**Input**

`flush_output_fn` pointer to function.

**Remarks**

GAUSS_HookFlushProgramOutput specifies the function called to flush buffered output by the following *GAUSS* functions: `con`, `cons`, `keyw`, `lshow`, `print`, `printfm`, `show`, and `sleep`.

Many *GAUSS* programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The GAUSS_Hook* commands are used to specify those functions. See section 3.1.3.

The callbacks are thread specific. This function must be called by every thread that will use the callback function.

**See also**

GAUSS_HookProgramOutput
8. C API: REFERENCE

GAUSS_HookGetCursorPosition

- **Purpose**
  Specifies the function GAUSS calls to get the position of the cursor.

- **Format**
  
  ```c
  void GAUSS_HookGetCursorPosition( int ( *get_cursor_fn )( void ) );
  GAUSS_HookGetCursorPosition( get_cursor_fn );
  ```

- **Input**
  `get_cursor_fn` pointer to function.

- **Remarks**
  
  GAUSS_HookGetCursorPosition specifies the function called by the GAUSS csrcol and csrln commands to get the position of the cursor. Your get cursor position function must take nothing and return an int, the position of the cursor.

  Many GAUSS programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The GAUSS_Hook* commands are used to specify those functions. See section 3.1.3.

  The callbacks are thread specific. This function must be called by every thread that will use the callback function.

- **See also**
  GAUSS_HookProgramOutput
GAUSS_HookProgramErrorOutput

- **Purpose**
  Specifies the function GAUSS calls to display error messages.

- **Format**
  ```c
  void GAUSS_HookProgramErrorOutput( void (*dpy_err_str_fn)( char * ));
  GAUSS_HookProgramErrorOutput( dpy_err_str_fn );
  ```

- **Input**
  `dpy_err_str_fn` pointer to function.

- **Remarks**
  GAUSS_HookProgramErrorOutput specifies the function that GAUSS calls to display its error messages. Your display error string function must take a char * (a pointer to the error string to print), and return nothing.

  Many GAUSS programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The GAUSS_Hook* commands are used to specify those functions. See section 3.1.3.

  The callbacks are thread specific. This function must be called by every thread that will use the callback function.

- **Example**
  ```c
  void program_error( char *str )
  {
    FILE *fp;
    fp = fopen("test.log", "a");
    fputs(str, fp);
    fclose(fp);
  }
  ```

  This function will write the GAUSS program error output to a file called test.log. It should be hooked at the beginning of a thread as follows:

  ```c
  GAUSS_HookProgramErrorOutput( program_error );
  ```

- **See also**
  GAUSS_HookProgramOutput
8. C API: Reference

GAUSS_HookProgramInputChar

- **Purpose**
  Specifies the function GAUSS calls to get a character of input.

- **Format**
  
  ```c
  void GAUSS_HookProgramInputChar( int (*input_char_function)( void ));
  ```

- **Input**
  ```c
  input_char_function  pointer to function.
  ```

- **Remarks**
  GAUSS_HookProgramInputChar specifies the function called by the GAUSS key command to get a character of input if available. Your input character function must take no arguments and return an `int`, the value of the character of input.

  Many GAUSS programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The GAUSS_Hook* commands are used to specify those functions. See section 3.1.3.

  The callbacks are thread specific. This function must be called by every thread that will use the callback function.

- **See also**
  GAUSS_HookProgramInputCharBlocking, GAUSS_HookProgramInputCheck, GAUSS_HookProgramInputString
**Purpose**

Specifies the function `GAUSS` calls to wait for a character of input.

**Format**

```c
void GAUSS_HookProgramInputCharBlocking( int (*inp_char_blking_fn)( void ) );
GAUSS_HookProgramInputCharBlocking( inp_char_blking_fn );
```

**Input**

`inp_char_blking_fn` function pointer.

**Remarks**

`GAUSS_HookProgramInputCharBlocking` specifies the function called by the `GAUSS keyw` and `show` commands to get (blocking) character input from your application. Your input character blocking function must take no arguments and return an `int`, the value of the character of input.

Many `GAUSS` programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions that it can call for both normal and critical I/O. The `GAUSS_Hook*` commands are used to specify those functions. See section 3.1.3.

The callbacks are thread specific. This function must be called by every thread that will use the callback function.

**See also**

`GAUSS_HookProgramInputChar`, `GAUSS_HookProgramInputCheck`, `GAUSS_HookProgramInputString`
Purpose

Specifies the function GAUSS calls to check for pending input.

Format

```c
void GAUSS_HookProgramInputCheck( int (*input_check_fn)( void ) );
```

```c
GAUSS_HookProgramInputCheck( input_check_fn );
```

Input

`input_check_fn` pointer to function.

Remarks

GAUSS_HookProgramInputCheck specifies the function called by the GAUSS keyav command calls to check if input is pending. Your input check function must take no arguments and return an `int`, 1 if input is available, 0 otherwise.

Many GAUSS programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The GAUSS_Hook* commands are used to specify those functions. See section 3.1.3.

The callbacks are thread specific. This function must be called by every thread that will use the callback function.

See also

GAUSS_HookProgramInputChar, GAUSS_HookProgramInputCharBlocking, GAUSS_HookProgramInputString
GAUSS_HookProgramInputString

- **Purpose**
  Specifies the function GAUSS calls to wait for a string of input.

- **Format**
  ```c
  void GAUSS_HookProgramInputString( int *input_string_fn)( char *, int );
  GAUSS_HookProgramInputString( input_string_fn );
  ```

- **Input**
  ```c
  input_string_fn  pointer to function.
  ```

- **Remarks**
  GAUSS_HookProgramInputString specifies the function called by the GAUSS con and cons commands to get (blocking) string input from your application. Your input string function must take a character pointer (the buffer in which to place the string) and an integer specifying the length of the buffer. Your function must return an int which gives the length of the string, not including the null terminating byte.

  Many GAUSS programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The GAUSS_Hook* commands are used to specify those functions. See section 3.1.3.

  The callbacks are thread specific. This function must be called by every thread that will use the callback function.

- **See also**
  GAUSS_HookProgramInputChar, GAUSS_HookProgramInputCharBlocking, GAUSS_HookProgramInputCheck
Purpose

Specifies the function GAUSS calls to display program output.

Format

```c
void GAUSS_HookProgramOutput( void ( *display_string_fn )( char * ) );
GAUSS_HookProgramOutput( display_string_fn );
```

Input

`display_string_fn` pointer to function.

Remarks

`GAUSS_HookProgramOutput` specifies the function GAUSS calls to display its program output. Your display string function must take a `char *` (a pointer to the string to print) and return nothing.

Many GAUSS programs perform I/O, but the engine has no connections of its own to the outside world. Instead, it relies on you to supply it with functions it can call for both normal and critical I/O. The `GAUSS_Hook*` commands are used to specify those functions. See section 3.1.3.

The callbacks are thread specific. This function must be called by every thread that will use the callback function.

Example

```c
void program_output( char *str )
{
    FILE *fp;

    fp = fopen("progout.log", "a");
    fputs(str, fp);
    fclose(fp);
}
```

This function will write the normal GAUSS program output to a file called `progout.log`. It should be hooked at the beginning of a thread as follows:

```c
GAUSS_HookProgramOutput( program_output );
```

See also

`GAUSS_HookProgramErrorOutput`
GAUSS.Initialize

- **Purpose**
  
  Initializes the engine.

- **Format**
  
  ```c
  int GAUSS.Initialize( void );
  ```

  ```c
  ret = GAUSS.Initialize();
  ```

- **Output**

  - `ret` success flag, 0 if successful, otherwise:
    - 85 Invalid file type.
    - 482 **GAUSS Engine** already initialized.
    - 483 Cannot determine home directory.
    - 487 License expired.
    - 488 Cannot stat file.
    - 489 File has no execute permissions.
    - 490 License manager initialization error.
    - 491 License manager error.
    - 492 Licensing failure.

- **Remarks**

  **GAUSS.Initialize** reads the configuration file. You need to call it once at the beginning of your application. If **GAUSS.Initialize** fails, you should terminate your application.

  Call **GAUSS.SetHome** or **GAUSS.SetHomeVar** before calling **GAUSS.Initialize**.

- **See also**

  **GAUSS.SetHome**, **GAUSS.SetHomeVar**, **GAUSS.Shutdown**
8. C API: REFERENCE

- **Purpose**
  Inserts an empty argument into an `ArgList_t`.

- **Format**
  ```c
  int GAUSS_InsertArg( ArgList_t *args, int argnum );
  newargnum = GAUSS_InsertArg( args, argnum );
  ```

- **Input**
  - `args` pointer to an argument list structure.
  - `argnum` number of argument.

- **Output**
  - `newargnum` number of inserted argument.

- **Remarks**
  The `GAUSS_InsertArg` function inserts an empty argument descriptor into an `ArgList_t` before the `argnum` argument. Fill in the argument descriptor with the following commands:
    - `GAUSS_CopyMatrixToArg`
    - `GAUSS_CopyStringArrayToArg`
    - `GAUSS_CopyStringToArg`
    - `GAUSS_MoveMatrixToArg`
    - `GAUSS_MoveStringArrayToArg`
    - `GAUSS_MoveStringToArg`

  If `GAUSS_InsertArg` fails, `newargnum` will be -1. Use `GAUSS_GetError` to get the number of the error. The `GAUSS_InsertArg` function may fail with either of the following errors:
    - 30  Insufficient memory.
    - 494 Invalid argument number.

- **See also**
  - `GAUSS_CreateArgList`
  - `GAUSS_FreeArgList`
  - `GAUSS_CallProc`
  - `GAUSS_CallProcFreeArgs`
  - `GAUSS_DeleteArg`
  - `GAUSS_GetError`
**Purpose**

Loads a compiled program stored in a character buffer.

**Format**

```c
ProgramHandle_t *GAUSS__LoadCompiledBuffer( WorkspaceHandle_t *wh, char *buff );
```

```c
ph = GAUSS__LoadCompiledBuffer( wh, buff );
```

**Input**

- `wh` pointer to a workspace handle.
- `buff` pointer to a buffer containing the program.

**Output**

- `ph` pointer to a program handle.

**Remarks**

The buffer can be created with the `mkcb` utility and then compiled into your application using a C compiler. Execute `mkcb` with no arguments to get the syntax. `mkcb` converts a `.gcg` file to a C character string definition that can be compiled into your application.

`GAUSS__LoadCompiledBuffer` returns a program handle pointer you can use in `GAUSS__Execute` to execute the program.

Call `GAUSS__LoadCompiledBuffer` with a `WorkspaceHandle_t` pointer returned from `GAUSS__CreateWorkspace`.

If `GAUSS__LoadCompiledBuffer` fails, `ph` will be NULL. Use `GAUSS__GetError` to get the number of the error. `GAUSS__LoadCompiledBuffer` may fail with either of the following errors:

- 30  Insufficient memory.
- 495 Workspace inactive or corrupt.

**See also**

`GAUSS__Execute`, `GAUSS__LoadCompiledFile`, `GAUSS__CompileFile`, `GAUSS__CompileStringAsFile`, `GAUSS__GetError`
GAUSS_LoadCompiledFile

Purpose

Loads a compiled file into a program handle.

Format

ProgramHandle_t *GAUSS_LoadCompiledFile( WorkspaceHandle_t *wh, char *gcgfile );

ph = GAUSS_LoadCompiledFile( wh, gcgfile );

Input

wh pointer to a workspace handle.

gcgfile pointer to name of a compiled file.

Output

ph pointer to a program handle.

Remarks

GAUSS_LoadCompiledFile takes a compiled file and loads it into a workspace. It returns a program handle pointer you can use in GAUSS_Execute to execute the program.

Call GAUSS_LoadCompiledFile with a WorkspaceHandle_t pointer returned from GAUSS_CreateWorkspace.

If GAUSS_LoadCompiledFile fails, ph will be NULL. Use GAUSS_GetError to get the number of the error. GAUSS_LoadCompiledFile may fail with either of the following errors:

30 Insufficient memory.
494 Invalid argument number.

Example

ProgramHandle_t *ph1, *ph2;
int ret;

if ( ( ph1 = GAUSS_CompileString(
wh1,
"{ a, rs } = rndKMn( 4,4,31 ); b = det( a );",
0,
The above example compiles a string into one workspace, saves the program information into a file, and then loads the program information into another workspace. It assumes that `wh1` and `wh2` are pointers to valid workspace handles.

**See also**

`GAUSS_Execute`, `GAUSS_CompileFile`, `GAUSS_CompileStringAsFile`, `GAUSS_SaveProgram`
8. C API: REFERENCE

**GAUSS_LoadWorkspace**

- **Purpose**
  Loads workspace information stored in a file.

- **Format**
  ```c
  WorkspaceHandle_t *GAUSS_LoadWorkspace( char *file );
  wh = GAUSS_LoadWorkspace( file );
  ```

- **Input**
  - `file` pointer to name of a compiled file.

- **Output**
  - `wh` pointer to a workspace handle.

- **Remarks**
  `GAUSS_LoadWorkspace` gets the workspace information saved in a file and returns it in a workspace handle.

  If `GAUSS_LoadWorkspace` fails, `wh` will be NULL. Use `GAUSS_GetError` to get the number of the error. `GAUSS_LoadWorkspace` may fail with either of the following errors:

  - 30 Insufficient memory.
  - 495 Workspace inactive or corrupt.

- **See also**
  `GAUSS_CreateWorkspace`, `GAUSS_SaveWorkspace`, `GAUSS_FreeWorkspace`
**GAUSS_Matrix**

- **Purpose**
  Creates a `Matrix_t` for a real matrix and copies the matrix data.

- **Format**
  ```c
  Matrix_t *GAUSS_Matrix(unsigned int rows, unsigned int cols, double *addr);
  
  mat = GAUSS_Matrix(rows, cols, addr);
  ```

- **Input**
  - `rows` number of rows.
  - `cols` number of columns.
  - `addr` pointer to matrix.

- **Output**
  - `mat` pointer to a matrix descriptor.

- **Remarks**
  `GAUSS_Matrix` malloc's a `Matrix_t` and fills it in with your input information. It makes a copy of the matrix and sets the `mdata` member of the `Matrix_t` to point to the copy. `GAUSS_Matrix` should only be used for real matrices. To create a `Matrix_t` for a complex matrix, use `GAUSS_ComplexMatrix`. To create a `Matrix_t` for a real matrix without making a copy of the matrix, use `GAUSS_MatrixAlias`.

  To create a `Matrix_t` for an empty matrix, set `rows` and `cols` to 0 and `addr` to NULL.

  If `mat` is NULL, there was insufficient memory to malloc space for the matrix and its descriptor.

  Use this function to create a matrix descriptor that you can use in the following functions:

  ```c
  GAUSS_CopyMatrixToArg
  GAUSS_CopyMatrixToGlobal
  GAUSS_MoveMatrixToArg
  GAUSS_MoveMatrixToGlobal
  ```

  Free the `Matrix_t` with `GAUSS_FreeMatrix`.

- **Example**
  ```c
  ```

double m[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };  
int ret;  
if ( ret = GAUSS_MoveMatrixToGlobal(  
    wh,  
    GAUSS_Matrix( 2, 3, &m[0][0] ),  
    "a"  
) )  
{
    char buff[100];  
    printf( "GAUSS_MoveMatrixToGlobal failed: %s\n",  
        GAUSS_ErrorText( buff, ret ) );  
    return -1;  
}

The above example uses `GAUSS_Matrix` to copy a local matrix into a `Matrix_t` structure, and moves the matrix into a `GAUSS` workspace. It assumes that `wh` is a pointer to a valid workspace handle.

**See also**

- `GAUSS_ComplexMatrix`
- `GAUSS_MatrixAlias`
- `GAUSS_CopyMatrixToGlobal`
- `GAUSS_CopyMatrixToArg`
- `GAUSS_MoveMatrixToGlobal`
- `GAUSS_MoveMatrixToArg`
- `GAUSS_FreeMatrix`
GAUSS_MatrixAlias

- **Purpose**

  Creates a `Matrix_t` for a real matrix.

- **Format**

  ```c
  Matrix_t *GAUSS_MatrixAlias( unsigned int rows, unsigned int cols, double *addr );
  ```

  ```c
  mat = GAUSS_MatrixAlias( rows, cols, addr );
  ```

- **Input**

  - `rows` number of rows.
  - `cols` number of columns.
  - `addr` pointer to matrix.

- **Output**

  ```c
  mat     pointer to a matrix descriptor.
  ```

- **Remarks**

  `GAUSS_MatrixAlias` is similar to `GAUSS_Matrix`; however, it sets the `mdata` member of the `Matrix_t` to point to the matrix indicated by `addr` instead of making a copy of the matrix. `GAUSS_MatrixAlias` should only be used for real matrices. For complex matrices, use `GAUSS_ComplexMatrixAlias`.

  If `mat` is NULL, there was insufficient memory to `malloc` space for the matrix descriptor.

  Use this function to create a matrix descriptor that you can use in the following functions:

  ```c
  GAUSS_CopyMatrixToArg
  GAUSS_CopyMatrixToGlobal
  GAUSS_MoveMatrixToArg
  GAUSS_MoveMatrixToGlobal
  ```

  Free the `Matrix_t` with `GAUSS_FreeMatrix`. It will not free the matrix data.

- **Example**

  ```c
  136
  ```
Matrix_t *mat;
double *a;
int ret;

a = (double *)malloc( 9*sizeof(double) );
memset( a, 0, 9*sizeof(double) );

if ( ( mat = GAUSS_MatrixAlias( 3, 3, &a ) ) == NULL )
{
    char buff[100];
    printf( "MatrixAlias failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ret = GAUSS_CopyMatrixToGlobal( wh, mat, "c" ) )
{
    char buff[100];
    printf( "CopyMatrixToGlobal failed: %s\n", GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeMatrix( mat );
    return -1;
}

This example `malloc`'s a matrix of zeroes and then creates a `Matrix_t` for the matrix. It copies the matrix to `wh`, which it assumes to be a pointer to a valid workspace.

### See also

- `GAUSS___Matrix`
- `GAUSS___ComplexMatrixAlias`
- `GAUSS___CopyMatrixToGlobal`
- `GAUSS___CopyMatrixToArg`
- `GAUSS___MoveMatrixToGlobal`
- `GAUSS___MoveMatrixToArg`
- `GAUSS___FreeMatrix`
**GAUSS_MoveArgToArg**

---

### Purpose

Moves an argument from one `ArgList_t` to another.

### Format

```c
int GAUSS_MoveArgToArg( ArgList_t *targs, int targnum, ArgList_t *sargs, int sargnum );
```

```c
ret = GAUSS_MoveArgToArg( targs, targnum, sargs, sargnum );
```

### Input

- `targs`: pointer to target argument list structure.
- `targnum`: number of argument in target argument list.
- `sargs`: pointer to source argument list structure.
- `sargnum`: number of argument in source argument list.

### Output

- `ret`: success flag, 0 if successful, otherwise:
  - 30: Insufficient memory.
  - 94: Argument out of range.

### Remarks

`GAUSS_MoveArgToArg` moves the `sargnum` argument in `sargs` to `targs`. It clears the `sargnum` argument descriptor after moving the argument indicated by it. However, it does not change the number of arguments in `sargs`. Therefore, you can overwrite the `sargnum` argument of `sargs` by copying or moving another argument into it.

To add an argument to the end of an argument list or to an empty argument list, set `targnum` to 0. To replace an argument, set `targnum` to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call `GAUSS_InsertArg` and then set `targnum` to the number of the inserted argument. Arguments are numbered starting with 1.

The argument’s data will be freed when you call `GAUSS_CallProcFreeArgs` or `GAUSS_FreeArgList` later.

If you want to retain the argument in `sargs`, use `GAUSS_CopyMatrixToArg` instead. However, `GAUSS_MoveMatrixToArg` saves time and memory space.

### Example
8. C API: REFERENCE

GAUSS_MoveArgToArg

ArgList_t *marg( WorkspaceHandle_t *wh, ArgList_t *args )
{
    ProgramHandle_t *ph;
    ArgList_t *ret;

    if ( ( ph = GAUSS_CompileExpression(wh, "rndKMi(100,4);", 1, 1) ) == NULL )
    {
        char buff[100];
        printf("Compile failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ));
        return NULL;
    }

    if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
    {
        char buff[100];
        printf("Execute failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ));
        GAUSS_FreeProgram( ph );
        return NULL;
    }

    if ( GAUSS_MoveArgToArg( args, 2, ret, 2 ) )
    {
        char buff[100];
        printf("MoveArgToArg failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ));
        GAUSS_FreeProgram( ph );
        GAUSS_FreeArgList( ret );
        return NULL;
    }

    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( ret );

    return args;
}
The above example compiles an expression in `wh`, which gives its return in an `ArgList_t`. It moves the second argument contained in `ret` into `args` as its second argument. It assumes that `args` has at least two arguments, and it overwrites the second argument of `args`.

**See also**

- `GAUSS_MoveArgToArg`
- `GAUSS_CreateArgList`
- `GAUSS_InsertArg`
- `GAUSS_FreeArgList`
- `GAUSS_CallProc`
- `GAUSS_CallProcFreeArgs`
Purpose
Moves a matrix from an ArgList_t to a Matrix_t structure.

Format
Matrix_t *GAUSS_MoveArgToMatrix( ArgList_t *args, int argnum);

mat = GAUSS_MoveArgToMatrix( args, argnum );

Input
args     pointer to an argument list structure.
argnum   number of argument in the argument list.

Output
mat      pointer to a matrix descriptor.

Remarks
GAUSS_MoveArgToMatrix creates a matrix descriptor, mat, and moves a matrix contained in args into it. mat belongs to you. Free it with GAUSS_FreeMatrix.

GAUSS_MoveArgToMatrix clears the argnum argument descriptor after moving the matrix indicated by it. However, it does not change the number of arguments in args. Therefore, you can overwrite the argnum argument of args by copying or moving another argument into it. Arguments are numbered starting with 1.

If you want to retain the matrix in the ArgList_t, use GAUSS_CopyArgToMatrix instead. However, GAUSS_MoveArgToMatrix saves time and memory space.

If GAUSS_MoveArgToMatrix fails, mat will be NULL. Use GAUSS_GetError to get the number of the error. GAUSS_MoveArgToMatrix may fail with any of the following errors:

30       Insufficient memory.
71       Type mismatch.
94       Argument out of range.

Example
programHandle_t *ph;
ArgumentList_t *ret;
Matrix_t *mat;

if ( ( ph = GAUSS_CompileExpression(wh,
  "band( reshape( seqa( 1,2,20 ),5,4 ),2 );",
    1,
    1
  ) ) == NULL )
{
  char buff[100];

  printf( "Compile failed: %s\n",
    GAUSS_ErrorText( buff, GAUSS_GetError() ) );
  return -1;
}

if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
{
  char buff[100];

  printf( "Execute failed: %s\n",
    GAUSS_ErrorText( buff, GAUSS_GetError() ) );
  GAUSS_FreeProgram( ph );
  return -1;
}

if ( ( mat = GAUSS_MoveArgToMatrix( ret, 1 ) ) == NULL )
{
  char buff[100];

  printf( "MoveArgToMatrix failed: %s\n",
    GAUSS_ErrorText( buff, GAUSS_GetError() ) );
  GAUSS_FreeProgram( ph );
  GAUSS_FreeArgList( ret );
  return -1;
}

This example assumes that wh is a pointer to a valid workspace handle.

**See also**

- GAUSS_MoveArgToMatrix
- GAUSS_CopyArgToMatrix
- GAUSS_CallProc
- GAUSS_CallProcFreeArgs
- GAUSS_ExecuteExpression
- GAUSS_FreeMatrix
- GAUSS_GetArgType
- GAUSS_GetError
GAUSS_MoveArgToString

- **Purpose**
  Moves a string from an ArgList_t to a String_t structure.

- **Format**
  
  ```c
  String_t *GAUSS_MoveArgToString( ArgList_t *args, int argnum );
  ```

  ```c
  str = GAUSS_MoveArgToString( args, argnum );
  ```

- **Input**
  - `args` pointer to an argument list structure.
  - `argnum` number of argument in the argument list.

- **Output**
  - `str` pointer to a string descriptor.

- **Remarks**
  GAUSS_MoveArgToString creates a String_t, `str`, and moves a string contained in `args` into it. `str` belongs to you. Free it with GAUSS_FreeString.

  GAUSS_MoveArgToString clears the `argnum` argument descriptor after moving the string indicated by it. However, it does not change the number of arguments in `args`. Therefore, you can overwrite the `argnum` argument of `args` by copying or moving another argument into it. Arguments are numbered starting with 1.

  If you want to retain the string in the ArgList_t, use GAUSS_CopyArgToString instead. However, GAUSS_MoveArgToString saves time and memory space.

  If GAUSS_MoveArgToString fails, `str` will be NULL. Use GAUSS_GetError to get the number of the error. GAUSS_MoveArgToString may fail with any of the following errors:

  - 30 Insufficient memory.
  - 71 Type mismatch.
  - 94 Argument out of range.

- **Example**
GAUSS_MoveArgToString

ProgramHandle_t *ph;
ArgList_t *ret;
String_t *str;

if ( ( ph = GAUSS_CompileExpression(wh, ""\"output\"$+"\".log\";", 1, 1) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
{
    char buff[100];

    printf( "Execute failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( ( str = MoveArgToString( args, 1 ) ) == NULL )
{
    char buff[100];

    printf( "MoveArgToString failed: %s\n", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( ret );
    return -1;
}

This example assumes that wh is a pointer to a valid workspace handle.

- See also

GAUSS_CopyArgToString, GAUSS_CallProc, GAUSS_CallProcFreeArgs,
GAUSS_ExecuteExpression, GAUSS_FreeString, GAUSS_GetArgType,
GAUSS_GetError

144
Purpose

Moves a string array from an `ArgList_t` to a `StringArray_t` structure.

Format

```
StringArray_t *GAUSS_MoveArgToStringArray( ArgList_t *args, int argnum);
```

```
sa = GAUSS_MoveArgToStringArray( args, argnum );
```

Input

- `args` : pointer to an argument list structure.
- `argnum` : number of argument in the argument list.

Output

- `sa` : pointer to a string array descriptor.

Remarks

`GAUSS_MoveArgToStringArray` creates a `StringArray_t`, `sa`, and moves a string array contained in `args` into it. `sa` belongs to you. Free it with `GAUSS_FreeStringArray`.

`GAUSS_MoveArgToStringArray` clears the `argnum` argument descriptor after moving the string array indicated by it. However, it does not change the number of arguments in `args`. Therefore, you can overwrite the `argnum` argument of `args` by copying or moving another argument into it. Arguments are numbered starting with 1.

If you want to retain the string array in the `ArgList_t`, use `GAUSS_CopyArgToStringArray` instead. However, `GAUSS_MoveArgToStringArray` saves time and memory space.

If `GAUSS_MoveArgToStringArray` fails, `sa` will be NULL. Use `GAUSS_GetError` to get the number of the error. `GAUSS_MoveArgToStringArray` may fail with any of the following errors:

- 30 : Insufficient memory.
- 71 : Type mismatch.
- 94 : Argument out of range.

Example
ProgramHandle_t *ph;
ArgList_t *ret;
StringArray_t *sa;

if ( ( ph = GAUSS_CompileExpression(wh, ""one" $| "two" $| "three";", 1, 1) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s
", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    return -1;
}

if ( ( ret = GAUSS_ExecuteExpression( ph ) ) == NULL )
{
    char buff[100];

    printf( "Execute failed: %s
", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    return -1;
}

if ( ( sa = GAUSS_MoveArgToStringArray( args, 1 ) ) == NULL )
{
    char buff[100];

    printf( "MoveArgToStringArray failed: %s
", GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( sa );
    return -1;
}

This example assumes that wh is a pointer to a valid workspace handle.

- See also

  GAUSS_MoveArgToStringArray, GAUSS_CallProc, GAUSS_CallProcFreeArgs,
  GAUSS_ExecuteExpression, GAUSS_FreeStringArray, GAUSS_GetArgType,
  GAUSS_GetError
Purpose

Moves a matrix contained in a Matrix_t to an ArgList_t and frees the Matrix_t.

Format

```
int GAUSS_MoveMatrixToArg( ArgList_t *args, Matrix_t *mat, int argnum );
```

```
ret = GAUSS_MoveMatrixToArg( args, mat, argnum );
```

Input

- `args` pointer to an argument list structure.
- `mat` pointer to a matrix descriptor.
- `argnum` number of argument.

Output

- `ret` success flag, 0 if successful, otherwise:
  - 30 Insufficient memory.
  - 494 Invalid argument number.

Remarks

`GAUSS_MoveMatrixToArg` moves the matrix contained in `mat` into `args` and frees `mat`.

To add an argument to the end of an argument list or to an empty argument list, set `argnum` to 0. To replace an argument, set `argnum` to the number of the argument you want to replace. It will overwrite that argument's information and free its data. To insert an argument, call `GAUSS_InsertArg` and then set `argnum` to the number of the inserted argument. Arguments are numbered starting with 1.

The matrix will be freed when you call `GAUSS_CallProcFreeArgs` or `GAUSS_FreeArgList` later.

If you want to retain `mat`, use `GAUSS_CopyMatrixToArg` instead. However, `GAUSS_MoveMatrixToArg` saves time and memory space.

Call `GAUSS_MoveMatrixToArg` with a Matrix_t returned from `GAUSS_Matrix`, `GAUSS_ComplexMatrix`, or `GAUSS_GetMatrix`.

See also

- `GAUSS_CopyMatrixToArg`, `GAUSS_Matrix`, `GAUSS_ComplexMatrix`
- `GAUSS_CreateArgList`, `GAUSS_FreeArgList`, `GAUSS_InsertArg`, `GAUSS_CallProc`, `GAUSS_CallProcFreeArgs`
GAUSS_MoveMatrixToGlobal

- **Purpose**
  
  Moves a matrix contained in a Matrix_t into a GAUSS workspace and frees the Matrix_t.

- **Format**
  
  ```c
  int GAUSS_MoveMatrixToGlobal( WorkspaceHandle_t *wh, Matrix_t *mat, char *name );
  ```

  ```c
  ret = GAUSS_MoveMatrixToGlobal( wh, mat, name );
  ```

- **Input**
  
  - `wh` pointer to a workspace handle.
  - `mat` pointer to a matrix descriptor.
  - `name` name of matrix.

- **Output**
  
  - `ret` success flag, 0 if successful, otherwise:
    
    - 26 Too many symbols.
    - 30 Insufficient memory.
    - 91 Symbol too long.
    - 481 GAUSS assignment failed.
    - 495 Workspace inactive or corrupt.

- **Remarks**
  
  GAUSS_MoveMatrixToGlobal moves the matrix contained in `mat` into a GAUSS workspace and frees `mat`. GAUSS takes ownership of the matrix and frees it when necessary.

  If you want to retain `mat`, use GAUSS_CopyMatrixToGlobal instead. However, GAUSS_MoveMatrixToGlobal saves time and memory space.

  Call GAUSS_MoveMatrixToGlobal with a Matrix_t returned from GAUSS_Matrix, GAUSS_ComplexMatrix, or GAUSS_GetMatrix.

  Input a WorkspaceHandle_t returned from GAUSS_CreateWorkspace.

- **See also**
  
  GAUSS_CopyMatrixToGlobal, GAUSS_Matrix, GAUSS_ComplexMatrix, GAUSS_AssignFreeableMatrix, GAUSS_GetMatrix, GAUSS_PutDouble
GAUSS_MoveStringArrayToArg

- **Purpose**
  Moves a string array contained in a `StringArray_t` to an `ArgList_t` and frees the `StringArray_t`.

- **Format**
  ```c
  int GAUSS_MoveStringArrayToArg( ArgList_t *args, StringArray_t *sa, int argnum );
  ```
  
  ```c
  ret = GAUSS_MoveStringArrayToArg( args, sa, argnum );
  ```

- **Input**
  - `args` pointer to an argument list structure.
  - `sa` pointer to a string array descriptor.
  - `argnum` number of argument.

- **Output**
  - `ret` success flag, 0 if successful, otherwise:
    - 30 Insufficient memory.
    - 494 Invalid argument number.

- **Remarks**
  **GAUSS_MoveStringArrayToArg** moves the string array contained in `sa` into `args` and frees `sa`.

  To add an argument to the end of an argument list or to an empty argument list, set `argnum` to 0. To replace an argument, set `argnum` to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call **GAUSS_InsertArg** and then set `argnum` to the number of the inserted argument. Arguments are numbered starting with 1.

  The string array will be freed when you call **GAUSS_CallProcFreeArgs** or **GAUSS_FreeArgList** later.

  If you want to retain `sa`, use **GAUSS_CopyStringArrayToArg** instead. However, **GAUSS_MoveStringArrayToArg** saves time and memory space.

  Create a `StringArray_t` with **GAUSS_StringArray** or **GAUSS_StringArrayL**, or use a `StringArray_t` returned from **GAUSS_GetStringArray**.

- **See also**
  **GAUSS_MoveStringArrayToArg**, **GAUSS_StringArray**, **GAUSS_StringArrayL**, **GAUSS_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_InsertArg**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**
GAUSS__MoveStringArrayToGlobal

- **Purpose**

  Moves a string array contained in a `StringArray_t` into a GAUSS workspace and frees the `StringArray_t`.

- **Format**

  ```c
  int GAUSS__MoveStringArrayToGlobal( WorkspaceHandle_t *wh, StringArray_t *sa, char *name );
  ``

  ```c
  ret = GAUSS__MoveStringArrayToGlobal( wh, sa, name );
  ```

- **Input**

  - `wh`  pointer to a workspace handle.
  - `sa`  pointer to string array descriptor.
  - `name`  pointer to name of string array.

- **Output**

  - `ret`  success flag, 0 if successful, otherwise:
    - 26  Too many symbols.
    - 30  Insufficient memory.
    - 91  Symbol too long.
    - 481  GAUSS assignment failed.
    - 495  Workspace inactive or corrupt.

- **Remarks**

  `GAUSS__MoveStringArrayToGlobal` moves the string array contained in `sa` into a GAUSS workspace and frees `sa`. GAUSS takes ownership of the string array and frees it when necessary.

  If you want to retain `sa`, use `GAUSS__CopyStringArrayToGlobal` instead. However, `GAUSS__MoveStringArrayToGlobal` saves time and memory space.

  Create a `StringArray_t` with `GAUSS__StringArray` or `GAUSS__StringArrayL`, and call `GAUSS__MoveStringArrayToGlobal` with a `WorkspaceHandle_t` returned from `GAUSS__CreateWorkspace`.

- **See also**

  `GAUSS__CopyStringArrayToGlobal`, `GAUSS__StringArray`, `GAUSS__StringArrayL`, `GAUSS__GetStringArray`
GAUSS_MoveStringToArg

Purpose

Moves a string contained in a String_t to an ArgList_t and frees the String_t.

Format

```c
int GAUSS_MoveStringToArg( ArgList_t *args, String_t *str, int argnum );
```

Input

- `args` pointer to an argument list structure.
- `str` pointer to a string descriptor.
- `argnum` number of argument.

Output

- `ret` success flag, 0 if successful, otherwise:
  - 30 Insufficient memory.
  - 494 Invalid argument number.

Remarks

GAUSS_MoveStringToArg moves the string contained in `str` into `args` and frees `str`.

To add an argument to the end of an argument list or to an empty argument list, set `argnum` to 0. To replace an argument, set `argnum` to the number of the argument you want to replace. It will overwrite that argument’s information and free its data. To insert an argument, call GAUSS_InsertArg and then set `argnum` to the number of the inserted argument. Arguments are numbered starting with 1.

The string will be freed when you call GAUSS_CallProcFreeArgs or GAUSS_FreeArgList later.

If you want to retain `str`, use GAUSS_CopyStringToArg instead. However, GAUSS_MoveStringToArg saves time and memory space.

Call GAUSS_MoveStringToArg with a String_t returned from GAUSS_String, GAUSS_StringL, or GAUSS_GetString.

See also

- GAUSS_CopyStringToArg
- GAUSS_String
- GAUSS_StringL
- GAUSS_CallProc
- GAUSS_CallProcFreeArgs
GAUSS_MoveStringToGlobal

**Purpose**

Moves a string contained in a *String_t* into a GAUSS workspace and frees the *String_t*.

**Format**

```c
int GAUSS_MoveStringToGlobal( WorkspaceHandle_t *wh, String_t *str, char *name );
```

```c
ret = GAUSS_MoveStringToGlobal( wh, str, name );
```

**Input**

- `wh`: pointer to a workspace handle.
- `str`: pointer to string descriptor.
- `name`: pointer to name of string.

**Output**

- `ret`: success flag, 0 if successful, otherwise:
  - 26: Too many symbols.
  - 30: Insufficient memory.
  - 91: Symbol too long.
  - 481: GAUSS assignment failed.
  - 495: Workspace inactive or corrupt.

**Remarks**

`GAUSS_MoveStringToGlobal` moves the string contained in `str` into a GAUSS workspace and frees `str`. GAUSS takes ownership of the string and frees it when necessary.

If you want to retain `str`, use `GAUSS_CopyStringToGlobal` instead. However, `GAUSS_MoveStringToGlobal` saves time and memory space.

Call `GAUSS_MoveStringToGlobal` with a `String_t` returned from `GAUSS_String`, `GAUSS_StringL`, or `GAUSS_GetString`.

Input a `WorkspaceHandle_t` returned from `GAUSS_CreateWorkspace`.

**See also**

`GAUSS_CopyStringToGlobal`, `GAUSS_String`, `GAUSS_StringL`, `GAUSS_GetString`
8. C API: REFERENCE

GAUSS_PutDouble

- **Purpose**
  
  Puts a `double` into a **GAUSS** workspace.

- **Format**

  ```c
  int GAUSS_PutDouble( WorkspaceHandle_t *wh, double d, char *name );
  
  ret = GAUSS_PutDouble( wh, d, name );
  ```

- **Input**

  - `wh` pointer to a workspace handle.
  - `d` data.
  - `name` pointer to name of symbol.

- **Output**

  - `ret` success flag, 0 if successful, otherwise:
    
    | Code | Description               |
    |------|---------------------------|
    | 26   | Too many symbols.         |
    | 91   | Symbol too long.          |
    | 481  | **GAUSS** assignment failed. |
    | 495  | Workspace inactive or corrupt. |

- **Remarks**

  **GAUSS_PutDouble** puts a double into a **GAUSS** workspace.

  Call **GAUSS_PutDouble** with a `WorkspaceHandle_t` returned from **GAUSS_CreateWorkspace**.

- **See also**

  **GAUSS_GetDouble**, **GAUSS_CopyMatrixToGlobal**, **GAUSS_MoveMatrixToGlobal**
GAUSS___SaveProgram

■ Purpose

Saves a compiled program as a file.

■ Format

```c
int GAUSS___SaveProgram( ProgramHandle_t *ph, char *fn);
```

`ret = GAUSS___SaveProgram( ph, fn );`

■ Input

`ph` pointer to a program handle.

`fn` pointer to name of file.

■ Output

`ret` success code, 0 if successful, 1 if not.

- 10 Can’t open output file.
- 30 Insufficient memory.
- 132 Can’t write, disk probably full.
- 495 Workspace inactive or corrupt.
- 496 Program inactive or corrupt.

■ Remarks

`GAUSS___SaveProgram` saves a compiled program given by a program handle into a file. It saves all of the workspace information, which is contained in the program handle. The file will have the name given by `fn`. Load the program with `GAUSS___LoadCompiledFile`.

■ See also

`GAUSS___CompileString`, `GAUSS___CompileFile`, `GAUSS___CompileStringAsFile`, `GAUSS___LoadCompiledFile`, `GAUSS___FreeProgram`
8. **C API: REFERENCE**

**GAUSS_SaveWorkspace**

- **Purpose**
  
  Saves workspace information in a file.

- **Format**

  ```c
  int GAUSS_SaveWorkspace( WorkspaceHandle_t *wh, char *fn );
  ```

  ```c
  ret = GAUSS_SaveWorkspace( wh, fn );
  ```

- **Input**

  - `wh`  
    
    pointer to a workspace handle.

  - `fn`  
    
    pointer to name of file.

- **Output**

  - `ret`  
    
    success code, 0 if successful, otherwise:

    - 10  
      Can’t open output file.

    - 30  
      Insufficient memory.

    - 132  
      Can’t write, disk probably full.

    - 495  
      Workspace inactive or corrupt.

- **Remarks**

  `GAUSS_SaveWorkspace` saves workspace information contained in a workspace handle into a file. The file will have the name given by `fn`. Load the workspace information with `GAUSS_LoadWorkspace`.

- **See also**

  `GAUSS_CreateWorkspace`, `GAUSS_LoadWorkspace`, `GAUSS_FreeWorkspace`
GAUSS_SetError

■ Purpose

Sets the stored error number and returns the previous error number.

■ Format

```c
int GAUSS_SetError( int newerrnum );
olderrnum = GAUSS_SetError( newerrnum );
```

■ Input

`newerrnum`  new error number.

■ Output

`olderrnum`  previous error number.

■ Remarks

The GAUSS Enterprise Engine stores the error number of the most recently encountered error in a system variable. If a GAUSS Enterprise Engine command fails, it automatically resets this variable with the number of the error. However, the command does not clear the variable if it succeeds.

Use GAUSS_SetError to manually reset the variable. It returns the error number that was previously stored in the variable.

The system variable is global to the current thread.

■ See also

GAUSS_GetError, GAUSS_ErrorText
Purpose

Sets the home path for the GAUSS Enterprise Engine.

Format

```c
int GAUSS__SetHome( char *path );
```

```
ret = GAUSS__SetHome( path );
```

Input

`path` pointer to path to be set.

Output

`ret` success code, 0 if successful, otherwise 486 if character argument too long.

Remarks

`GAUSS__SetHome` specifies the home directory used to locate the Run-Time Library, source files, library files, etc. in a normal engine installation. It overrides any environment variable. Call `GAUSS__SetHome` before calling `GAUSS__Initialize`.

See also

`GAUSS__SetHomeVar`, `GAUSS__GetHome`, `GAUSS__GetHomeVar`, `GAUSS__Initialize`
GAUSS_SetHomeVar

- **Purpose**
  
  Sets the name of the home environment variable for the GAUSS Enterprise Engine.

- **Format**
  
  ```c
  int GAUSS_SetHomeVar( char *newname );
  ```

  ```c
  ret = GAUSS_SetHomeVar( newname );
  ```

- **Input**
  
  ```c
  newname  pointer to new name to be set.
  ```

- **Output**
  
  ```c
  ret  success code, 0 if successful, otherwise 486 if character argument too long.
  ```

- **Remarks**
  
  The default value is **MTENGBOME**. Use the C library function `getenv` to get the value of the environment variable.

  It is better to use **GAUSS_SetHome** which sets the home directory, overriding the environment variable. Call **GAUSS_SetHomeVar** or **GAUSS_SetHome** before calling **GAUSS.Initialize**.

- **See also**
  
  **GAUSS_SetHome**, **GAUSS_GetHomeVar**, **GAUSS_GetHome**, **GAUSS.Initialize**
SetLogFile

Purpose
Sets the file for logged errors.

Format

```c
int GAUSS_SetLogFile( char *logfn, char *mode );
```

```c
ret = GAUSS_SetLogFile( logfn, mode );
```

Input

- `logfn` name of log file.
- `mode` "w" to overwrite the contents of the file.
  "a" to append to the contents of the file.

Output

- `ret` success flag, 0 if successful, otherwise:
  - 484 Cannot open log file.
  - 485 Cannot write to log file.

Remarks

The GAUSS Enterprise Engine logs certain system level errors in 2 places: a file and an open file pointer. The default file is `/tmp/mteng.###.log` where `###` is the process ID number. The default file pointer is `stderr`

`GAUSS_SetLogFile` allows you to set the file that the errors will be logged in. You can turn off the error logging to file by inputting a NULL pointer for `logfn`.

See also

- `GAUSS_GetLogFile`
- `GAUSS_SetLogStream`
- `GAUSS_GetLogStream`
GAUSS__SetLogStream

■ Purpose
Sets the file pointer for logged errors.

■ Format
void GAUSS__SetLogStream( FILE *logfp );
GAUSS__SetLogStream( logfp );

■ Input

logfp file pointer of an open file.

■ Remarks
The GAUSS Enterprise Engine logs certain system level errors in 2 places: a file and an open file pointer. The default file is /tmp/mteng.###.log where ### is the process ID number. The default file pointer is stderr.

GAUSS__SetLogStream allows you to set the file pointer that the errors will be logged to. You can turn off the error logging to file pointer by inputting a NULL pointer for logfp.

■ See also

GAUSS__GetLogStream, GAUSS__SetLogFile, GAUSS__GetLogFile
GAUSS\_SetWorkspaceName

- **Purpose**
  Sets the name of a GAUSS workspace.

- **Format**
  
  \[
  \text{int GAUSS\_SetWorkspaceName( WorkspaceHandle\_t *wh, char *name );}
  \]

  
  \[
  ret = \text{GAUSS\_SetWorkspaceName( wh, name );}
  \]

- **Input**
  
  \[
  wh \quad \text{pointer to a workspace handle.}
  \]
  
  \[
  name \quad \text{pointer to the new workspace name.}
  \]

- **Input**
  
  \[
  ret \quad \text{success flag, 0 if successful, otherwise 495 if the workspace is inactive or corrupt.}
  \]

- **Remarks**
  
  **GAUSS\_SetWorkspaceName** sets the name of a GAUSS workspace indicated by a WorkspaceHandle\_t.

- **See also**
  
  **GAUSS\_GetWorkspaceName, GAUSS\_CreateWorkspace**
GAUSS_Shutdown

- **Purpose**
  
  Shuts down the engine, preparatory to ending the application.

- **Format**

  ```
  void GAUSS_Shutdown( void );
  
  GAUSS_Shutdown();
  ```

- **Remarks**

  **GAUSS_Shutdown** cleans up any temporary files generated by the engine. It also closes any dynamic libraries used by the foreign language interface. You should call it once at the close of your application after freeing any open pointers.

- **See also**

  **GAUSS_Initialize**
8. **C API: REFERENCE**

**GAUSS_String**

- **Purpose**
  
  Creates a `String_t` and copies the string data.

- **Format**
  
  ```c
  String_t *GAUSS_String( char *str);
  strdesc = GAUSS_String( str);
  ```

- **Input**
  
  `str` pointer to string.

- **Output**
  
  `strdesc` pointer to a string descriptor.

- **Remarks**
  
  **GAUSS_String malloc**'s a `String_t` and fills it in with your input information. It makes a copy of the string and sets the `stdata` member of the `String_t` to point to the copy. To create a `String_t` for your string without making a copy of it, use **GAUSS_StringAlias**.

  This function uses `strlen` to determine the length of the string. Since `strlen` only computes the length of a string to the first null byte, your string may not contain embedded 0's. To create a `String_t` with a string that contains embedded 0's, use **GAUSS_StringL**.

  If `strdesc` is NULL, there was insufficient memory to `malloc` space for the string and its descriptor.

  Use this function to create a string descriptor that you can use in the following functions:

  ```c
  GAUSS_CopyStringToArg
  GAUSS_CopyStringToGlobal
  GAUSS_MoveStringToArg
  GAUSS_MoveStringToGlobal
  ```

  Free the `String_t` with **GAUSS_FreeString**.

- **See also**
  
  **GAUSS_StringL**, **GAUSS_StringAlias**, **GAUSS_StringAliasL**, **GAUSS_FreeString**

163
GAUSS\_StringAlias

- **Purpose**
  Creates a String\_t.

- **Format**

  \[
  \text{String}\_t \ *\text{GAUSS\_StringAlias}( \text{char } *str );
  \]

  \[
  strdesc = \text{GAUSS\_StringAlias}( str );
  \]

- **Input**

  \[
  str \quad \text{pointer to string.}
  \]

- **Output**

  \[
  strdesc \quad \text{pointer to a string descriptor.}
  \]

- **Remarks**

  \text{GAUSS\_StringAlias} is similar to \text{GAUSS\_String}; however, it sets the stdata member of the String\_t to point to \text{str} instead of making a copy of the string.

  This function uses strlen to determine the length of the string. Since strlen only computes the length of a string to the first null byte, your string may not contain embedded 0’s. To create a String\_t with a string that contains embedded 0’s, use \text{GAUSS\_StringAliasL}.

  If strdesc is NULL, there was insufficient memory to malloc space for the string descriptor.

  Use this function to create a string descriptor that you can use in \text{GAUSS\_CopyStringToArg} and \text{GAUSS\_CopyStringToGlobal}.

  Free the String\_t with \text{GAUSS\_FreeString}. It will not free the string data.

- **See also**

  \text{GAUSS\_String, GAUSS\_StringAliasL, GAUSS\_StringL, GAUSS\_FreeString}
Purpose

Creates a String_t with string of user-specified length.

Format

String_t *GAUSS_StringAliasL( char *str, int len );

strdesc = GAUSS_StringAliasL( str, len );

Input

str    pointer to string.
len    length of string, including null terminator.

Output

strdesc    pointer to a string descriptor.

Remarks

GAUSS_StringAliasL is similar to GAUSS_StringL; however, it sets the stdata member of the String_t to point to str instead of making a copy of the string.

This function takes the length of the string from the len argument rather than calling strlen, which computes the length of a string only to the first null byte. This allows your string to contain embedded 0’s. If your string does not contain embedded 0’s, you can use GAUSS_StringAlias to create your String_t.

If strdesc is NULL, there was insufficient memory to malloc space for the string descriptor.

Use this function to create a string descriptor that you can use in GAUSS_CopyStringToArg and GAUSS_CopyStringToGlobal.

You can free the String_t with GAUSS_FreeString. It will not free the string data.

See also

GAUSS_String, GAUSS_StringAlias, GAUSS_StringL, GAUSS_FreeString
GAUSS__StringArray

- **Purpose**
  Creates a StringArray_t and copies the string array data.

- **Format**
  ```c
  StringArray_t *GAUSS__StringArray( unsigned int rows, unsigned int cols, char **strs );
  ```
  ```c
  sa = GAUSS__StringArray( rows, cols, strs );
  ```

- **Input**
  - `rows` number of rows.
  - `cols` number of columns.
  - `strs` pointer to an array of character pointers containing the strings of the array.

- **Output**
  - `sa` pointer to a string array descriptor.

- **Remarks**
  `GAUSS__StringArray` malloc’s a StringArray_t and fills it in with your input information. It makes a copy of all the strings in the array and creates an array of `rows`*`cols` StringElement_t’s. The table member of the StringArray_t is set to the address of the array of StringElement_t’s.

  This function uses strlen to determine the lengths of the strings. Since strlen only computes the length of a string to the first null byte, your strings may not contain embedded 0’s. To create a StringArray_t with strings that contain embedded 0’s, use `GAUSS__StringArrayL`.

  If `sa` is NULL, there was insufficient memory to malloc space for the string array and its descriptor.

  Use this function to create a string array descriptor that you can use in the following functions:
  ```c
  GAUSS__CopyStringArrayToArg
  GAUSS__CopyStringArrayToGlobal
  GAUSS__MoveStringArrayToArg
  GAUSS__MoveStringArrayToGlobal
  ```

  Free the StringArray_t with `GAUSS__FreeStringArray`.

- **See also**
  - `GAUSS__StringArrayL`  `GAUSS__FreeStringArray`
8. **C API: REFERENCE**

---

**Purpose**

Creates a `StringArray_t` with strings of user-specified length and copies the string array data.

**Format**

```c
StringArray_t *GAUSS_StringArrayL( unsigned int rows, unsigned int cols, char **strs, size_t *lens );
```

```c
sa = GAUSS_StringArrayL( rows, cols, strs, lens );
```

**Input**

- `rows` number of rows.
- `cols` number of columns.
- `strs` pointer to an array of character pointers containing the strings of the array.
- `lens` pointer to an array of `size_t`'s containing the length of each string, including null terminator.

**Output**

- `sa` pointer to a string array descriptor.

**Remarks**

`GAUSS_StringArrayL` malloc's a `StringArray_t` and fills it in with your input information. It makes a copy of all the strings in the array and creates an array of `rows*cols StringElement_t`'s. The `table` member of the `StringArray_t` is set to the address of the array of `StringElement_t`'s.

This function takes the length of the strings from the `lens` argument rather than calling `strlen`, which computes the length of a string only to the first null byte. This allows your strings to contain embedded 0's. If your strings do not contain embedded 0's, you can use `GAUSS_StringArray` to create your `StringArray_t`.

If `sa` is NULL, there was insufficient memory to malloc space for the string array and its descriptor.

Use this function to create a string array descriptor that you can use in the following functions:

```c
GAUSS_CopyStringArrayToArg
GAUSS_CopyStringArrayToGlobal
GAUSS_MoveStringArrayToArg
GAUSS_MoveStringArrayToGlobal
```

You can free the `StringArray_t` with `GAUSS_FreeStringArray`.

**See also**

`GAUSS_StringArray`, `GAUSS_FreeStringArray`
**GAUSS\_StringLength**

- **Purpose**
  Creates a `String\_t` with string of user-specified length and copies the string data.

- **Format**
  
  ```c
  String\_t \*GAUSS\_StringLength( char \*str, int len );
  ```
  
  ```c
  strdesc = GAUSS\_StringLength( str, len );
  ```

- **Input**
  
  - `str` pointer to string.
  - `len` length of string, including null terminator.

- **Output**
  
  - `strdesc` pointer to a string descriptor.

- **Remarks**
  
  The `GAUSS\_StringLength` function allocates a `String\_t` and fills it in with your input information. It makes a copy of the string and sets the `sdata` member of the `String\_t` to point to the copy. To create a `String\_t` for your string without making a copy of it, use `GAUSS\_StringAliasL`.

  This function takes the length of the string from the `len` argument rather than calling `strlen`, which computes the length of a string only to the first null byte. This allows your string to contain embedded 0’s. If your string does not contain embedded 0’s, you can use `GAUSS\_String` to create your `String\_t`.

  If `strdesc` is NULL, there was insufficient memory to `malloc` space for the string and its descriptor.

  Use this function to create a string descriptor that you can use in the following functions:

  ```c
  GAUSS\_CopyStringToArg
  GAUSS\_CopyStringToGlobal
  GAUSS\_MoveStringToArg
  GAUSS\_MoveStringToGlobal
  ```

  Free the `String\_t` with `GAUSS\_FreeString`.

- **See also**
  
  `GAUSS\_String`, `GAUSS\_StringAliasL`, `GAUSS\_StringAlias`, `GAUSS\_FreeString`


8. C API: REFERENCE

GAUSS_TranslateDataloopFile

- **Purpose**
  Translates a dataloop file.

- **Format**
  ```c
  int GAUSS_TranslateDataloopFile( char *transfile, char *srcfile );
  
  errs = GAUSS_TranslateDataloopFile( transfile, srcfile );
  ```

- **Input**
  
  - `transfile` pointer to name of translated file.
  - `srcfile` pointer to name of source file.

- **Output**
  
  - `errs` number of errors.

- **Remarks**
  
  `GAUSS_TranslateDataloopFile` translates a file that contains a dataloop, so it can be read by the compiler. After translating a file, you can compile it with `GAUSS_CompileFile` and then run it with `GAUSS_Execute`.

  If you want to see any errors that `GAUSS_TranslateDataloopFile` encounters, then you must call `GAUSS_HookProgramErrorOutput` before calling `GAUSS_TranslateDataloopFile`.

- **See also**
  
  - `GAUSS_CompileFile`
  - `GAUSS_Execute`
GAUSS_TranslateDataloopFile
Chapter 9

Structure Reference
GAUSS_MatrixInfo_t

Purpose

2-dimensional matrix info descriptor structure.

Format

A GAUSS_MatrixInfo_t is a structure with the following members:

```c
unsigned int  rows;
unsigned int  cols;
int            complex;
double *       maddr;
```

- `rows`  number of rows.
- `cols`  number of columns.
- `complex`  0 if matrix is real, 1 if complex.
- `maddr`  pointer to matrix.

Remarks

GAUSS_MatrixInfo_t structures are used only with GAUSS_GetMatrixInfo. A GAUSS_MatrixInfo_t gives you a pointer to the actual data of a matrix in a GAUSS workspace. Therefore, any changes you make to the matrix after getting it will be reflected in the GAUSS workspace. The matrix data of a GAUSS_MatrixInfo_t still belongs to GAUSS, and GAUSS will free it when necessary. You should not attempt to free a matrix indicated by a GAUSS_MatrixInfo_t.

The matrix is always stored in row-major order in memory. If the matrix is complex, it will be stored in memory with the entire real part first, followed by the imaginary part.

See also

GAUSS_GetMatrixInfo, Matrix_t
9. STRUCTURE REFERENCE

Matrix_t

- **Purpose**

  2-dimensional matrix descriptor structure.

- **Format**

  A `Matrix_t` is a structure with the following members:

  ```
  double * mdata;
  int rows;
  int cols;
  int complex;
  ```

  - `mdata` : pointer to matrix.
  - `rows` : number of rows.
  - `cols` : number of columns.
  - `complex` : 0 if matrix is real, 1 if complex.

- **Remarks**

  A `Matrix_t` is used to hold the information for a matrix. To create a `Matrix_t`, use one of the following functions:

  ```
  GAUSS_ComplexMatrix
  GAUSS_ComplexMatrixAlias
  GAUSS_Matrix
  GAUSS_MatrixAlias
  ```

  The matrix data of a `Matrix_t` are always stored in row-major order in memory. If the matrix is complex, it will be stored with the entire real part first, followed by the imaginary part.

  Use `GAUSS_FreeMatrix` to free a `Matrix_t`.

- **See also**

  ```
  GAUSS_Matrix, GAUSS_MatrixAlias, GAUSS_ComplexMatrix,
  GAUSS_ComplexMatrixAlias, GAUSS_FreeMatrix, GAUSS_MatrixInfo_t
  ```
String_t

■ Purpose

String descriptor structure.

■ Format

A String_t is a structure with the following members:

```
char * stdata;
size_t length;
```

- `stdata` pointer to string.
- `length` length of string, including null terminator.

■ Remarks

A String_t is used to hold the information for a string. To create a String_t, use one of the following functions:

```
GAUSS_String
GAUSS_StringAlias
GAUSS_StringAliasL
GAUSS_StringL
```

GAUSS strings are null-terminated, but they can also contain embedded 0’s. Therefore, you can’t rely on strlen to determine the length of a string; it must be explicitly stated. For this reason, the engine returns strings using a String_t structure rather than the simpler char pointer.

Use GAUSS_FreeString to free a String_t.

■ See also

GAUSS_String, GAUSS_StringAlias, GAUSS_StringL, GAUSS_StringAliasL, GAUSS_FreeString
9. STRUCTURE REFERENCE

- **Purpose**

String array descriptor structure.

- **Format**

A `StringArray_t` is a structure with the following members:

```c
StringElement_t * table;
unsigned int rows;
unsigned int cols;
size_t baseoffset;
```

- **Remarks**

A `StringArray_t` is used to hold the information for a string array. To create a `StringArray_t`, use one of the following functions:

```
GAUSS_StringAlias
GAUSS_StringAliasL
```

A `StringArray_t` contains a pointer to an array of `StringElement_t`'s, one for each string in the array.

The engine returns string arrays using `StringArray_t` and `StringElement_t` structures rather than the simpler `char *` array. The reason for this is that even though `GAUSS` strings are null-terminated, they can also contain embedded 0's. Therefore, you cannot rely on `strlen` to determine the length of a string; it must be explicitly stated.

Use `GAUSS_FreeStringArray` to free a `StringArray_t`.

- **See also**

`GAUSS_StringArray`, `GAUSS_StringArrayL`, `GAUSS_FreeStringArray`, `StringElement_t`
**StringElement**

- **Purpose**

  String descriptor structure used for strings in a string array.

- **Format**

  A `StringElement_t` is a structure with the following members:

  ```
  size_t offset;
  size_t length;
  ```

  - `offset` : offset of string.
  - `length` : length of string.

- **Remarks**

  A `StringElement_t` is used to hold the information for a string in a string array. The `table` member of a `StringArray_t` points at an array of `rows*cols StringElement_t`'s. The array of `StringElement_t`'s is followed in memory by the array of strings. The `baseoffset` member of a `StringArray_t` is the offset of the array of strings from `table`.

  ```
  baseoffset = rows*cols*sizeof( StringElement_t )
  ```

  The address of the string `[r,c]` in a `StringArray_t` can be computed as follows, assuming `r` and `c` are base 1 indices as in GAUSS:

  ```
  StringArray_t *sa;
  StringElement_t *se;
  char *str;

  sa = GAUSS_GetStringArray( wh, "gsa" );
  se = sa->table + ( r-1 )*sa->cols + c-1;
  str = ( char * )( sa->table ) + sa->baseoffset + se->offset;
  ```

- **See also**

  `StringArray_t`, `GAUSS_StringArray`, `GAUSS_StringArrayL`, `GAUSS_FreeStringArray`
Index

GAUSS__AssignFreeableMatrix, 34
GAUSS__CallProc, 36
GAUSS__CallProcFreeArgs, 39
GAUSS__CompileExpression, 42
GAUSS__CompileFile, 44
GAUSS__CompileString, 46
GAUSS__CompileStringAsFile, 48
GAUSS__ComplexMatrix, 50
GAUSS__ComplexMatrixAlias, 52
GAUSS__CopyArgToArg, 54
GAUSS__CopyArgToMatrix, 57
GAUSS__CopyArgToString, 59
GAUSS__CopyArgToStringArray, 61
GAUSS__CopyGlobal, 63
GAUSS__CopyMatrixToArg, 65
GAUSS__CopyMatrixToGlobal, 67
GAUSS__CopyStringArrayToArg, 69
GAUSS__CopyStringArrayToGlobal, 71
GAUSS__CopyStringToArg, 73
GAUSS__CopyStringToGlobal, 75
GAUSS__CreateArgList, 77
GAUSS__CreateProgram, 79
GAUSS__CreateWorkspace, 80
GAUSS__DeleteArg, 81
GAUSS__ErrorText, 83
GAUSS__Execute, 84
GAUSS__ExecuteExpression, 86
GAUSS__FreeArgList, 88
GAUSS__FreeMatrix, 90
GAUSS__FreeProgram, 92
GAUSS__FreeString, 94
GAUSS__FreeStringArray, 95
GAUSS__FreeWorkspace, 96
GAUSS__GetArgType, 98
GAUSS__GetDouble, 101
GAUSS__GetError, 103
GAUSS__GetHome, 104
GAUSS__GetLogFile, 105
GAUSS__GetLogStream, 106
GAUSS__GetMatrix, 107
GAUSS__GetMatrixAndClear, 109
GAUSS__GetMatrixInfo, 111
GAUSS__GetString, 113
GAUSS__GetStringArray, 115
GAUSS__GetSymbolType, 117
GAUSS__GetWorkspaceName, 119
GAUSS__HookFlushProgramOutput, 120
GAUSS__HookGetCursorPosition, 121
GAUSS__HookProgramErrorOutput, 122
GAUSS__HookProgramInputChar, 123
GAUSS__HookProgramInputCharBlocking, 124
GAUSS__HookProgramInputCheck, 125
GAUSS__HookProgramInputString, 126
GAUSS__HookProgramOutput, 127
GAUSS__Initialize, 128
GAUSS__InsertArg, 129
GAUSS__LoadCompiledBuffer, 130
GAUSS__LoadCompiledFile, 131
GAUSS__LoadWorkspace, 133
GAUSS__Matrix, 134
GAUSS__MatrixAlias, 136
GAUSS__MatrixInfo_t, 172
GAUSS__MoveArgToArg, 138
GAUSS__MoveArgToMatrix, 141
GAUSS__MoveArgToString, 143
INDEX

GAUSS_MoveArgToStringArray, 145
GAUSS_MoveMatrixToArg, 147
GAUSS_MoveMatrixToGlobal, 148
GAUSS_MoveStringArrayToArg, 149
GAUSS_MoveStringArrayToGlobal, 150
GAUSS_MoveStringToArg, 151
GAUSS_MoveStringToGlobal, 152
GAUSS_PutDouble, 153
GAUSS_SaveProgram, 154
GAUSS_SaveWorkspace, 155
GAUSS_SetError, 156
GAUSS_SetHome, 157
GAUSS_SetHomeVar, 158
GAUSS_SetLogFile, 159
GAUSS_SetLogStream, 160
GAUSS_SetWorkspaceName, 161
GAUSS_Shutdown, 162
GAUSS_String, 163
GAUSS_StringAlias, 164
GAUSS_StringAliasL, 165
GAUSS_StringArray, 166
GAUSS_StringArrayL, 167
GAUSS_StringL, 168
GAUSS_TranslateDataloopFile, 169
GRTE, 3, 5

W

workspace, 11
workspace handle, 11

M

Matrix_t, 173

P

POSIX Threads, 4
procedures, 16
program handle, 11

R

readonly, 17, 18, 27
Run-Time Engine, 3

S

String_t, 174
StringArray_t, 175
StringElement_t, 176

178