# GAUSS Engine Manual

## *Programmer's Manual*

007578
Version 13
Documentation November 1, 2012

# Contents

# 1 Installation

## 1.1 Installation Under Linux/Mac

### 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/mteng13`.

```
mkdir /usr/local/mteng13
```

Go to that directory.

```
cd /usr/local/mteng13
```

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 MTENGHOME13 that points to the installation directory.

C shell

```
setenv MTENGHOME13 /usr/local/mteng13
```

Korn, Bourne shell

```
MTENGHOME13=/usr/local/mteng13
export MTENGHOME13
```

The engine looks in $MTENGHOME13 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 MTENGHOME13 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

Execute **`rlmhostid`** in the `lm` 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. From a command prompt window, cd to the lm directory and type rlmhostid ¿ hostid.txt. Email the content of hostid.txt to *license@Aptech.com*. You will be sent a license and instructions for its 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 × rows × 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`**, the graphical user interface, **`gauss`**, 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

### 1.2.1 Installing the Files

#### From CD

Insert the CD into a CD drive, and setup should start automatically. If setup does not start automatically, browse to the CD-ROM drive drive and double-click on the `*.msi` file to launch the installer. Follow the prompts to select 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. Browse the folder where you saved the `.zip` file and extract it, then double-click on the `*.msi` file to launch the installer. Follow the prompts to select a directory to install to and copy the **GAUSS Engine** files to your hard disk.

### 1.2.2 Configuring the Environment

The **GAUSS Engine** examples require an environment variable called MTENGHOME13 that points to the installation directory.

### 1.2.3 Licensing

Execute `linfo.exe` in the `res` 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 × rows × columns bytes.

### 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.

# 2 Sample Programs

At least five sample programs are provided in the **GAUSS Engine** installation directory: `eng2d.c`, `mtexpr.c`, `mtcall.c`, `grte01.c`, and `grte02.c`.

The examples that start with **grte** will run with the **GAUSS 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.

For C++ examples outlining the necessary procedure for creating and distributing an application using the **GAUSS Run-Time Engine**, see Aptech's ftp site.

## 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/mteng11
```

### 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

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++ 7.0. 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 `c:\mteng11`.

```
c:    cd \mteng11
```

### eng2d

Run **nmake** to build `eng2d`.

```
nmake 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.

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

# 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 5.

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

## 3.1 Setup and Initialization

### 3.1.1 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.

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

| Functions Hooked By | Are Called By |
|---|---|
| **GAUSS_HookProgramInputChar** | **key** |
| **GAUSS_HookProgramInputCharBlocking** | **keyw**, **show** |
| **GAUSS_HookProgramInputCheck** | **keyav** |
| **GAUSS_HookProgramInputString** | **con**, **cons** |

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.

| | |
|---|---|
| **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.

```
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 10×10 matrix of random numbers and *xi* is its inverse.

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

```
Matrix_t *mat;

mat = GAUSS_GetMatrix( w1, "xi" );
```

The following code copies the retrieved matrix to another workspace as *xinv*.

```
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.

```
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 11 for detailed information on the structures.

| | |
|---|---|
| **Array_t** | N-dimensional array, real or complex |
| **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*.

```
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*.

```
Matrix_t *matalias;

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

The call to **GAUSS_MatrixAlias** calls **malloc** only 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*.

```
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.

### 3.2.4 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.

```
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.

```
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.

```
WorkspaceHandle_t *w1;
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" );
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 1000000×100 real matrix is created and placed in a workspace.

```
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.

```
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.

```
Array_t *arr;
Matrix_t *mat;
StringArray_t *sa;
String_t *st;
int type;

arr = NULL;
mat = NULL;
sa =  NULL;
st =  NULL;

type = GAUSS_GetSymbolType( w1, "xi" );

switch( type )
{
    case GAUSS_ARRAY:
        arr = GAUSS_GetArray( w1, "xi" );
        break;

    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_Initialize` again after calling `GAUSS_Shutdown`.

# Using the GRTE 4



The **GAUSS Run-Time Engine** (**GRTE**) allows you to create distributable applications that take advantage of the computational speed and power of **GAUSS**.

## 4.1 Creating a Distributable Application

To use the **GAUSS Engine** in an application on Windows, you must link the application with `mteng.lib`, and your application directory must contain `mteng.dll`. On Linux, link the application with `-lmteng` and make sure that `libmteng.so` is in your application directory. On both platforms, your application will run only if a valid license file with the MTENG feature can be found in your application directory. This license is linked to a particular hostid, so it will run only on your development machine.

To create a distributable application, you must use the **GAUSS Run-Time Engine**. To use the **GAUSS Run-Time Engine** on Windows, link your application with `mtengrt.lib` (instead of `mteng.lib`) and distribute `mtengrt.dll` with your application. On Linux, you must link your application with `-lmtengrt` (instead of `-lmteng`), and distribute `libmtengrt.so` with your application. For the application to run, you must also distribute a license file with the **MTGRTE** feature, located in your application directory with a `g.gkf` file name.

Applications that use the **GAUSS Run-Time Engine** will not be able to create globals in a **GAUSS** workspace. Therefore, any global variables or procedures that are needed by the application must be compiled with the **GAUSS Engine** into a `.gcg` file that is distributed with the application. You may use either the **`compile`** command from the command line interface, **`engauss`**, or the **`GC`** compiler to compile a **GAUSS** program containing global declarations.

### 4.1.1 List of Files To Distribute

There are several files which must be distributed with your application in order for the application to be able to use the **GAUSS Run-Time Engine**. The list of files differs between platforms.

#### Windows

On Windows, the necessary files are:

1. The `.gcg` file containing compiled declarations of all global variables and procedures needed by the application.
2. The shared library files `cmx20.dll`, `compobj.dll`, `gauss.dll`, `libiomp5md.dll`, `import.dll`, `mtengrt.dll`, `opnx32.dll`, `pthreadVC.dll`, and `xls.dll`.
3. The **GAUSS** configuration file, `gauss.cfg`. The distributed copy of `gauss.cfg` must have both the *user_lib* and *gauss_lib* options set to **`off`**. By default, they are both set to **`on`**.
4. A license file with the **`MTGRTE`** feature, which must have a `g.gkf` file name and be located in the directory containing the shared library and configuration files.

#### Linux

On Linux, the necessary files are:

1. The `.gcg` file containing compiled declarations of all global variables and procedures needed by the application.

2. The shared library files `libgauss.so` and `libmtengrt.so`.
3. The **GAUSS** configuration file, `gauss.cfg`. The distributed copy of `gauss.cfg` must have both the `user_lib` and `gauss_lib` options set to **`off`**. By default, they are both set to **`on`**.
4. A license file with the **`MTGRTE`** feature, which must have a `g.gkf` file name and be located in the directory containing the shared library and configuration files.

### 4.1.2 Setting the Home Directory

Before the end user is able to run the application, the home path for the **GAUSS Run-Time Engine** must be set, so it can find the shared library, configuration, and license files. There are three ways to do this:

1. The end user can set the environment variable **`MTENGHOME13`** to the path of the directory containing the shared library and configuration files.
2. You can specify the name of a new home environment variable in your application with **`GAUSS_SetHomeVar`**. The end user would then need to set that environment variable to the path of the directory containing the shared library and configuration files.
3. You can include code in your application that will find the correct path and set it using **`GAUSS_SetHome`**.

On Linux, the path of the directory containing the shared library files must also be included in the environment variable LD_LIBRARY_PATH, or the shared library files must be placed in the canonical system location.

## 4.2 The `grte01` and `grte02` Executables

The **GAUSS Run-Time Engine** is shipped with two complete examples demonstrating how you may create and distribute an application that uses the functionality of **GAUSS**. These examples can be found in the main directory of your **GAUSS Engine** installation directory. The main directory contains a `README` file, which gives instructions on bulding and running the examples.

### 4.2.1 Building the Executable

The grte01 and grte02 examples are made up of five files:

1. grte01.c - the example application code for grte01.
2. grte02.c - the example application code for grte02.
3. grte01.gau - the **GAUSS** program file containing declarations of all of the global variables and procedures that are used in grte01.
4. grte02.gau - the **GAUSS** program file containing declarations of all of the global variables and procedures that are used in grte02.
5. Makefile - the Makefile needed to build the grte01 and grte02 executables.

To build the application, you must **make** the grte01 and grte02 executables and compile grte01.gau and grte02.gau into grte01.gcg and grte02.gcg respectively. You may use either the **compile** command from the command line interface, **engauss**, or the GUI version, **gauss**, or the **GC** compiler to compile the .gau file.

### 4.2.2 Including the Necessary Files

After building the example applications, you should create a directory named distribute and copy all of the files needed to run the application into distribute as follows:

#### Windows

1. Copy or move grte01.exe, grte02.exe, grte02.gcg, and grte01.gcg into distribute.
2. Copy the shared library files cmx20.dll, compobj.dll, gauss.dll, import.dll, mtengrt.dll, opnx32.dll, pthreadVC.dll, and xls.dll, as well as the **GAUSS** configuration file, gauss.cfg, from your **GAUSS Engine** installation directory into distribute. Then set both the *user_lib* and *gauss_lib* options in the distribute copy of gauss.cfg to **off**.

3. Copy your **GAUSS Run-Time Engine** license file (which should be called `g.gkf`) into the `distribute` directory.

#### Linux

1. Copy or move `grte01.exe`, `grte02.exe`, `grte02.gcg`, and `grte01.gcg` into `distribute`.
2. Copy the shared library files `libgauss.so` and `libmtengrt.so`, as well as the **GAUSS** configuration file, `gauss.cfg`, from your **GAUSS Engine** installation directory into `distribute`. Then set both the *user_lib* and *gauss_lib* options in the `distribute` copy of `gauss.cfg` to **`off`**.
3. Copy your **GAUSS Run-Time Engine** license file (which should be called `g.gkf`) into the `distribute` directory.

### 4.2.3 Running the Executable

After copying the files as specified above, the `distribute` directory should contain all of the files needed to run the `grte01` and `grte02` executables. This will allow the example to run if it is moved to another location.

# 5 Multi-threaded Applications

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.

## 5.1 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 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.

## 5.2 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.

### 5.2.1 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 be used in a thread-safe fashion (both flags true) in subsequent compiles and executes in the same workspace.

## 5.3 Calling GAUSS Procedures

The functions **GAUSS_CallProc** and **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 **GAUSS_CompileFile**, **GAUSS_CompileString** or **GAUSS_CreateProgram**. If the program handle is created with **readonlyE** true, then **GAUSS_CallProc** and **GAUSS_CallProcFreeArgs** read lock the workspace, otherwise they use a write lock.

### 5.3.1 Assuring Concurrency

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

```
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.

# Using the Command Line Interface

# 6

**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. |

## 6.1 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**.

## 6.2 Interactive Commands

### 6.2.1 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
```

### 6.2.2 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*

### 6.2.3 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.

### 6.2.4 run

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

The format for **run**:

**run** *filename*

### 6.2.5 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*

### 6.2.6 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:

```
config
```

#### Run Menu

| | |
|---|---|
| **Translator** | Toggles on/off the translation of a file using **dataloop**. The translator is not necessary for **GAUSS** program files not using **dataloop**. |
| **Translator line number tracking** | Toggles on/off execution time line number tracking of the original file before translation. |
| **Line number tracking** | Toggles on/off the execution time line number tracking. If the translator is on, the line numbers refer to the translated file. |

#### Compile Menu

| | |
|---|---|
| **Autoload** | Toggles on/off the autoloader. |
| **Autodelete** | Toggles on/off autodelete. |
| **GAUSS Library** | Toggles on/off the **GAUSS** library functions. |
| **User Library** | Toggles on/off the user library functions. |

| | |
|---|---|
| **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. |

## 6.3 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:

| | |
|---|---|
| *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 indices of the upper left corner of the block 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. |

### 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.

# 7 GAUSS Utilities

There are several **GAUSS** utilities that are included with the **GAUSS Engine**. The **GAUSS** Profiler utilities include the collector tool, **encollect** (the **GAUSS Engine** equivalent to **tcollect**), and the **GAUSS** Profiler analysis tool, **gaussprof**. Also included are **ATOG**, a conversion utility that converts ASCII files into **GAUSS** data sets, and **vwr** or **vwrmp** depending upon your platform (Windows, Linux, etc.).

The **GAUSS** User's Guide and/or accompanying README files in the **GAUSS Engine** home directory provide details on how to use these tools. A short list of options and syntax is also often available by starting the utility without any options or by typing `utility -help` at a command prompt.

Note: in all cases, these standalone utilities are not run from within the **GAUSS Engine** but rather from a command prompt window. You can easily go to a command prompt from a **GAUSS Engine** prompt by typing **dos**. This will take you to a command prompt in your current working directory.

# 8 The GC Compiler

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.

# C API: Overview 9



## 9.1 Functions

### 9.1.1 Pre-initialization setup

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

| | |
|---|---|
| GAUSS_GetHome | Gets the engine home path. |
| GAUSS_GetHomeVar | Gets the name of the environment variable containing the home path. |
| GAUSS_HookProgramErrorOutput | Sets the callback function for program error output. |
| GAUSS_HookProgramInputChar | Sets callback function for **key** function. |
| GAUSS_HookProgramInputCharBlocking | Sets callback function for **keyw** and **show** functions. |
| GAUSS_HookProgramInputCheck | Sets callback function for **keyav** function. |

| | |
|---|---|
| GAUSS_HookProgramInputString | Sets callback function for **con** and **cons** functions. |
| GAUSS_HookProgramOutput | Sets the callback function for normal program output. |
| GAUSS_SetHome | Sets the engine home path directly. |
| GAUSS_SetHomeVar | Sets the name of an environment variable containing the home path. |
| GAUSS_SetLogFile | Sets the file name and path for logging system errors. |
| GAUSS_SetLogStream | Sets the file pointer for logging system errors. |

### 9.1.2 Initialization and Shutdown

| | |
|---|---|
| GAUSS_Initialize | Initializes the engine. Call at the beginning of your application, after setup functions. |
| GAUSS_Shutdown | Shuts the engine down. Call prior to ending your application. |

### 9.1.3 Compiling and Executing GAUSS programs

| | |
|---|---|
| GAUSS_CompileExpression | Compiles a right-hand side expression. |
| GAUSS_CompileFile | Compiles a file containing **GAUSS** code. |
| GAUSS_CompileString | Compiles a character string containing **GAUSS** code. |
| GAUSS_CompileStringAsFile | Compiles a character string containing **GAUSS** code as a file. |
| GAUSS_CreateWorkspace | Creates a workspace handle. |
| GAUSS_Execute | Executes a program. |
| GAUSS_ExecuteExpression | Executes a right-hand side expression. |

| | |
|---|---|
| GAUSS_FreeProgram | Frees a program handle created in a compile. |
| GAUSS_FreeWorkspace | Frees a workspace handle. |
| GAUSS_LoadCompiledBuffer | Loads a compiled program from a buffer. |
| GAUSS_LoadCompiledFile | Loads a compiled program from a file. |
| GAUSS_LoadWorkspace | Loads workspace information saved in a file. |
| GAUSS_SaveProgram | Saves a compiled program as a file. |
| GAUSS_SaveWorkspace | Saves workspace information in a file. |
| GAUSS_TranslateDataloopFile | Translates a dataloop file. |

### 9.1.4 Calling Procedures

| | |
|---|---|
| GAUSS_CallProc | Calls a procedure |
| GAUSS_CallProcFreeArgs | Calls a procedure and frees its arguments. |
| GAUSS_CopyArgToArg | Copies an argument from one argument list to another. |
| GAUSS_CopyArgToArray | Copies an array from an argument list descriptor to an array descriptor. |
| GAUSS_CopyArgToMatrix | Copies a matrix from an argument list descriptor to a matrix descriptor. |
| GAUSS_CopyArgToString | Copies a string from an argument list descriptor to a string descriptor. |
| GAUSS_CopyArgToStringArray | Copies a string array from an argument list descriptor to a string array descriptor. |
| GAUSS_CopyArrayToArg | Copies an array to an argument list descriptor. |
| GAUSS_CopyMatrixToArg | Copies a matrix to an argument list descriptor. |
| GAUSS_CopyStringArrayToArg | Copies a string array to an argument list descriptor. |
| GAUSS_CopyStringToArg | Copies a string to an argument list descriptor. |
| GAUSS_CreateArgList | Creates an empty argument list descriptor. |
| GAUSS_CreateProgram | Creates a program handle to use when calling a procedure. |
| GAUSS_DeleteArg | Deletes an argument from an argument list descriptor. |

| | |
|---|---|
| GAUSS_FreeArgList | Frees an argument list descriptor. |
| GAUSS_GetArgType | Gets the type of an argument in an argument list descriptor. |
| GAUSS_InsertArg | Inserts an argument in an argument list descriptor. |
| GAUSS_MoveArgToArg | Moves an argument from one argument list to another. |
| GAUSS_MoveArgToArray | Moves an array from an argument list descriptor to an array descriptor. |
| GAUSS_MoveArgToMatrix | Moves a matrix from an argument list descriptor to a matrix descriptor. |
| GAUSS_MoveArgToString | Moves a string from an argument list descriptor to a string descriptor. |
| GAUSS_MoveArgToStringArray | Moves a string array from an argument list descriptor to a string array descriptor. |
| GAUSS_MoveArrayToArg | Moves an array to an argument list descriptor. |
| GAUSS_MoveMatrixToArg | Moves a matrix to an argument list descriptor. |
| GAUSS_MoveStringArrayToArg | Moves a string array to an argument list descriptor. |
| GAUSS_MoveStringToArg | Moves a string to an argument list descriptor. |

### 9.1.5 Creating and Freeing GAUSS Format Data

| | |
|---|---|
| GAUSS_ComplexArray | Creates an array descriptor for a complex array and copies the array. |
| GAUSS_ComplexArrayAlias | Creates an array descriptor for a complex array. |
| GAUSS_ComplexMatrix | Creates a matrix descriptor for a complex matrix and copies the matrix. |
| GAUSS_ComplexMatrixAlias | Creates a matrix descriptor for a complex matrix. |
| GAUSS_FreeArray | Frees an array descriptor. |
| GAUSS_FreeMatrix | Frees a matrix descriptor. |
| GAUSS_FreeString | Frees a string descriptor. |
| GAUSS_FreeStringArray | Frees a string array descriptor. |

| | |
|---|---|
| GAUSS_Array | Creates an array descriptor and copies array. |
| GAUSS_ArrayAlias | Creates an array descriptor. |
| GAUSS_Matrix | Creates a matrix descriptor and copies matrix. |
| GAUSS_MatrixAlias | Creates a matrix descriptor. |
| GAUSS_String | Creates a string descriptor and copies the string. |
| GAUSS_StringAlias | Creates a string descriptor. |
| GAUSS_StringAliasL | Creates a string descriptor for a string of user-specified length. |
| GAUSS_StringArray | Creates a string array descriptor and copies the string array. |
| GAUSS_StringArrayL | Creates a string array descriptor for strings of user-specified length and copies the string array. |
| GAUSS_StringL | Creates a string descriptor for string of user-specified length and copies the string. |



### 9.1.6 Moving Data Between GAUSS and Your Application

| | |
|---|---|
| GAUSS_AssignFreeableArray | Assigns **malloc**'d data to a global array. |
| GAUSS_AssignFreeableMatrix | Assigns **malloc**'d data to a global matrix. |
| GAUSS_CopyGlobal | Copies a symbol from one workspace to another. |
| GAUSS_CopyArrayToGlobal | Copies an array to **GAUSS**. |
| GAUSS_CopyMatrixToGlobal | Copies a matrix to **GAUSS**. |
| GAUSS_CopyStringToGlobal | Copies a string to **GAUSS**. |
| GAUSS_CopyStringArrayToGlobal | Copies a string array to **GAUSS**. |
| GAUSS_GetDouble | Gets a double from a **GAUSS** global. |
| GAUSS_GetArray | Gets an array from a **GAUSS** global. |
| GAUSS_GetArrayAndClear | Gets an array from a **GAUSS** global and clears the global. |
| GAUSS_GetMatrix | Gets a matrix from a **GAUSS** global. |
| GAUSS_GetMatrixAndClear | Gets a matrix from a **GAUSS** global and clears the global. |
| GAUSS_GetMatrixInfo | Gets information for a matrix in a **GAUSS** global. |
| GAUSS_GetString | Gets a string from a **GAUSS** global. |

| | |
|---|---|
| GAUSS_GetStringArray | Gets a string array from a **GAUSS** global. |
| GAUSS_GetSymbolType | Gets the type of a symbol in a **GAUSS** global. |
| GAUSS_MoveArrayToGlobal | Moves an array to **GAUSS** and frees the descriptor. |
| GAUSS_MoveMatrixToGlobal | Moves a matrix to **GAUSS** and frees the descriptor. |
| GAUSS_MoveStringToGlobal | Moves a string to **GAUSS** and frees the descriptor. |
| GAUSS_MoveStringArrayToGlobal | Moves a string array to **GAUSS** and frees the descriptor. |
| GAUSS_PutDouble | Puts a double into **GAUSS**. |

### 9.1.7 GAUSS Engine Error Handling

| | |
|---|---|
| GAUSS_ClearInterrupt | Clears a program interrupt request. |
| GAUSS_CheckInterrupt | Checks for a program interrupt request. |
| GAUSS_ErrorText | Gets the text for an error number. |
| GAUSS_GetError | Gets the stored error number. |
| GAUSS_GetLogFile | Gets the current error log file. |
| GAUSS_GetLogStream | Gets the current error log stream. |
| GAUSS_SetError | Sets the stored error number. |
| GAUSS_SetInterrupt | Sets a program interrupt request. |

## 9.2 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.

# C API: Reference 10



## GAUSS_Array

PURPOSE Creates an **Array_t** for a real array and copies the array data.

FORMAT **Array_t *GAUSS_Array( size_t** *dims*, **double *** *orders*, **double *** *addr* **);**

*arr* **= GAUSS_Array(** *dims*, *orders*, *addr* **);**

INPUT *dims* number of dimensions.

*orders* vector of orders.

*addr* pointer to array.

OUTPUT *mat* pointer to an array descriptor.

REMARKS

**GAUSS_Array** **malloc**'s an **Array_t** and fills it in with your input information. It makes a copy of the array and sets the **adata** member of the **Array_t** to point to the copy. **GAUSS_Array** should only be used for real arrays. To create an **Array_t** for a complex array, use **GAUSS_ComplexArray**. To create an **Array_t** for a real array without making a copy of the array, use **GAUSS_ArrayAlias**.

Set **orders** to NULL if the vector of orders of the array is located at the beginning of the block of memory that contains the array data. In this case, **addr** should point to the vector of orders, followed by the array data. Otherwise, set **orders** to point to the block of memory that contains vector of orders. The vector of orders should contain **dims** doubles.

To create an **Array_t** for an empty array, set **dims** to 0 and **addr** to NULL.

If **arr** is NULL, there was insufficient memory to **malloc** space for the array and its descriptor.

Use this function to create an array descriptor that you can use in the following functions:

**GAUSS_CopyArrayToArg**
**GAUSS_CopyArrayToGlobal**
**GAUSS_MoveArrayToArg**
**GAUSS_MoveArrayToGlobal**

Free the **Array_t** with **GAUSS_FreeArray**.

EXAMPLE

```
int ret;
double orders[3] = { 2.0, 2.0, 3.0 };
double a[2][2][3] = {
             { { 1.0, 2.0, 3.0 }, { 4.0, 5.0, 6.0 } }
             { { 7.0, 8.0, 9.0 }, { 10.0, 11.0, 12.0 } }
                    };

if ( ret = GAUSS_MoveArrayToGlobal(
                  wh,
                  GAUSS_Array( 3, orders, a ),
                  "a"
    ) )
```

```
{
    char buff[100];

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

The above example uses **GAUSS_Array** to copy a local array into an **Array_t** structure, and moves the array into a **GAUSS** workspace. It assumes that *wh* is a pointer to a valid workspace handle.

SEE ALSO **GAUSS_ComplexArray**, **GAUSS_ArrayAlias**, **GAUSS_CopyArrayToGlobal**, **GAUSS_CopyArrayToArg**, **GAUSS_MoveArrayToGlobal**, **GAUSS_MoveArrayToArg**, **GAUSS_FreeArray**

## GAUSS_ArrayAlias



PURPOSE Creates an **Array_t** for a real array.

FORMAT **Array_t *GAUSS_ArrayAlias( size_t** *dims*, **double ***addr* **);**

*arr* = **GAUSS_ArrayAlias(** *dims*, *addr* **);**

INPUT *dims* number of dimensions.

*addr* pointer to matrix.

OUTPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_ArrayAlias** is similar to **GAUSS_Array**; however, it sets the **adata** member of the **Array_t** to point to the array indicated by **addr** instead of making a copy of the array. **GAUSS_ArrayAlias** should only be used for real arrays. For complex arrays, use **GAUSS_ComplexArrayAlias**.

The argument **addr** should point to a **malloc**'d block containing two sections. The first section, which is the vector of orders for the array, contains **dims** doubles. The second section contains the array data. The number of doubles in the section that contains the array data is the product of the elements in the vector of orders. These two sections are laid out contiguously in memory.

If **arr** is NULL, there was insufficient memory to **malloc** space for the array descriptor.

Use this function to create an array descriptor that you can use in the following functions:

**GAUSS_CopyArrayToArg**
**GAUSS_CopyArrayToGlobal**
**GAUSS_MoveArrayToArg**
**GAUSS_MoveArrayToGlobal**

Free the **Array_t** with **GAUSS_FreeArray**. It will not free the array data.

EXAMPLE

```
Array_t *arr;
double *a;
int ret;
size_t dims;

dims = 3;
a = ( double *)malloc( ( 12+dims )*sizeof(double) );

*a = 2.0;
*( a+1 ) = 3.0;
*( a+2 ) = 2.0;
memset( a+dims, 0, 12*sizeof( double ) );

if ( ( arr = GAUSS_ArrayAlias( dims, a ) ) == NULL )
{
    char buff[100];

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

```
if ( ret = GAUSS_MoveArrayToGlobal( wh, arr, "c" ) )
{
    char buff[100];

    printf( "CopyArrayToGlobal failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeArray( arr );
    return -1;
}
```

This example **malloc**'s an array of zeros and then creates an **Array_t** for the array. It moves the array to *wh*, which it assumes to be a pointer to a valid workspace. The array data is freed by **GAUSS** when necessary.

SEE ALSO **GAUSS_Array**, **GAUSS_ComplexArrayAlias**, **GAUSS_CopyArrayToGlobal**, **GAUSS_CopyArrayToArg**, **GAUSS_MoveArrayToGlobal**, **GAUSS_MoveArrayToArg**, **GAUSS_FreeArray**



## GAUSS_AssignFreeableArray

PURPOSE Assigns a **malloc**'d N-dimensional array to a **GAUSS** workspace.

FORMAT **int GAUSS_AssignFreeableArray( WorkspaceHandle_t ***wh**, size_t** *dims***, int** *complex***, double ***address**, char ***name **);**

*ret* **= GAUSS_AssignFreeableArray(** *wh***,** *dims***,** *complex***,** *address***,** *name* **);**

INPUT *wh* pointer to a workspace handle.

*dims* number of dimensions.

*complex* 0 if array is real, 1 if complex.

*address* pointer to array.

*name* pointer to name of array 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_AssignFreeableArray** assigns an array that is created using **malloc** to a **GAUSS** workspace. **GAUSS** takes ownership of the array and frees it when necessary. The data are not moved or reallocated, making this the most efficient way to move a large array to a **GAUSS** workspace. Do not attempt to free an array that has been assigned to **GAUSS** with **GAUSS_AssignFreeableArray**.

The argument *address* should point to a **malloc**'d block containing two sections in the case of a real array or three sections in the case of a complex array. The first section, which is the vector of orders for the array, contains *dims* doubles. The second section contains the real part of the array. The optional third section contains the imaginary part. The number of doubles in the real section is the product of the vector of orders. The number of doubles in the imaginary section is the same as the real section. These three sections are laid out contiguously in memory.

Call **GAUSS_AssignFreeableArray** with a **WorkspaceHandle_t** returned from **GAUSS_CreateWorkspace**.

EXAMPLE
```
int zmat(
          WorkspaceHandle_t *wh,
          char *name,
          size_t dims,
          double *orders
        )
{
    double *fm, *tmp;
    size_t i, nelems;
    int err;

    nelems = 1;
    tmp = orders;

    for ( i=0; i<dims; i++ )
```

```
        nelems *= ( size_t )( *tmp++ );

    fm = malloc( ( nelems+dims )*sizeof( double ) );

    if ( fm == NULL )
    {
        printf( "Malloc failed for fm\n" );
        return -1;
    }

    err = 0;

    memcpy( fm, orders, dims*sizeof( double ) );
    memset( fm+dims, 0, nelems*sizeof( double ) );

    if ( GAUSS_AssignFreeableArray( wh, dims, 0, fm, name ) )
    {
        char buff[100];

        err = GAUSS_GetError();
        printf( "Assign failed for %s: %s\n", name,
            GAUSS_ErrorText( buff, err ) );
        free( fm );
    }

    return err;
}
```

The function above uses **GAUSS_AssignFreeableArray** to create an array of *dims* dimensions, where the size of each dimension is contained in *orders*. The first value in the block of memory indicated by *orders* is the size of the slowest moving dimension of the array, and the last value is the size of the fastest moving dimension. In the example, each value in the array is set to zero, and the array is then assigned to a **GAUSS** workspace. The data are freed if **GAUSS_AssignFreeableArray** fails, otherwise **GAUSS** owns the array and will free it when necessary.

SEE ALSO **GAUSS_CopyArrayToGlobal**, **GAUSS_MoveArrayToGlobal**, **GAUSS_GetArray**

## GAUSS_AssignFreeableMatrix

PURPOSE Assigns a **malloc**'d matrix to a **GAUSS** workspace.

FORMAT
```
int GAUSS_AssignFreeableMatrix( WorkspaceHandle_t *wh, size_t rows,
size_t cols, int complex, double *address, char *name );

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

```
int zmat( WorkspaceHandle_t *wh, char *name, size_t r, size_t c )
{
    double *fm;
    int err;

    fm = malloc( r*c*sizeof( double ) );

    if ( fm == NULL )
    {
        printf( "Malloc failed for fm\n" );
        return -1;
    }

    err = 0;

    memset( fm, 0, r*c*sizeof( double ) );

    if ( GAUSS_AssignFreeableMatrix( wh, r, c, 0, fm, name ) )
    {
        char buff[100];

        err = GAUSS_GetError();
        printf( "Assign failed for %s: %s\n", name,
            GAUSS_ErrorText( buff, err ) );
        free( fm );
    }

    return err;
}
```

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 **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_CopyArrayToArg**
**GAUSS_CopyMatrixToArg**
**GAUSS_CopyStringArrayToArg**
**GAUSS_CopyStringToArg**
**GAUSS_MoveArrayToArg**
**GAUSS_MoveMatrixToArg**
**GAUSS_MoveStringArrayToArg**
**GAUSS_MoveStringToArg**
**GAUSS_PutDoubleInArg**

**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_CopyArgToArray**
**GAUSS_CopyArgToMatrix**
**GAUSS_CopyArgToString**
**GAUSS_CopyArgToStringArray**
**GAUSS_MoveArgToArray**
**GAUSS_MoveArgToMatrix**
**GAUSS_MoveArgToString**
**GAUSS_MoveArgToStringArray**

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

```
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\n",
        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\n",
        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**,

## GAUSS_CallProcFreeArgs

PURPOSE Calls a **GAUSS** procedure and frees the argument list.

FORMAT **ArgList_t *GAUSS_CallProcFreeArgs( ProgramHandle_t *ph, char *procname, ArgList_t *args );**

*rets* **= GAUSS_CallProcFreeArgs(** *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 the argument list structure containing the returns for the procedure.

REMARKS **GAUSS_CallProcFreeArgs** is similar to **GAUSS_CallProc**; however, the **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 **GAUSS_CreateArgList** to create an empty **ArgList_t** structure and the following functions to add arguments to it:

GAUSS_CopyArrayToArg
GAUSS_CopyMatrixToArg
GAUSS_CopyStringArrayToArg
GAUSS_CopyStringToArg
GAUSS_MoveArrayToArg
GAUSS_MoveMatrixToArg
GAUSS_MoveStringArrayToArg
GAUSS_MoveStringToArg
GAUSS_PutDoubleInArg

GAUSS_CallProcFreeArgs returns a pointer to *args*, which has been rewritten with 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_CopyArgToArray
GAUSS_CopyArgToMatrix
GAUSS_CopyArgToString
GAUSS_CopyArgToStringArray
GAUSS_MoveArgToArray
GAUSS_MoveArgToMatrix
GAUSS_MoveArgToString
GAUSS_MoveArgToStringArray

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

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

SEE ALSO **GAUSS_CallProc**, **GAUSS_CreateProgram**, **GAUSS_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_GetArgType**, **GAUSS_InsertArg**, **GAUSS_DeleteArg**

## GAUSS_CheckInterrupt

PURPOSE Checks for a program interrupt request on a thread.

FORMAT **int GAUSS_CheckInterrupt( pthread_t** *tid* **);**

*ret* **= GAUSS_CheckInterrupt(** *tid* **);**

INPUT *tid* thread id of thread to check.

OUTPUT *ret* UTC time of request or 0 if there is no request.

REMARKS If *tid* is 0, the total number of pending interrupts is returned.

Interrupts are checked during certain I/O statements, not every instruction. The **GAUSS** language command **CheckInterrupt** can be used in a **GAUSS** program to check for interrupts and terminate if one is pending.

```
CheckInterrupt;
```

SEE ALSO **GAUSS_SetInterrupt**, **GAUSS_ClearInterrupt**, **GAUSS_ClearInterrupts**

## GAUSS_ClearInterrupts

PURPOSE Clears all program interrupt requests.

FORMAT **int GAUSS_ClearInterrupts( void );**

*ret* **= GAUSS_ClearInterrupts();**

OUTPUT *ret* 0 if successful or 1 if there are no requests found.

REMARKS Normally, this function is not necessary because the compiler and executer clear the requests when they terminate.

SEE ALSO **GAUSS_SetInterrupt**, **GAUSS_CheckInterrupt**, **GAUSS_ClearInterrupt**

## GAUSS_ClearInterrupt

PURPOSE Clears a program interrupt request.

FORMAT **int GAUSS_ClearInterrupt( pthread_t** *tid* **);**

*ret* **= GAUSS_ClearInterrupt(** *tid* **);**

INPUT *tid* thread id of thread.

OUTPUT *ret* 0 if successful or 1 if there is no request found.

REMARKS On most platforms, if *tid* is 0, all interrupt requests are cleared. On Windows, a **pthread_t** is a structure. On Windows, to clear all interrupts use

```
pthread_t nil = { NULL, 0 };

ret = GAUSS_ClearInterrupt(nil);
```

Normally, this function is not necessary because the compiler and executer clear the requests when they terminate.

SEE ALSO **GAUSS_SetInterrupt**, **GAUSS_CheckInterrupt**, **GAUSS_ClearInterrupts**

## GAUSS_CompileExpression

PURPOSE Compiles an expression.

FORMAT **ProgramHandle_t *GAUSS_CompileExpression( WorkspaceHandle_t *wh, char *str, int** *readonlyC*, **int** *readonlyE* **);**

*ph* **= GAUSS_CompileExpression(** *wh*, *str*, *readonlyC*, *readonlyE* **);**

INPUT *wh* pointer to a workspace handle.

*str* pointer to string containing expression.

*readonlyC* 1 or 0, if 1, the compile cannot create or redefine any global symbols. See Section 5.1.

*readonlyE* 1 or 0, if 1, the program cannot assign to global symbols.

OUTPUT *ph* pointer to a program handle.

REMARKS This function compiles an expression and creates a **ProgramHandle_t**. An expression is the right-hand side of an assignment statement without the assignment, for example:

```
x*y + z*inv( k );

diag( chol( x'x ) );
```

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. |
| **530** | User interrupt. |

EXAMPLE

```
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\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;
}
```

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_CompileFile**, **GAUSS_CompileString**, **GAUSS_CompileStringAsFile**, **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 5.1.

*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 any of the following errors:

**30** Insufficient memory.
**495** Workspace inactive or corrupt.
**511** **GAUSS** compile error.
**530** User interrupt.

EXAMPLE

```
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\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 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

**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 5.1.

*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_GetError** 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. |
| **530** | User interrupt. |

EXAMPLE

```
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 **ProgramHandle_t *GAUSS_CompileStringAsFile( WorkspaceHandle_t *wh, char *fn, int** *readonlyC*, **int** *readonlyE* **);**

*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 5.1.

*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:

| | |
|---|---|
| **30** | Insufficient memory. |
| **83** | Error creating temporary file. |
| **495** | Workspace inactive or corrupt. |
| **500** | Cannot create temporary filename. |
| **511** | **GAUSS** compile error. |
| **530** | User interrupt. |

EXAMPLE
```
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_ComplexArray

PURPOSE Creates a **Array_t** for a complex array and copies the array data.

FORMAT **Array_t *GAUSS_ComplexArray( size_t** *dims*, **double ***orders*, **double ***real*, **double ***imag* **);**

*arr* **= GAUSS_ComplexArray(** *dims*, *orders*, *real*, *imag* **);**

INPUT *dims* number of dimensions.

*orders* pointer to orders of dimensions.

*real* pointer to real part of array.

*imag* pointer to imaginary part of array.

OUTPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_ComplexArray** **malloc**'s an **Array_t** and fills it in with your input information. It makes a copy of the array and sets the *adata* member of the **Array_t** to point to the copy. **GAUSS_ComplexArray** should be used only for complex arrays. To create an **Array_t** for a real array, use **GAUSS_Array**. To create an **Array_t** for a complex array without making a copy of the array, use **GAUSS_ComplexArrayAlias**.

Set *imag* to NULL if the array 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 array. Set *orders* to NULL if the vector of orders of the array is located at the beginning of the block of memory that contains the real part of the array. In this case, *real* should point to the vector of orders, followed by the real part of the array. Otherwise, set *orders* to point to the block of memory that contains vector of orders. The vector of orders should contain *dims* doubles.

If *arr* is NULL, there was insufficient memory to **malloc** space for the array and its descriptor.

Use this function to create an array descriptor that you can use in the following functions:

**GAUSS_CopyArrayToArg**
**GAUSS_CopyArrayToGlobal**
**GAUSS_MoveArrayToArg**
**GAUSS_MoveArrayToGlobal**

You can free the **Array_t** with **GAUSS_FreeArray**.

EXAMPLE
```
double orders[3] = { 2.0, 3.0, 3.0 };
double ar[2][3][3]={
          { {3.0, -4.0, 6.0}, {1.0, 2.0, 3.0}, {4.0, 8.0, -2.0} }
          { {4.0, 2.0, -1.0}, {5.0, -6.0, 9.0}, {7.0, 3.0, 1.0} }
                  };
```

```
double ai[2][3][3]={
           { {8.0, 0.0, -1.0}, {-13.0, 5.0, 2.0}, {6.0, 7.0, 4.0} }
           { {-9.0, 3.0, 7.0}, {4.0, 8.0, 2.0}, {5.0, -6.0, 11.0} }
                  };

if ( GAUSS_MoveArrayToGlobal(
            wh,
            GAUSS_ComplexArray( 3, orders, ar, ai ),
            "a"
    ) )
{
    char buff[100];

    printf( "MoveArrayToGlobal 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_ComplexArrayAlias**, **GAUSS_Array**, **GAUSS_CopyArrayToGlobal**, **GAUSS_CopyArrayToArg**, **GAUSS_MoveArrayToGlobal**, **GAUSS_MoveArrayToArg**, **GAUSS_FreeArray**



## GAUSS_ComplexArrayAlias

PURPOSE Creates an **Array_t** for a complex array.

FORMAT **Array_t *GAUSS_ComplexArrayAlias( size_t** *dims*, **double ***addr **);**

*arr* = **GAUSS_ComplexArrayAlias(** *dims*, *addr* **);**

INPUT *dims* number of dimensions.

*addr* pointer to array.

OUTPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_ComplexArrayAlias** is similar to **GAUSS_ComplexArray**; however, it sets the *adata* member of the **Array_t** to point to *addr* instead of making a copy of the array. **GAUSS_ComplexArrayAlias** should be used only for complex arrays. For real arrays, use **GAUSS_Array**.

The argument *addr* should point to a **malloc**'d block containing three sections. The first section, which is the vector of orders for the array, contains *dims* doubles. The second section contains the real part of the array, and the third section contains the imaginary part. The number of doubles in the real section is the product of the vector of orders. The number of doubles in the imaginary section is the same as the real section. These three sections are laid out contiguously in memory.

If *arr* is NULL, there was insufficient memory to **malloc** space for the array descriptor.

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

**GAUSS_CopyArrayToArg**
**GAUSS_CopyArrayToGlobal**
**GAUSS_MoveArrayToArg**
**GAUSS_MoveArrayToGlobal**

You can free the **Array_t** with **GAUSS_FreeArray**. It will not free the array data if the **Array_t** was created with **GAUSS_ComplexArrayAlias**.

EXAMPLE

```
Array_t *arr;
double *x;
size_t dims;

dims = 3;
x = ( double * )malloc( ( 24+dims )*sizeof( double ) );
*x = 2.0;
*( x+1 ) = 3.0;
*( x+2 ) = 2.0;
memset( x+dims, 0, 24*sizeof( double ) );
```

```
if ( ( arr = GAUSS_ComplexArrayAlias( dims, x ) ) == NULL )
{
    char buff[100];

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

if ( GAUSS_CopyArrayToGlobal( wh, arr, "a" ) )
{
    char buff[100];

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

This example **malloc**'s a block of memory containing the vector of orders for the array, followed by the real data of the array and then the imaginary data. In this case, each real and imaginary element of the 2x3x2 array is set to zero. The example create a **Array_t** for the complex array contained in the **malloc**'d block. It then copies the matrix to *wh*, which it assumes to be a pointer to a valid workspace.



SEE ALSO **GAUSS_ComplexArray**, **GAUSS_ArrayAlias**, **GAUSS_CopyArrayToGlobal**, **GAUSS_CopyArrayToArg**, **GAUSS_MoveArrayToGlobal**, **GAUSS_MoveArrayToArg**, **GAUSS_FreeArray**

## GAUSS_ComplexMatrix

PURPOSE Creates a **Matrix_t** for a complex matrix and copies the matrix data.

FORMAT **Matrix_t *GAUSS_ComplexMatrix( size_t** *rows*, **size_t** *cols*,

```
double *real, double *imag );

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:

**GAUSS_CopyMatrixToArg**
**GAUSS_CopyMatrixToGlobal**
**GAUSS_MoveMatrixToArg**
**GAUSS_MoveMatrixToGlobal**

You can free the **Matrix_t** with **GAUSS_FreeMatrix**.

EXAMPLE

```
double mr[3][3]={ {3.0, -4.0, 6.0}, {1.0, 2.0, 3.0}, {4.0, 8.0, -2.0} };
double mi[3][3]={ {8.0, 0.0, -1.0}, {-13.0, 5.0, 2.0}, {6.0, 7.0, 4.0} };
```

```
if ( GAUSS_MoveMatrixToGlobal(
            wh,
            GAUSS_ComplexMatrix( 3, 3, mr, mi ),
            "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**



## GAUSS_ComplexMatrixAlias

PURPOSE Creates a **Matrix_t** for a complex matrix.

FORMAT **Matrix_t *GAUSS_ComplexMatrixAlias( size_t** *rows*, **size_t** *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

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

*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:

**30** Insufficient workspace memory.

**94** Argument out of range.

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

```
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_MoveArgToArg**, **GAUSS_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_InsertArg**, **GAUSS_DeleteArg**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**



## GAUSS_CopyArgToArray

PURPOSE Copies an array from an **ArgList_t** to an **Array_t** structure.

FORMAT **Array_t *GAUSS_CopyArgToArray( ArgList_t *_args_, int _argnum_ );**

*arr* **= GAUSS_CopyArgToArray(** *args*, *argnum* **);**

INPUT *args* pointer to an argument list structure.

*argnum* argument number.

OUTPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_CopyArgToArray** creates an array descriptor, *arr*, and copies an array contained in *args* into it. *arr* belongs to you. Free it with **GAUSS_FreeArray**.

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

This function allows you to retain the array in the **ArgList_t**. If you want to move the array from the **ArgList_t** into an **Array_t**, use **GAUSS_MoveArgToArray**.

If **GAUSS_CopyArgToArray** fails, *arr* will be NULL. Use **GAUSS_GetError** to get the number of the error. **GAUSS_CopyArgToArray** 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;
Array_t *arr;

if ( ( ph = GAUSS_CompileExpression(
                    wh,
                    "sin( reshape( pi*seqa( .2, .4, 40 ), 10, 4 ) )",
                    1,
                    1
      ) ) == NULL )
{
    char buff[100];

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

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

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

if ( ( arr = GAUSS_CopyArgToArray( arg, 1 ) ) == NULL )
{
    char buff[100];

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

The above example copies the array returned from an executed expression into an **Array_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_MoveArrayToArg**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**, **GAUSS_ExecuteExpression**, **GAUSS_GetArgType**, **GAUSS_FreeArgList**



## GAUSS_CopyArgToMatrix

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,
                    "sin( reshape( pi*seqa( .2, .4, 40 ), 10, 4 ) )",
                    1,
                    1
      ) ) == NULL )
{
    char buff[100];

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

```
if ( ( arg = 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_CopyArgToMatrix( arg, 1 ) ) == NULL )
{
    char buff[100];

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

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**



## GAUSS_CopyArgToString

PURPOSE Copies a string from an **ArgList_t** to a **String_t** structure.

FORMAT **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

```
ProgramHandle_t *ph;
ArgList_t *arg;
String_t *str;

if ( ( ph = GAUSS_CompileExpression(
                    wh,
                    "strsect( \"This is a string\", 11, 16 )",
                    1,
                    1
      ) ) == NULL )
{
    char buff[100];

    printf( "Compile failed: %s\n",
```

```
            GAUSS_ErrorText( buff, GAUSS_GetError() ) );
        return -1;
    }

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

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

    if ( ( str = GAUSS_CopyArgToString( arg, 1 ) ) == NULL )
    {
        char buff[100];

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

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 **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

```
ProgramHandle_t *ph;
ArgList_t *arg;
StringArray_t *sa;

if ( ( ph = GAUSS_CompileExpression(
                    wh,
                    "\"cats\" $| \"dogs\"",
                    1,
                    1
      ) ) == NULL )
{
```

```
    char buff[100];

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

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

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

if ( ( sa = GAUSS_CopyArgToStringArray( arg, 1 ) ) == NULL )
{
    char buff[100];

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



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**

## GAUSS_CopyArrayToArg

PURPOSE Copies an array contained in an **Array_t** to an **ArgList_t**.

FORMAT **int GAUSS_CopyArrayToArg( ArgList_t *_args_, Array_t *_arr_, int _argnum_ );**

*ret* = **GAUSS_CopyArrayToArg(** *args*, *arr*, *argnum* **);**

INPUT *args* pointer to an argument list structure.

*arr* pointer to an array descriptor.

*argnum* argument number.

OUTPUT *ret* success flag, 0 if successful, otherwise:

**30** Insufficient memory.
**494** Invalid argument number.

REMARKS **GAUSS_CopyArrayToArg malloc**'s a copy of the array contained in the *arr* 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 array copy will be freed when you call **GAUSS_CallProcFreeArgs** or **GAUSS_FreeArgList** later.

This function allows you to retain *arr*. If you want to move the array to the argument list and free the **Array_t**, use **GAUSS_MoveArrayToArg** instead.

Call **GAUSS_CopyArrayToArg** with an **Array_t** that was returned from one of the following functions:

**GAUSS_ComplexArray**
**GAUSS_ComplexArrayAlias**
**GAUSS_GetArray**
**GAUSS_Array**
**GAUSS_ArrayAlias**

EXAMPLE

```
ArgList_t *args;
Array_t *arr;
int ret;
double orders[3] = { 2.0, 2.0, 3.0 };
double ad[2][2][3] = {
              { { 3.0, 4.0, 2.0 }, { 7.0, 9.0, 5.0 } }
              { { 5.0, 6.0, 1.0 }, { 2.0, 0.0, 4.0 } }
                    };

args = GAUSS_CreateArgList();

arr = GAUSS_Array( 3, orders, ad );

if ( ret = GAUSS_CopyArrayToArg( args, arr, 0 ) )
{
    char buff[100];

    printf( "CopyArrayToArg failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeArgList( args );
    GAUSS_FreeArray( arr );
    return -1;
}
```

The above example copies the array *ad* into the **ArgList_t** *arg*. It retains *arr*, which should be freed later with **GAUSS_FreeArray**.

SEE ALSO **GAUSS_MoveArrayToArg**, **GAUSS_Array**, **GAUSS_ComplexArray**, **GAUSS_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_InsertArg**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**

## GAUSS_CopyArrayToGlobal

PURPOSE

Copies an array contained in an **Array_t** into a **GAUSS** workspace.

FORMAT

```
int GAUSS_CopyArrayToGlobal( WorkspaceHandle_t *wh, Array_t
*arr, char *name );

ret = GAUSS_CopyArrayToGlobal( wh, arr, name );
```

INPUT

*wh* pointer to a workspace handle.

*arr* pointer to an array descriptor.

*name* pointer to name of array.

OUTPUT

*ret* success flag, 0 if successful, otherwise:

| | |
|---|---|
| **26** | Too many symbols. |
| **30** | Insufficient memory. |
| **91** | Symbol too long. |
| **471** | Null pointer. |
| **481** | **GAUSS** assignment failed. |
| **495** | Workspace inactive or corrupt. |

REMARKS

**GAUSS_CopyArrayToGlobal** **malloc**'s a copy of the array contained in *arr* and assigns the copy to *wh*. **GAUSS** frees it when necessary. This function allows you to retain *arr*.

If you want to move the array to *wh* and free the **Array_t**, use **GAUSS_MoveArrayToGlobal** instead.

Call **GAUSS_CopyArrayToGlobal** with an **Array_t** returned from one of the following functions:

**GAUSS_ComplexArray**
**GAUSS_ComplexArrayAlias**
**GAUSS_GetArray**
**GAUSS_Array**
**GAUSS_ArrayAlias**

Input a **WorkspaceHandle_t** returned from **GAUSS_CreateWorkspace**.

EXAMPLE

```
Array_t *arr;
int ret;
double orders[3] = { 2.0, 2.0, 3.0 };
double ad[2][2][3] = {
                { { 3.0, 4.0, 2.0 }, { 7.0, 9.0, 5.0 } }
                { { 6.0, 9.0, 3.0 }, { 8.0, 5.0, 1.0 } }
                      };

arr = GAUSS_Array( 3, orders, ad );

if ( ret = GAUSS_CopyArrayToGlobal( wh, arr, "a" ) )
{
    char buff[100];

    printf( "CopyArrayToGlobal failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeArray( arr );
    return -1;
}
```

The above example copies the array **ad** into the **GAUSS** workspace indicated by *wh*. It assumes that *wh* is a pointer to a valid workspace handle. It retains *arr*, which should be freed later with **GAUSS_FreeArray**.

SEE ALSO **GAUSS_MoveArrayToGlobal**, **GAUSS_Array**, **GAUSS_ComplexArray**, **GAUSS_AssignFreeableArray**, **GAUSS_GetArray**

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

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

*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
```
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 *a* from *wh1* into *wh2* and calls the matrix copy *rmat*.

SEE ALSO **GAUSS_GetMatrix**, **GAUSS_GetString**, **GAUSS_GetStringArray**

## GAUSS_CopyMatrixToArg

PURPOSE Copies a matrix contained in a **Matrix_t** to an **ArgList_t**.

## GAUSS_CopyMatrixToArg

FORMAT **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

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


args = GAUSS_CreateArgList();

mat = GAUSS_Matrix( 2, 3, md );

if ( ret = GAUSS_CopyMatrixToArg( args, mat, 0 ) )
{
    char buff[100];

    printf( "CopyMatrixToArg failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeArgList( args );
    GAUSS_FreeMatrix( mat );
    return -1;
}
```

The above example copies the matrix *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**



## GAUSS_CopyMatrixToGlobal

PURPOSE Copies a matrix contained in a **Matrix_t** into a **GAUSS** workspace.

FORMAT **int GAUSS_CopyMatrixToGlobal( WorkspaceHandle_t ***wh**, Matrix_t ***mat**, char ***name **);**

*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:

**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 );

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**

## GAUSS_CopyStringArrayToArg



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

FORMAT **int GAUSS_CopyStringArrayToArg( ArgList_t *** *args*, **StringArray_t *** *sa*, **int** *argnum* **);**

*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

```
ArgList_t *args;
StringArray_t *sa;
int ret;

args = GAUSS_CreateArgList();

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

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

if ( ( ret = GAUSS_CopyStringArrayToArg( args, sa, 0 ) ) == NULL )
{
    char buff[100];

    printf( "CopyStringArrayToArg failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
```

```
    GAUSS_FreeArgList( args );
    GAUSS_FreeStringArray( sa );
    return -1;
}
```

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**

## 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:
- **26** Too many symbols.
- **30** Insufficient memory.
- **91** Symbol too long.
- **481** **GAUSS** assignment failed.
- **495** Workspace inactive or corrupt.

REMARKS

**GAUSS_CopyStringArrayToGlobal** **malloc**'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

```
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**

# GAUSS_CopyStringToArg

PURPOSE Copies a string contained in a **String_t** to an **ArgList_t**.

FORMAT **int GAUSS_CopyStringToArg( ArgList_t *** *args* **, String_t *** *str* **, int** *argnum* **);**

*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 **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_StringL**

EXAMPLE

```
ArgList_t *args;
char s[] = "This is a string.";
int ret;

args = GAUSS_CreateArgList();

str = GAUSS_StringL( s, 18 );

if ( ret = GAUSS_CopyStringToArg( args, str, 0 )
{
    char buff[100];

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

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**

## GAUSS_CopyStringToGlobal

PURPOSE Copies a string contained in a **String_t** into a **GAUSS** workspace.

FORMAT

```
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 assignes 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:

**GAUSS_GetString**
**GAUSS_String**
**GAUSS_StringAlias**
**GAUSS_StringAliasL**
**GAUSS_StringL**

Input a **WorkspaceHandle_t** returned from **GAUSS_CreateWorkspace**.

EXAMPLE

```
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 *e* into the **GAUSS** workspace indicated by *wh*. It assumes that *wh* is a pointer to a valid workspace handle. It retains *str*, which should be freed later with **GAUSS_FreeString**.

SEE ALSO **GAUSS_MoveStringToGlobal**, **GAUSS_String**, **GAUSS_StringAlias**, **GAUSS_GetString**

## GAUSS_CreateArgList

PURPOSE Creates an empty argument list.

FORMAT **ArgList_t *GAUSS_CreateArgList( void );**

*args* **= GAUSS_CreateArgList();**

OUTPUT *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

```
ArgList_t *args;

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

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

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**

## GAUSS_CreateProgram

PURPOSE Creates an empty program handle.

FORMAT **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
```
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 **WorkspaceHandle_t *GAUSS_CreateWorkspace( char *_name_ );**

*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_GetError** 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

```
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**

## GAUSS_DeleteArg

PURPOSE Deletes an argument from an **ArgList_t**.

FORMAT **int GAUSS_DeleteArg( ArgList_t *** *args*, **int** *argnum* **);**

*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

```
ProgramHandle_t *ph;
ArgList_t *args;

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

## GAUSS_ErrorText

PURPOSE Returns the error message that corresponds to a given error number.

FORMAT **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

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

*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.
- **530** User interrupt.

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

```
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\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.
**530** User interrupt.

EXAMPLE

```
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 **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
```
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;
}

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**

## GAUSS_FreeArray

PURPOSE Frees an array descriptor and the data it contains.

FORMAT **void GAUSS_FreeArray( Array_t *** *arr* **);**

**GAUSS_FreeArray(** *arr* **);**

INPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_FreeArray** frees an array descriptor and the array it points to.

EXAMPLE

```
Array_t *arr;
ArgList_t *args;
double orders[3] = { 2.0, 4.0, 2.0 };
double x[2][4][2] = {
          { { 3.0, -4.0 }, { 6.0, 9.0 }, {-5.0, 0.0 }, { -1.0, -8.0 } }
          { { 9.0, -2.0 }, { 0.0, -3.0 }, { 1.0, 4.0 }, { 7.0, 5.0 } }
                    };

args = GAUSS_CreateArgList();

if ( ( arr = GAUSS_Array( 3, orders, x ) ) == NULL )
{
    char buff[100];

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

if ( GAUSS_CopyArrayToArg( args, arr, 0 ) )
{
    char buff[100];

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

GAUSS_FreeArray( arr );
```

The above example creates an array descriptor, *arr*, and copies it to *args* as its first argument. It then frees *arr*.

SEE ALSO **GAUSS_Array**, **GAUSS_ArrayAlias**, **GAUSS_ComplexArray**, **GAUSS_ComplexArrayAlias**, **GAUSS_GetArray**

## GAUSS_FreeMatrix

PURPOSE Frees a matrix descriptor and the data it contains.

FORMAT **void GAUSS_FreeMatrix( Matrix_t ****mat* **);**

**GAUSS_FreeMatrix(** *mat* **);**

INPUT *mat* pointer to a matrix descriptor.

REMARKS **GAUSS_FreeMatrix** frees a matrix descriptor and the matrix it points to.

EXAMPLE
```
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**

## GAUSS_FreeProgram

PURPOSE Frees a program handle.

FORMAT **void GAUSS_FreeProgram( ProgramHandle_t *_ph_ );**

**GAUSS_FreeProgram(** *ph* **);**

INPUT *ph* pointer to a program handle.

REMARKS **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

```
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\n",
        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**

## GAUSS_FreeString

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**



## GAUSS_FreeStringArray

PURPOSE Frees a string array descriptor and the data it contains.

FORMAT **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
```
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 **void GAUSS_FreeWorkspace( WorkspaceHandle_t *_wh_ );**

**GAUSS_FreeWorkspace(** *wh* **);**

INPUT *wh* pointer to a workspace handle.

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

EXAMPLE
```
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 **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_ARRAY**
**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_CopyArgToArray**
**GAUSS_CopyArgToMatrix**
**GAUSS_CopyArgToString**
**GAUSS_CopyArgToStringArray**
**GAUSS_MoveArgToArray**
**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

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

if ( ( ph = GAUSS_CompileExpression(
                    wh,
                    "prodc( seqa( 1, .01, 25 ) );",
                    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 ( ( GAUSS_GetArgType( ret, 1 ) ) != GAUSS_MATRIX )
{
    printf( "Argument corrupt\n" );
    GAUSS_FreeProgram( ph );
    GAUSS_FreeArgList( ret );
    return -1;
}

if ( ( mat = GAUSS_MoveArgToMatrix( args, 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. It executes an expression, which places its return in an *ArgList_t*. The example checks to make sure that the return is of type **GAUSS_MATRIX** before moving it to a matrix descriptor.

SEE ALSO **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**, **GAUSS_ExecuteExpression**

## GAUSS_GetArray

PURPOSE Gets a global array from a **GAUSS** workspace.

FORMAT **Array_t *GAUSS_GetArray( WorkspaceHandle_t *_wh_, char *_name_ );**

*arr* = **GAUSS_GetArray(** *wh*, *name* **);**

INPUT *wh* pointer to a workspace handle.

*name* pointer to name of array.

OUTPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_GetArray** finds an array in a **GAUSS** workspace and **malloc**'s an array descriptor, filling it in with the information for the array. It makes a copy of the array and sets the *adata* member of the array descriptor to point to the copy. This gives you a safe copy of the array that you can work with without affecting

the contents of the **GAUSS** symbol table. This copy of the array then belongs to you. Free it with **GAUSS_FreeArray**.

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

If the array is empty, the *dims* and *nelems* members of the *Array_t* will be set to 0, and the *adata* member will be NULL.

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

If **GAUSS_GetArray** fails, *arr* will be NULL. Use **GAUSS_GetError** to get the number of the error. **GAUSS_GetArray** 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;
Array_t *arr;
int ret;

if ( ( ph = GAUSS_CompileString(
                    wh,
                    "orders = { 3,4,5,6,7 };
                        a = areshape(seqa(1,1,prodc(orders)),orders);
                        b = atranspose(a,2|4|3|5|1);",
                    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 ( ( arr = GAUSS_GetArray( wh, "b" ) ) == NULL )
{
    char buff[100];

    printf( "GetArray 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_GetArrayAndClear**, **GAUSS_CopyArrayToGlobal**, **GAUSS_MoveArrayToGlobal**

## GAUSS_GetArrayAndClear

PURPOSE Gets a global array from a **GAUSS** workspace and clears the array in that workspace.

FORMAT **Array_t *GAUSS_GetArrayAndClear( WorkspaceHandle_t *wh, char *name );**

*arr* **= GAUSS_GetArrayAndClear(** *wh*, *name* **);**

INPUT *wh* pointer to a workspace handle.

*name* pointer to name of array.

OUTPUT *arr* pointer to a array descriptor.

REMARKS **GAUSS_GetArrayAndClear** finds an array in a **GAUSS** workspace and **malloc**'s an array descriptor, filling it in with the information for the array. It sets the *adata* member of the *Array_t* to point to the array and sets the array to a 1-dimensional array of 1 element with a value of 0 in the **GAUSS** symbol table. This allows you to get large arrays from a **GAUSS** workspace without using the time and memory space needed to copy the array. The array then belongs to you. Free it with **GAUSS_FreeArray**.

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

If the array is empty, the *dims* and *nelems* members of the *Array_t* will be set to 0, and the *adata* member will be NULL.

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

If **GAUSS_GetArrayAndClear** fails, *arr* will be NULL. Use **GAUSS_GetError** to get the number of the error. **GAUSS_GetArrayAndClear** 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;
Array_t *arr;
int ret;

if ( ( ph = GAUSS_CompileString(
```

```
                        wh,
                        "a = dimensioninit(100|100|20|10|5,1);",
                        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_GetArrayAndClear( wh, "a" ) ) == NULL )
{
    char buff[100];

    printf( "GetArrayAndClear 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 5-dimensional array of ones, *a*, and resets *a* in *wh* to a 1-dimensional array of 1 element that is set to zero.

SEE ALSO **GAUSS_GetArray**, **GAUSS_CopyArrayToGlobal**, **GAUSS_MoveArrayToGlobal**,

## GAUSS_GetDouble

PURPOSE Gets a global double from a **GAUSS** workspace.

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

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

if ( ( ph = GAUSS_CompileString(
                    wh,
                    "{ a, rs } = rndKMn( 1, 1, 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_GetDouble( wh, &d, "a" ) )
{
    char buff[100];

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

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

SEE ALSO **GAUSS_PutDouble**, **GAUSS_GetMatrix**

## GAUSS_GetError

PURPOSE Returns the stored error number.

FORMAT **int GAUSS_GetError( void );**

*errnum* = **GAUSS_GetError();**

OUTPUT *errnum* error number.

REMARKS The **GAUSS Engine** stores the error number of the most recently encountered error in a system variable. If a **GAUSS 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 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
```
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**

## GAUSS_GetHomeVar

PURPOSE Gets the name of the current home environment variable for the **GAUSS Engine**.

FORMAT **char *GAUSS_GetHomeVar( char *_buff_ );**

*hvar* = **GAUSS_GetHomeVar(** *buff* **);**

INPUT *buff* pointer to buffer to put name of home environment variable.

OUTPUT *hvar* pointer to buffer.

REMARKS **GAUSS_GetHomeVar** fills *buff* with the name of the current home environment variable and returns a pointer to that buffer.

The default home environment variable is **MTENGHOME13**. Use the C library function *getenv* to get the value of the environment variable.

EXAMPLE
```
char buff[100];

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

SEE ALSO **GAUSS_GetHome**, **GAUSS_SetHomeVar**, **GAUSS_SetHome**, **GAUSS_Initialize**

## GAUSS_GetLogFile



PURPOSE Gets the name of the current log file.

FORMAT **char *GAUSS_GetLogFile( char *buff );**

*logfn* = **GAUSS_GetLogFile(** *buff* **);**

INPUT *buff* pointer to buffer for log file name to be put in.

OUTPUT *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
```
char buff[40];
```

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

SEE ALSO **GAUSS_SetLogFile**, **GAUSS_GetLogStream**, **GAUSS_SetLogStream**

## GAUSS_GetLogStream

PURPOSE Gets the current log file pointer.

FORMAT **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**

## GAUSS_GetMatrix

PURPOSE Gets a global matrix from a **GAUSS** workspace.

FORMAT **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
```
Matrix_t *GAUSS_GetMatrixAndClear( WorkspaceHandle_t *wh,
char *name );

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. |

EXAMPLE

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

if ( ( ph = GAUSS_CompileString(
                    wh,
                    "{ a, rs } = 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**

## GAUSS_GetMatrixInfo

PURPOSE Gets information for a matrix in a **GAUSS** workspace.

FORMAT **int GAUSS_GetMatrixInfo( WorkspaceHandle_t ****wh***, GAUSS_MatrixInfo_t ****matinfo***, char ****name** **);**

*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 *wh* is a pointer to a valid workspace handle.

SEE ALSO **GAUSS_GetMatrix**, **GAUSS_GetMatrixAndClear**, **GAUSS_CopyMatrixToGlobal**, **GAUSS_MoveMatrixToGlobal**, **GAUSS_AssignFreeableMatrix**, **GAUSS_GetDouble**

## GAUSS_GetString

PURPOSE Gets a global string from a **GAUSS** workspace.

FORMAT **String_t *GAUSS_GetString( WorkspaceHandle_t *wh, char *name );**

*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

```
ProgramHandle_t *ph;
String_t *str;
int ret;

if ( ( ph = GAUSS_CompileString(
                    wh,
                    "s = \"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 ( str = GAUSS_GetString( wh, "s" ) )
{
    char buff[100];
```

```
    printf( "GetString 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_FreeString**, **GAUSS_GetError**, **GAUSS_CopyStringToGlobal**, **GAUSS_MoveStringToGlobal**, **GAUSS_GetStringArray**

## GAUSS_GetStringArray



PURPOSE Gets a global string array from a **GAUSS** workspace.

FORMAT **StringArray_t *GAUSS_GetStringArray( WorkspaceHandle_t *wh, char *name );**

*sa* = **GAUSS_GetStringArray(** *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 **GAUSS_GetStringArray** 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. **GAUSS_GetStringArray** 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 **GAUSS_FreeStringArray**.

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

If **GAUSS_GetStringArray** fails, *sa* will be NULL. Use **GAUSS_GetError** to get the number of the error. **GAUSS_GetStringArray** 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;
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**



## GAUSS_GetSymbolType

PURPOSE Gets the type of a symbol in a **GAUSS** workspace.

FORMAT **int GAUSS_GetSymbolType( WorkspaceHandle_t *wh, char *name );**

*typ* **= GAUSS_GetSymbolType(** *wh*, *name* **);**

INPUT *wh* pointer to a workspace handle.

*name* pointer to name of symbol.

OUTPUT *typ* type of symbol:

GAUSS_ARRAY
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:

| | |
|---|---|
| **91** | Symbol too long. |
| **495** | Workspace inactive or corrupt. |

EXAMPLE

```
ProgramHandle_t *ph;
Matrix_t *mat;
int ret, typ;

if ( ( ph = GAUSS_CompileString(
                    wh,
                    "b = { \"apple\" \"orange\" \"pear\" };",
                    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 ( ( typ = GAUSS_GetSymbolType( wh, "b" ) ) != GAUSS_MATRIX )
{
    printf( "Wrong symbol type\n" );
    GAUSS_FreeProgram( ph );
    return -1;
}

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

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

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_GetArray**, **GAUSS_GetMatrix**, **GAUSS_GetString**, **GAUSS_GetStringArray**



## GAUSS_GetWorkspaceName

PURPOSE Gets the name of a **GAUSS** workspace.

FORMAT **char *GAUSS_GetWorkspaceName( WorkspaceHandle_t *_wh_, char *_buff_ );**

*bp* = **GAUSS_GetWorkspaceName(** *wh*, *buff* **);**

INPUT *wh* pointer to a workspace handle.

*buff* pointer to character buffer at least 64 bytes in length.

OUTPUT *bp* pointer, same as buff.

REMARKS **GAUSS_GetWorkspaceName** fills in the character buffer, *buff*, with the name of a **GAUSS** workspace indicated by a **WorkspaceHandle_t**. If the buffer is shorter than 64 bytes, this can core dump.

SEE ALSO **GAUSS_CreateWorkspace**, **GAUSS_SetWorkspaceName**

## GAUSS_HookFlushProgramOutput

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

FORMAT **void GAUSS_HookFlushProgramOutput( void ( *_flush_output_fn_ )( void ) );**

**GAUSS_HookFlushProgramOutput(** *flush_output_fn* **);**

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**

## GAUSS_HookGetCursorPosition

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

FORMAT
```
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 **csrlin** commands to get the position of the cursor. Your get cursor postion function must take nothing and return an **int**, the position of the cursor.

Many **GAUSS** programs perform I/O, but the **GAUSS 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 **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
```
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:

```
GAUSS_HookProgramErrorOutput( program_error );
```

SEE ALSO **GAUSS_HookProgramOutput**

## GAUSS_HookProgramInputChar

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

FORMAT **void GAUSS_HookProgramInputChar( int ( *_input_char_function_ )( void ) );**

**GAUSS_HookProgramInputChar(** *input_char_function* **);**

INPUT *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**

## GAUSS_HookProgramInputCharBlocking

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

FORMAT
```
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**

## GAUSS_HookProgramInputCheck

PURPOSE Specifies the function **GAUSS** calls to check for pending input.

FORMAT **void GAUSS_HookProgramInputCheck( int ( *_input_check_fn_ )( void ) );**

**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 **void GAUSS_HookProgramInputString( int ( *_input_string_fn_ )( char *, int ) );**

**GAUSS_HookProgramInputString(** *input_string_fn* **);**

INPUT *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 **GAUSS 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**

## GAUSS_HookProgramOutput

PURPOSE Specifies the function **GAUSS** calls to display program output.

FORMAT **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 **GAUSS 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

```
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:

```
GAUSS_HookProgramOutput( program_output );
```

SEE ALSO **GAUSS_HookProgramErrorOutput**



## GAUSS_Initialize

PURPOSE Initializes the engine.

FORMAT **int GAUSS_Initialize( void );**

*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**

## GAUSS_InsertArg

PURPOSE Inserts an empty argument into an **ArgList_t**.

FORMAT **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 **GAUSS_InsertArg** 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. **GAUSS_InsertArg** 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**



## GAUSS_IsMissingValue

PURPOSE Checks a double to see if it contains a **GAUSS** missing value.

FORMAT
```
int GAUSS_IsMissingValue( double *d );

ret = GAUSS_IsMissingValue( d );
```

INPUT *d* data.

OUTPUT *ret* 1 if *d* contains a **GAUSS** missing value, 0 if not.

EXAMPLE
```
double d;
```

```
if ( ret = GAUSS_GetDouble( wh, &d, "a" ) )
{
    char buff[100];

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

if ( ! GAUSS_IsMissingValue( &d ) )
    printf( "a = %lf", d );
```

This example assumes that *a* is a global 1x1 matrix in the **GAUSS** workspace indicated by *wh*. It finds *a* in the **GAUSS** workspace and sets *d* to its value. The example then checks to see if *d* contains a **GAUSS** missing value and prints its value if it does not.

SEE ALSO **GAUSS_MissingValue**

## GAUSS_LoadCompiledBuffer

PURPOSE Loads a compiled program stored in a character buffer.

FORMAT **ProgramHandle_t *GAUSS_LoadCompiledBuffer( WorkspaceHandle_t *wh, char *buff );**

*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,
                    0
     ) ) == NULL )
{
    char buff[100];

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

if ( ret = GAUSS_SaveProgram( ph1, "det.gcg" ) )
{
    char buff[100];

    printf( "GAUSS_SaveProgram failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );
    GAUSS_FreeProgram( ph1 );
    return -1;
}
```

```
if ( ( ph2 = GAUSS_LoadCompiledFile( wh2, "det.gcg" ) ) == NULL )
{
    char buff[100];

    printf( "GAUSS_LoadCompiledFile failed: %s\n",
        GAUSS_ErrorText( buff, GAUSS_GetError() ) );
    GAUSS_FreeProgram( ph1 );
    return -1;
}
```

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**



## GAUSS_LoadWorkspace

PURPOSE Loads workspace information stored in a file.

FORMAT **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_MakePathAbsolute

PURPOSE Takes a partial path and makes it absolute.

FORMAT **void GAUSS_MakePathAbsolute( char ***_path_ **);**

**GAUSS_MakePathAbsolute(** *path* **);**

INPUT *path* pointer to buffer containing partial path.

REMARKS **GAUSS_MakePathAbsolute** overwrites the input buffer containing the partial path with the corresponding absolute path.

SEE ALSO **GAUSS_SetHome**

## GAUSS_Matrix

PURPOSE Creates a **Matrix_t** for a real matrix and copies the matrix data.

FORMAT **Matrix_t *GAUSS_Matrix( size_t** *rows*, **size_t** *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:

**GAUSS_CopyMatrixToArg**
**GAUSS_CopyMatrixToGlobal**
**GAUSS_MoveMatrixToArg**
**GAUSS_MoveMatrixToGlobal**

Free the **Matrix_t** with **GAUSS_FreeMatrix**.

EXAMPLE

```
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 **Matrix_t *GAUSS_MatrixAlias( size_t** *rows*, **size_t** *cols*, **double *** *addr* **);**

*mat* **= GAUSS_MatrixAlias(** *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_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:

**GAUSS_CopyMatrixToArg**
**GAUSS_CopyMatrixToGlobal**
**GAUSS_MoveMatrixToArg**
**GAUSS_MoveMatrixToGlobal**

Free the **Matrix_t** with **GAUSS_FreeMatrix**. The matrix data will not be freed or overwritten by **GAUSS_FreeMatrix** or any other **Engine** commands. You are responsible for freeing the data pointed to by *addr*.



EXAMPLE

```
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() ) );

    free(a);
    return -1;
}

if ( ret = GAUSS_CopyMatrixToGlobal( wh, mat, "c" ) )
{
```

```
    char buff[100];

    printf( "CopyMatrixToGlobal failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );

    GAUSS_FreeMatrix( mat );
    free(a);
    return -1;
}

free(a);
```

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_MissingValue

PURPOSE Returns a **GAUSS** missing value.

FORMAT **double GAUSS_MissingValue( void );**

*miss* **= GAUSS_MissingValue();**

OUTPUT *miss* **GAUSS** missing value.

SEE ALSO **GAUSS_IsMissingValue**

## GAUSS_MoveArgToArg

PURPOSE Moves an argument from one **ArgList_t** to another.

FORMAT
```
int GAUSS_MoveArgToArg( ArgList_t *targs, int targnum,
ArgList_t *sargs, int sargnum );

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

```
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_CopyArgToArg**, **GAUSS_CreateArgList**, **GAUSS_InsertArg**, **GAUSS_FreeArgList**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**

## GAUSS_MoveArgToArray



PURPOSE Moves an array from an **ArgList_t** to an **Array_t** structure.

FORMAT **Array_t *GAUSS_MoveArgToArray( ArgList_t *_args_, int _argnum_ );**

*arr* **= GAUSS_MoveArgToArray(** *args*, *argnum* **);**

INPUT *args* pointer to an argument list structure.

*argnum* number of argument in the argument list.

OUTPUT *arr* pointer to an array descriptor.

REMARKS **GAUSS_MoveArgToArray** creates an array descriptor, *arr*, and moves an array contained in *args* into it. *arr* belongs to you. Free it with **GAUSS_FreeArray**.

**GAUSS_MoveArgToArray** clears the *argnum* argument descriptor after moving the 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 array in the **ArgList_t**, use **GAUSS_CopyArgToArray** instead. However, **GAUSS_MoveArgToArray** saves time and memory space.

If **GAUSS_MoveArgToArray** fails, *arr* will be NULL. Use **GAUSS_GetError** to get the number of the error. **GAUSS_MoveArgToArray** 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;
Array_t *arr;

if ( ( ph = GAUSS_CompileExpression(
                    wh,
                    "asum(areshape(seqa(1,1,120),2|3|4|5),3);",
                    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 ( ( arr = GAUSS_MoveArgToArray( ret, 1 ) ) == NULL )
{
    char buff[100];

    printf( "MoveArgToArray 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_CopyArgToArray**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**, **GAUSS_ExecuteExpression**, **GAUSS_FreeArray**, **GAUSS_GetArgType**, **GAUSS_GetError**



## GAUSS_MoveArgToMatrix

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_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 **String_t *GAUSS_MoveArgToString( ArgList_t *** *args* **, int** *argnum* **);**

*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

```
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**



## GAUSS_MoveArgToStringArray

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\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 ( ( sa = GAUSS_MoveArgToStringArray( args, 1 ) ) == NULL )
{
    char buff[100];

    printf( "MoveArgToStringArray failed: %s\n",
        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_CopyArgToStringArray**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**, **GAUSS_ExecuteExpression**, **GAUSS_FreeStringArray**, **GAUSS_GetArgType**, **GAUSS_GetError**



## GAUSS_MoveArrayToArg

PURPOSE Moves an array contained in an **Array_t** to an **ArgList_t** and frees the **Array_t**.

FORMAT **int GAUSS_MoveArrayToArg( ArgList_t ***args**, Array_t ***arr**, int** *argnum* **);**

*ret* = **GAUSS_MoveArrayToArg(** *args*, *arr*, *argnum* **);**

INPUT

*args* pointer to an argument list structure.

*arr* pointer to an array descriptor.

*argnum* number of argument.

OUTPUT

*ret* success flag, 0 if successful, otherwise:

**30** Insufficient memory.
**494** Invalid argument number.

REMARKS

**GAUSS_MoveArrayToArg** moves the array contained in *arr* into *args* and frees *arr*.

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 array will be freed when you call **GAUSS_CallProcFreeArgs** or **GAUSS_FreeArgList** later.

If you want to retain *arr*, use **GAUSS_CopyArrayToArg** instead. However, **GAUSS_MoveArrayToArg** saves time and memory space.

Call **GAUSS_MoveArrayToArg** with an **Array_t** returned from **GAUSS_Array**, **GAUSS_ComplexArray**, or **GAUSS_GetArray**.

SEE ALSO

**GAUSS_CopyArrayToArg**, **GAUSS_Array**, **GAUSS_ComplexArray** **GAUSS_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_InsertArg**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**

## GAUSS_MoveArrayToGlobal

PURPOSE

Moves an array contained in an **Array_t** into a **GAUSS** workspace and frees the **Array_t**.

FORMAT

```
int GAUSS_MoveArrayToGlobal( WorkspaceHandle_t *wh, Array_t
*arr, char *name );

ret = GAUSS_MoveArrayToGlobal( wh, arr, name );
```

INPUT

*wh* pointer to a workspace handle.

*arr* pointer to an array descriptor.

*name* pointer to name of array.

OUTPUT

*ret* success flag, 0 if successful, otherwise:

| | |
|---|---|
| **26** | Too many symbols. |
| **30** | Insufficient memory. |
| **91** | Symbol too long. |
| **471** | Null pointer. |
| **481** | **GAUSS** assignment failed. |
| **495** | Workspace inactive or corrupt. |

REMARKS

**GAUSS_MoveArrayToGlobal** moves the matrix contained in *arr* into a **GAUSS** workspace and frees *arr*. **GAUSS** takes ownership of the matrix and frees it when necessary.

If you want to retain *arr*, use **GAUSS_CopyArrayToGlobal** instead. However, **GAUSS_MoveArrayToGlobal** saves time and memory space.

Call **GAUSS_MoveArrayToGlobal** with an **Array_t** returned from one of the following functions;

**GAUSS_ComplexArray**
**GAUSS_ComplexArrayAlias**
**GAUSS_GetArray**
**GAUSS_Array**
**GAUSS_ArrayAlias**

Input a **WorkspaceHandle_t** returned from **GAUSS_CreateWorkspace**.

EXAMPLE

```
Array_t *arr;
int ret;
double orders[3] = { 2.0, 2.0, 3.0 };
double ad[2][2][3] = {
                { { 3.0, 4.0, 2.0 }, { 7.0, 9.0, 5.0 } }
                { { 6.0, 9.0, 3.0 }, { 8.0, 5.0, 1.0 } }
                      };

arr = GAUSS_Array( 3, orders, ad );

if ( ret = GAUSS_MoveArrayToGlobal( wh, arr, "a" ) )
{
    char buff[100];

    printf( "MoveArrayToGlobal failed: %s\n",
        GAUSS_ErrorText( buff, ret ) );

    return -1;
}
```

The above example moves the array **ad** into the **GAUSS** workspace indicated by *wh*. It assumes that *wh* is a pointer to a valid workspace handle. It frees *arr*.

SEE ALSO **GAUSS_CopyArrayToGlobal**, **GAUSS_Array**, **GAUSS_ComplexArray**, **GAUSS_AssignFreeableArray**, **GAUSS_GetArray**

## GAUSS_MoveMatrixToArg

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

```
int GAUSS_MoveMatrixToGlobal( WorkspaceHandle_t *wh,
Matrix_t *mat, char *name );

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

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

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 **int GAUSS_MoveStringToArg( ArgList_t ***args**, String_t ***str**, int** *argnum* **);**

*ret* **= GAUSS_MoveStringToArg(** *args***,** *str***,** *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_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_InsertArg**, **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
```
int GAUSS_MoveStringToGlobal( WorkspaceHandle_t *wh,
String_t *str, char *name );

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**



## GAUSS_ProgramErrorOutput

PURPOSE Passes a string to the program error callback function.

FORMAT **void GAUSS_ProgramErrorOutput( char *** *str* **);**

**GAUSS_ProgramErrorOutput(** *str* **);**

INPUT *str* pointer to a string.

REMARKS **GAUSS_ProgramErrorOutput** passes a string to the program error callback function hooked with **GAUSS_HookProgramErrorOutput**.

The callbacks are thread specific. **GAUSS_ProgramErrorOutput** will call the callback function that was hooked in that particular thread.

EXAMPLE

```
char strbuff[50];

strcpy( strbuff, "Test 1 error output:" );

GAUSS_ProgramErrorOutput( strbuff );
```

This example assumes that a program error output callback function has already been hooked with **GAUSS_HookProgramErrorOutput** in this thread. This call to **GAUSS_ProgramErrorOutput** will pass *strbuff* to that callback function.

SEE ALSO **GAUSS_HookProgramErrorOutput**, **GAUSS_ProgramOutput**

## GAUSS_ProgramInputString

PURPOSE Calls for user input using the program input string function.

FORMAT **int GAUSS_ProgramInputString( char *_buff_, int _bufflen_ );**

*len* = **GAUSS_ProgramInputString(** *buff*, *bufflen* **);**

INPUT *buff* pointer to buffer in which to place input.

*bufflen* length of buffer.

REMARKS **GAUSS_ProgramInputString** calls the program input string function hooked with **GAUSS_HookProgramInputString**. It passes the pointer to a character buffer in which the input is to be placed to the input string function, as well as the length of the buffer. **GAUSS_ProgramInputString** returns the length of the string inputted into the buffer.

The callbacks are thread specific. **GAUSS_ProgramInputString** will call the input string function that was hooked in that particular thread.

EXAMPLE

```
char strbuff[1024];
int len;

printf("Enter name of GAUSS program file to run:\n");
len = GAUSS_ProgramInputString( strbuff, 1024 );

if ( ( ph = GAUSS_CompileFile( wh, strbuff, 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;
}
```

This example assumes that a program input string function has already been hooked with **GAUSS_HookProgramInputString** in this thread. This call to **GAUSS_ProgramInputString** will get user input using that input string function and place it in *strbuff*. This example assumes that the input string will contain the name of a **GAUSS** program file, which it then attempts to run in **GAUSS**.

SEE ALSO **GAUSS_HookProgramInputString**



## GAUSS_ProgramOutput

PURPOSE Passes a string to the program output callback function.

FORMAT **void GAUSS_ProgramOutput( char *_str_ );**

**GAUSS_ProgramOutput(** *str* **);**

INPUT *str* pointer to a string.

REMARKS **GAUSS_ProgramOutput** passes a string to the program output callback function hooked with **GAUSS_HookProgramOutput**.

The callbacks are thread specific. **GAUSS_ProgramOutput** will call the callback function that was hooked in that particular thread.

EXAMPLE
```
char strbuff[50];

strcpy( strbuff, "Test 1 output:" );

GAUSS_ProgramOutput( strbuff );
```

This example assumes that a program output callback function has already been hooked with **GAUSS_HookProgramOutput** in this thread. This call to **GAUSS_ProgramOutput** will pass *strbuff* to that callback function.

SEE ALSO **GAUSS_HookProgramOutput**, **GAUSS_ProgramErrorOutput**

## GAUSS_PutDouble

PURPOSE Puts a **double** into a **GAUSS** workspace.

FORMAT **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:

**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_PutDoubleInArg



PURPOSE Puts a **double** into an **ArgList_t**.

FORMAT **int GAUSS_PutDoubleInArg( ArgList_t ***args**, double** *d***, int** *argnum* **);**

*ret* **= GAUSS_PutDoubleInArg(** *args***,** *d***,** *argnum* **);**

INPUT *args* pointer to an argument list structure.

*d* data.

*argnum* number of argument.

OUTPUT *ret* success flag, 0 if successful, otherwise:

**30** Insufficient memory.
**494** Invalid argument number.

REMARKS **GAUSS_PutDouble** puts the double *d* into *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.

SEE ALSO **GAUSS_CopyMatrixToArg**, **GAUSS_MoveMatrixToArg**, **GAUSS_CreateArgList**, **GAUSS_FreeArgList**, **GAUSS_InsertArg**, **GAUSS_CallProc**, **GAUSS_CallProcFreeArgs**

## GAUSS_SaveProgram

PURPOSE Saves a compiled program as a file.

FORMAT
```
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, otherwise:

| | |
|---|---|
| **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**

## GAUSS_SaveWorkspace

PURPOSE Saves workspace information in a file.

FORMAT
```
int GAUSS_SaveWorkspace( WorkspaceHandle_t *wh, char *fn );

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
```
int GAUSS_SetError( int newerrnum );

olderrnum = GAUSS_SetError( newerrnum );
```

INPUT *newerrnum* new error number.

OUTPUT *olderrnum* previous error number.

REMARKS The **GAUSS Engine** stores the error number of the most recently encountered error in a system variable. If a **GAUSS 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**

## GAUSS_SetHome

PURPOSE Sets the home path for the **GAUSS Engine**.

FORMAT
```
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 Engine**.

FORMAT
```
int GAUSS_SetHomeVar( char *newname );

ret = GAUSS_SetHomeVar( newname );
```

INPUT *newname* pointer to new name to be set.

OUTPUT *ret* success code, 0 if successful, otherwise 486 if character argument too long.

REMARKS The default value is **MTENGHOME13**. 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**

## GAUSS_SetInterrupt

PURPOSE Sets an program interrupt request on a thread.

FORMAT **int GAUSS_SetInterrupt( pthread_t** *tid* **);**

*ret* **= GAUSS_SetInterrupt(** *tid* **);**

INPUT *tid* thread id of thread to interrupt.

OUTPUT *ret* success code, ≥ 0 if successful.

REMARKS If ret is 0, the interrupt request is successful. The program or compile may not stop immediately. If the program is executing an intrinsic function on a large matrix, such an an inverse or matrix multiply, the operation will continue until it is finished and the program will stop at the next instruction.

If *ret* is greater than 0, the interrupt is already requested and *ret* is the UTC time of the original request.

If *ret* is -1, the system is out of memory.

If *ret* is -2, the Engine is shutdown and the interrupt request has been ignored.

Interrupts are checked during certain I/O statements, not every instruction. The **GAUSS** language command **CheckInterrupt** can be used in a **GAUSS** program to check for interrupts and terminate if one is pending.

```
CheckInterrupt;
```

SEE ALSO **GAUSS_CheckInterrupt**, **GAUSS_ClearInterrupt**

## GAUSS_SetLogFile

PURPOSE Sets the file for logged errors.

FORMAT **int GAUSS_SetLogFile( char *_logfn_, char *_mode_ );**

*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 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 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
```
char *GAUSS_SetWorkspaceName( WorkspaceHandle_t *wh, char *name );

newname = GAUSS_SetWorkspaceName( wh, name );
```

INPUT *wh* pointer to a workspace handle.

*name* pointer to the new workspace name.

OUTPUT *newname* pointer to new name, should be considered read only.

REMARKS **GAUSS_SetWorkspaceName** sets the name of a **GAUSS** workspace indicated by a **WorkspaceHandle_t**. The maximum length is 63 characters.

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**

## GAUSS_String

PURPOSE Creates a **String_t** and copies the string data.

FORMAT
```
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:

**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**

## GAUSS_StringAlias

PURPOSE Creates a **String_t**.

FORMAT **String_t *GAUSS_StringAlias( char *_str_ );**

*strdesc* **= GAUSS_StringAlias(** *str* **);**

INPUT *str* pointer to string.

OUTPUT *strdesc* pointer to a string descriptor.

REMARKS **GAUSS_StringAlias** is similar to **GAUSS_String**; however, it sets the *stdata* member of the **String_t** to point to *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 **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 **GAUSS_CopyStringToArg** and **GAUSS_CopyStringToGlobal**.

Free the **String_t** with **GAUSS_FreeString**. It will not free the string data.

SEE ALSO **GAUSS_String**, **GAUSS_StringAliasL**, **GAUSS_StringL**, **GAUSS_FreeString**

## GAUSS_StringAliasL



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 **StringArray_t *GAUSS_StringArray( size_t** *rows*, **size_t** *cols*, **char **** *strs* **);**

*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:

**GAUSS_CopyStringArrayToArg**
**GAUSS_CopyStringArrayToGlobal**
**GAUSS_MoveStringArrayToArg**
**GAUSS_MoveStringArrayToGlobal**

Free the **StringArray_t** with **GAUSS_FreeStringArray**.

SEE ALSO **GAUSS_StringArrayL**, **GAUSS_FreeStringArray**



## GAUSS_StringArrayL

PURPOSE Creates a **StringArray_t** with strings of user-specified length and copies the string array data.

FORMAT **StringArray_t *GAUSS_StringArrayL( size_t** *rows*, **size_t** *cols*, **char ****_strs_, **size_t ***_lens_ **);**

*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:

**GAUSS_CopyStringArrayToArg**
**GAUSS_CopyStringArrayToGlobal**
**GAUSS_MoveStringArrayToArg**
**GAUSS_MoveStringArrayToGlobal**

You can free the **StringArray_t** with **GAUSS_FreeStringArray**.

SEE ALSO **GAUSS_StringArray**, **GAUSS_FreeStringArray**

# GAUSS_StringL

PURPOSE Creates a **String_t** with string of user-specified length and copies the string data.

FORMAT
```
String_t *GAUSS_StringL( char *str, int len );

strdesc = GAUSS_StringL( str, len );
```

INPUT
*str* pointer to string.
*len* length of string, including null terminator.

OUTPUT *strdesc* pointer to a string descriptor.

REMARKS **GAUSS_StringL** **malloc**'s 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:

**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**

## GAUSS_TranslateDataloopFile

PURPOSE Translates a dataloop file.

FORMAT **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**

# Structure Reference 11

## Array_t

PURPOSE N-dimensional array descriptor structure.

FORMAT An **Array_t** is a structure with the following members:

```
double *        adata;
size_t          dims;
size_t          nelems;
int             complex;
```

*adata* pointer to array.

*dims* number of dimensions.

*nelems* number of elements in real part of array.

*complex* 0 if array is real, 1 if complex.

REMARKS An **Array_t** is used to hold the information for an array. To create an **Array_t**, use one of the following functions:

**GAUSS_ComplexArray**
**GAUSS_ComplexArrayAlias**
**GAUSS_Array**
**GAUSS_ArrayAlias**

The structure member *adata* points to a block of memory containing two sections in the case of a real array or three sections in the case of a complex array. The first section, which is the vector of orders for the array, contains *dims* doubles. The second section contains the real part of the array. The optional third section contains the imaginary part. The number of doubles in the real section is the product of the vector of orders and is contained in *nelems*. The number of doubles in the imaginary section is the same as the real section. These three sections are laid out contiguously in memory.

Use **GAUSS_FreeArray** to free an **Array_t**.

SEE ALSO **GAUSS_Array**, **GAUSS_ArrayAlias**, **GAUSS_ComplexArray**, **GAUSS_ComplexArrayAlias**, **GAUSS_FreeArray**

## GAUSS_MatrixInfo_t

PURPOSE 2-dimensional matrix info descriptor structure.

FORMAT A **GAUSS_MatrixInfo_t** is a structure with the following members:

```
size_t      rows;
size_t      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**

## Matrix_t



PURPOSE 2-dimensional matrix descriptor structure.

FORMAT A **Matrix_t** is a structure with the following members:

```
double *        mdata;
size_t          rows;
size_t          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 **GAUSS 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**

## StringArray_t



PURPOSE String array descriptor structure.

FORMAT A **StringArray_t** is a structure with the following members:

```
StringElement_t * table;
size_t            rows;
size_t            cols;
size_t            baseoffset;
```

*table* pointer to an array of string element descriptors.

*rows* number of rows.

*cols* number of columns.

*baseoffset* offset of base of memory block containing strings.

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 **GAUSS 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_t

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

## I



## M



## P



## R



## S



## V



## W