# Sage DLL Application Notes

D. Gedeon
Gedeon Associates

March 31, 2015

## Contents



# 1 DLL Overview

Sage is the name of a Windows application for modeling stirling-cycle machines and related thermodynamic cycles that has been sold commercially since 1995. A frequent request among users has been for a way to run Sage automatically from another computer program rather than by a human operator.

A Sage DLL (dynamic link library) provides a dynamic interface whereby another program running under the Windows operating system can dynamically access Sage functionality to open existing Sage model files, modify inputs, read outputs, etc. Built-in logic ensures that the Sage solution is always up to date when reading outputs. There are three Sage DLL classes corresponding to the Stirling, Pulse-Tube and Low-T Cooler model classes. They are essentially identical except for the OpenModel, sageOpenModel and sageNewModel functions which open or create a different model class in each case.

As of Sage version 10 there are both 32 and 64 bit DLLs available for use with application running under the 32 and 64 bit Windows operating systems (Win32 and Win64). The 64 bit DLLs have the suffix '64' appended to the name:

| **Win32 DLL** | **Win64 DLL** | **Opens Model Class** | **File Extension** |
|---|---|---|---|
| StirlingLib.dll | StirlingLib64.dll | Stirling-Cycle | .stl |
| PtubeLib.dll | PtubeLib64.dll | Pulse-Tube Cooler | .ptb |
| LTCoolerLib.dll | LTCoolerLib64.dll | Low-T Cooler | .ltc |

Sample applications TestStirlingLib.exe, TestPtubeLib.exe, TestLTCoolerLib.exe, demonstrate how to use the Sage DLLs (*see* Testing the DLL). These application were written under the Borland Delphi (Pascal language) programming environment, as were the Sage DLLs and the Sage application itself.

**Differences between Win32 and Win64** Both 32 and 64 bit DLLs share the same model file format so you can use a 32 bit DLL to read a model file produced by a 64 bit DLL and vice-versa.

There are some differences though in the function arguments supplied and values returned. Obviously, pointer types are 64 bit values (8 bytes) under Win64 compared to 32 bit (4 bytes) under Win32. Not so obviously, Microsoft depreciated the 80-bit extended-precision floating-point format to 64-bit double-precision format under Win64. The Win64 Delphi compiler (used for Sage) went along with this decision and dropped support for 80-bit extended precision by re-declaring the Extended data type to Double when compiling for the Win64 platform. The result is that the solution precision drops from about 18 significant figures under the Win32 DLLs to about 15 under the Win64 DLLs.

It may be that Intel will eventually abandon the 80-bit part of their processor floating-point unit and replace it with a 128-bit "quad" precision floating-point unit. If this ever happens a future 128 bit Windows (Win128) might support a 128-bit Quad floating-point format.

## 1.1 Installation

The Sage DLLs and related files are installed on your computer in the usual way, by running the Windows Control Panel Add/Remove programs utility with the distribution CD ROM in the appropriate drive or by double-clicking the

installation program (e.g. setup.exe) from Windows Exporer. Files are installed in the DLL subdirectory under the Sage installation director (e.g. c:\Program Files\Gedeon\Sage7\DLL\). You may access the DLL files from the installed location or copy them to another location on your computer as required.

## 1.2 Exception Handling — Pop-Up Messages vs Silent

Sage DLLs always handle software exceptions that occur within the DLL. When an exception occurs, some DLL functions pop-up a message dialog window that notifies you of the problem. This temporarily halts program execution until you manually click the OK button. Other DLL functions — those beginning with the sage prefix — handle exceptions silently by returning a nonzero value for a Status integer that you pass to the function as an argument. When Status returns nonzero you can read the error message by calling the sageGetLastErrorMsg function. This allows the calling program to deal with exceptions without the need for manual button clicking.

The preceding paragraph applies only to exceptions that float up all the way to the DLL interface level. Some exceptions are resolved internally at a relatively low level in the Sage code (many subroutines below the DLL interface) by popping-up dialogs that give the user a choice of how to resolve the exception. For example, to permit extrapolating a thermophysical property beyond its tabulated values or to ignore a divide-by-zero error in a user-defined expression. If you encounter one of these exceptions you will have to manually click a message-dialog button before control returns to your program. However if you are willing to risk the consequences you can override this behavior with the SetIgnoreExprError and SetIgnoreRangeError functions, in effect pre-ignoring such exceptions in advance.

## 1.3 AccessForm Window

Sage DLLs work with models using a higher-level data structure known as an AccessForm. The AccessForm has a visual presence that pops up in the form of a small window whenever you open a Sage model (from a disk file) or create a new model. The AccessForm window looks like this:


You do not have to pay any attention to the AccessForm window or press any buttons to close it. It just sits there quietly to confirm that the DLL active.

## 1.4 Solve and Optimize Status Dialogs

To provide feedback and also give you a means to cancel the solving or optimizing process the Sage DLLs show the same status dialogs as the Sage GUI whenever Sage is solving or optimizing a model. After solving or optimizing finishes the AccessForm closes the dialogs (hides it from view) and returns control to your program.

## 1.5 Calling Convention

All Win32 DLL functions use the stdcall calling convention, which means that parameters are passed in a stack frame, in right-to-left order and that the DLL cleans up the stack (resets the stack pointer). Your compiler of choice may require an explicit directive to adhere to this calling convention. Here is a list of directives that Borland compilers adhere to:

| Directive | Parameter order | Clean-up | Passes parameters in registers? |
|---|---|---|---|
| register | Left-to-right | Routine | Yes |
| pascal | Left-to-right | Routine | No |
| cdecl | Right-to-left | Caller | No |
| stdcall | Right-to-left | Routine | No |
| safecall | Right-to-left | Routine | No |

The cdecl convention is likely the default directive for C or C++ compilers. In Windows, the operating system APIs use stdcall and safecall. Other operating systems generally use cdecl.

Under Win64 there is only one calling convention so your compiler of choice will likely ignore any calling convention directives.

## 1.6 Variable Types

In this document and in the Delphi-Pascal interface files the types of function arguments and returned values are named according to the following table:

| Type Name | purpose | stored as (Win32/Win64) |
|---|---|---|
| THandle | Windows handle | 32/64 bit pointer |
| NativeInt | pointer | 32/64 bit pointer |
| PAnsiChar | pointer to null-terminated ANSI string | 32/64 bit pointer |
| Integer | integer value | 32 bit signed integer |
| Double | real value | 8-byte floating point |
| Extended | real value | 10-byte/8-byte floating point |
| Boolean | True or False value | byte |

The var declaration before a function argument means that the argument is passed by reference, instead of by value. In other words the function expects the address of some variable, rather than its value.

An ANSI string is a null terminated string of 8-bit ANSI characters. ANSI strings used to be the standard format for all Windows applications but some programs are now changing to 16-bit unicode character strings. The Sage DLL functions require ANSI strings for backward compatibility with earlier versions. Passing a pointer to a unicode string to a Sage DLL function will cause trouble. Instead convert it to an ANSI string first and then pass a pointer to the converted string to the DLL function. The Windows system function MultiByteToWideChar converts unicode to ansi strings. The application you are using to call the Sage DLL may have such a function as well.

## 1.7 Stream Identifiers

Sage maintains a list of unique integer constants, known as stream identifiers, for each class type that can be saved to a disk file. When loading a model from a disk file, these identifiers enable Sage to figure out what class type it is reading and call the appropriate load method to read its data. Several DLL functions take stream identifiers as arguments or return stream identifier values. Names of stream identifier arguments generally begin with the prefix ASid.

For purposes of using the DLL the stream identifiers you are concerned with are those for model components, connectors and variables. A complete list of such stream identifiers is found in the disk files:

| class types | stream-identifier files |
|---|---|
| model components | Stl.sid, Ptb.sid, Ltc.sid |
| connectors | Cnctobj.sid |
| variables | Sage.sid |

These files are included in the DLL distribution. You will find that the model-component stream-identifier files contain several *extra* sub-identifiers, indented under each model-component identifier. These are used to select the functions that normalize and evaluate the model-component's variables and are not relevant to any DLL functions

Stream identifiers are specified as offsets to base values. For example a *tubular canister* has stream identifier sid_TubCan = sid_Mdl + 1410, where sid_Mdl is the base value. Stream-identifier base values are:

| base values | |
|---|---|
| sid_Cnct | 512 |
| sid_Mdl | 1024 |

## 1.8 Interface files

In order for your program to link to the Sage DLL you generally must provide some sort of interface file that provides necessary information about the structure of the DLL. The format of the interface file depends on the compiler or programming environment being used to access the DLL.

For the stirling DLL, two sample interface files are provided with the DLL distribution.

| sample StirlingLib interface files | |
|---|---|
| StirlingInterface.pas | Delphi (Pascal) format |
| StirlingLib.h | C format |

The first is in Delphi Pascal format and is known to be correct. In the unlikely event you are using the Delphi compiler you can directly reference StirlingInterface.pas in your programs uses statement and be done with it. The second file is in what is believed to be Borland C format but has never been tested, so beware.

For the pulse-tube and Low-T Cooler DLL's the only interface files provided are the Delphi-version interface files PtubeInterface.pas and LTCoolerInterface.pas. These are essentially identical to the stirling interface file, except for exporting functions from PTubeLib.dll or LTCoolerLib.dll instead of StirlingLib.dll.

### 1.8.1 Delphi Format

In the Delphi format the interface file is a so-called *unit* file, containing a declaration, interface section and implementation section. For example, a minimalist Delphi interface file for only the single DLL function sageSetRealVal would look like this.

```
unit StirlingInterface;

interface

procedure sageSetRealVal(AAccessForm: NativeInt;
    AV: NativeInt;
    AValue: Double;
    var Status: Integer); stdcall;

implementation

procedure sageSetRealVal; external 'StirlingLib.dll';

end.
```

### 1.8.2 C Format

The same minimalist interface in C format is just a single statement in a header file that might look like this:

```
void _declspec(dllexport) _stdcall sageSetRealVal(
    long AAccessForm,
    long AV,
    double AValue,
    int* Status);
```

# 2 Available Functions

In the first Sage DLL version all functions employed pop-up exception notification. When silent exception handling was introduced (September 05) the pre-existing functions were retained for backwards compatibility rather than replaced with equivalent silent-exception functions. As a result there are often two DLL functions that do the same thing, apart from exception handling. The only difference in their names is the sage prefix.

Functions whose names begin with sage... all employ silent exception handling. The last argument is always an integer-valued Status variable passed by reference. If the function returns with Status set non-zero then some sort of error condition occured with the error message available as the returned result of the sageGetLastErrorMsg function. Currently 1 is the only non-zero value returned in Status although future versions may use different values to denote different error conditions.

**sageGetLastErrorMsg**

| *Arguments:* | *Returns:* |
|---|---|
| none | PAnsiChar |

Returns pointer to the last error message, a null-terminated string within the DLL. The calling routine need not allocate any memory for the string, except as needed to assign the string to a local copy. The string starts out as a valid but empty string and is updated whenever there is an error or exception encountered in a DLL function that implements silent exception handling. The DLL manages string memory internally, cleaning up memory used for old strings when updated by new ones. The function returns a pointer to a valid string whenever the DLL is loaded.

## 2.1 Sage Version

**GetVersion**

| *Arguments:* | *Returns:* |
|---|---|
| none | PAnsiChar |

Returns the Sage version number corresponding to the DLL.

## 2.2 Functions for Pop-Up Dialog Control

Normally on encountering certain types of math errors during the solution process the DLL suspends program execution and pops-up a message dialog giving you the option to ignore the error by clicking an *ignore* button. The following functions allow you to override this behavior, effectively ignoring these math errors in advance so that no error messages are generated, no message dialog pops up and execution proceeds with default exception handling. Choosing to

ignore exceptions should be done with care only after you are reasonably certain that the default exception handling will not adversely affect the accuracy of the Sage solution.

**SetIgnoreExprError**

| *Arguments:* | *Returns:* |
|---|---|
| AIgnoreExprError: Boolean | none |

Sets the way math errors are handled when evaluating user-defined expressions. Pass AIgnoreExprError = True to ignore such exceptions in advance, allowing the solution process to continue unimpeded. Pass AIgnoreExprError = False to restore the default pop-up dialog behavior.

Exceptions of this type are often divide-by-zero errors as a result of a variable that has been temporarily initialized to zero in the denominator of some expression. This type of error generally clears up as the solution process evolves. If this is the only type of expression error to be found in your model it is appropriate to ignore it. When you ignore the error the default exception handling generally sets the expression returned result to zero. Except for expression errors in recast inputs where the returned result is set to the most recent valid value.

**GetIgnoreExprError**

| *Arguments:* | *Returns:* |
|---|---|
| none | Boolean |

Returns the math exception handing behavior for user-defined expressions. See SetIgnoreExprError

**SetIgnoreRangeError**

| *Arguments:* | *Returns:* |
|---|---|
| AIgnoreRangeError: Boolean | none |

Sets the way out-of-range extrapolation is handled when evaluating cubic spline variables or material properties. Pass AIgnoreRangeError = True to ignore out-of-range extrapolation in advance. Pass AIgnoreRangeError to False to restore the default pop-up dialog behavior. This function must be called prior to opening a model or creating new model components. If you subsequently change the *IgnoreRangeError* value you need to re-load the Sage model before it takes effect. That is because the model components have local values of *IgnoreRangeError* designed to reflect what button you press in the pop-up dialog. That local value is assigned from the DLL value only on load and on creation.

Exceptions of this type are often the result of transient solved temperature values temporarily outside the range of tabulated thermophysical properties. If this is the only type of error likely in your model then it is appropriate to ignore

it. If instead you are consistently modeling well above or below the range of tabulated values a better strategy in the long run might be to increase the data range of the tabulated data points that are causing the problem. When you ignore the error the default exception handling extrapolates the data using the cubic spline function fit to the nearest data points. Errors can grow quickly for temperatures increasingly outside the range of tabulated data.

**GetIgnoreRangeError**

| *Arguments:* | *Returns:* |
|---|---|
| none | Boolean |

Returns the out-of-range interpolation handing behavior assigned to loaded or newly created model components. See SetIgnoreRangeError

**SetIgnoreDemagError**

| *Arguments:* | *Returns:* |
|---|---|
| AIgnoreDemagError: Boolean | none |

Sets the way demagnetization warnings are handled when the magnetic field applied to a permanent magnet exceeds the coercive force limit. Pass AIgnoreDemagError = True to ignore the warning in advance. Pass AIgnoreDemagError to False to restore the default pop-up dialog behavior. This function must be called prior to opening a model or creating new model components. If you subsequently change the *IgnoreDemagError* value you need to re-load the Sage model before it takes effect. That is because the model components have local values of *IgnoreDemagError* designed to reflect what button you press in the pop-up dialog. That local value is assigned from the DLL value only on load and on creation.

**GetIgnoreDemagError**

| *Arguments:* | *Returns:* |
|---|---|
| none | Boolean |

Returns the out-of-range interpolation handing behavior assigned to loaded or newly created model components. See SetIgnoreDemagError

**SetIgnoreOverfilledError**

| *Arguments:* | *Returns:* |
|---|---|
| AIgnoreOverfilledError: Boolean | none |

Sets the way overfilled-coil warnings are handled when the coil volume of an embedded moving coil exceeds the available volume of the magnetic gap it is moving in. Pass AIgnoreOverfilledError = True to ignore the warning in advance. Pass

AIgnoreOverfilledError to False to restore the default pop-up dialog behavior. This function must be called prior to opening a model or creating new model components. If you subsequently change the *IgnoreDemagError* value you need to re-load the Sage model before it takes effect. That is because the model components have local values of *IgnoreDemagError* designed to reflect what button you press in the pop-up dialog. That local value is assigned from the DLL value only on load and on creation.

**GetIgnoreOverfilledError**

| *Arguments:* | *Returns:* |
|---|---|
| none | Boolean |

Returns the out-of-range interpolation handing behavior assigned to loaded or newly created model components. See SetIgnoreOverfilledError

## 2.3 Functions for Sage Option Settings

The Sage GUI has two "Options" dialogs, one for Sage general options and one for model-class options. These options are saved and later reloaded from initialization files. Sage loads the model-class options when loading the associated model input file from either the GUI or DLL. Sage loads the general options as one of the initialization tasks when the GUI begins execution. But not when you load the DLL, because there is no corresponding initialization code in the DLL. So the DLL normally runs with the default general options.

The functions below allow you to read the values of most Sage options and change them. There are functions for all of the general options and most of the model-class options, except for changing the model-class dimensional units individually ("dimensions" tab). To change those, open the model under the GUI, change the dimensions in the model-class options dialog and save the model. Or you can try directly changing the values in the model initialization file (*.sin, *.pin, *.lin). There is one function available to change dimensional units all at once. The SetDefaultDims function restores all dimensional units to SI dimensions.

**SetDefaultDims**

| *Arguments:* | *Returns:* |
|---|---|
| none | none |

Sets DLL dimensional units to default SI dimensions. The Get/SetRealVal and Get/SetRealPart functions operate in the current dimensional units of unit SiUnits which are loaded from a model-specific initialization file during the OpenModel process. Calling this function after OpenModel will reset the dimensional units in SiUnits to default SI values.

**SetErrScale**

| *Arguments:* | *Returns:* |
|---|---|
| AErrScale: Integer | none |

Sets solver error tolerance scale-factor index. Valid range is (0..8), corresponding to increasing scale factors. Other values will be truncated to the nearest valid value. The value corresponds to the sliding scale in the Options|Sage|Solver|Convergence Tolerance dialog box in the graphical interface.

**GetErrScale**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the solver error tolerance scale-factor index. See SetErrScale.

**SetMaxTerribIter**

| *Arguments:* | *Returns:* |
|---|---|
| AMaxTerribIter: Integer | none |

Sets the maximum number of terrible-progress solver iterations to be endured before abandoning the solve process. The default is 5. A "terrible-progress iteration" is one where the solution convergence error has either been increasing or not decreasing fast enough for a number of consecutive iterations. A value of 1 ensures at least one terrible-progress iteration is allowed which gives at least one finite-difference interval update before throwing in the towel.

**GetMaxTerribIter**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the "MaxTerribIter" value. See SetMaxTerribIter.

**SetMaxTotalIter**

| *Arguments:* | *Returns:* |
|---|---|
| AMaxTotalIter: Integer | none |

Sets the maximum number of total solver iterations allowed before abandoning the solve process. The default is 50.

**GetMaxTotalIter**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the "MaxTotalIter" value. See SetMaxTotalIter.

**SetOptMaxIter**

| *Arguments:* | *Returns:* |
|---|---|
| AOptMaxIter: Integer | none |

Sets the maximum number of iterations allowed during optimization. After this many iterations the optimizer returns, with the values of optimized model variables at their values for the final iteration. The default is 100.

**GetOptMaxIter**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the "OptMaxIter" value. See SetOptMaxIter

**SetOptMaxStepSolveAttempts**

| *Arguments:* | *Returns:* |
|---|---|
| AOptMaxStepSolveAttempts: Integer | none |

Sets the maximum number of failed solve attempts tolerated during the optimization line-search process. During the line search process the optimizer takes a step of some length in the search direction (increments optimized variables) and attempts to solve the model at that point. If the solution fails the optimizer reduces the step length and tries again. If the solver fails after this many attempts then the optimizer gives up and returns, with the values of optimized model variables at their values for the last successful iteration. The default is 5.

**GetOptMaxStepSolveAttempts**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the "OptMaxStepSolveAttempts" value. See SetOptMaxStepSolveAttempts

**SetDFScale**

| *Arguments:* | *Returns:* |
|---|---|
| ADFScale: Integer | none |

Sets the relative step-change limit index for for certain key solved variables. Valid range is (0..8), corresponding to increasing step sizes allowed per solver iteration. Other values will be truncated to the nearest valid value. The value corresponds to the sliding scale in the Options|Sage|Solver|Allowed change per

step dialog box in the graphical interface. Smaller values produce slower convergence that may be more stable. Larger values produce faster convergence that may be less stable.

**GetDFScale**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the relative step change limit index. See SetDFScale.

**SetDisplayEnumDetail**

| *Arguments:* | *Returns:* |
|---|---|
| ADisplayEnumDetail: Integer | none |

Sets the enumerated variable display detail level that will appear in listing files. The value must be 0 for "name only" or 1 for "full detail". Other values will be truncated to the nearest valid value. Enumerated variables are those selected by name from a list. For example, the "Gas" variable of a stirling model, containing a large number of numeric property values (the details).

**GetDisplayEnumDetail**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the enumerated variable display detail level. See SetDisplayEnumDetail.

**SetDisplaySigFigs**

| *Arguments:* | *Returns:* |
|---|---|
| ADisplaySigFigs: Integer | none |

Sets the number of significant figures for floating-point outputs that appear in listing files. Values between 3 and 12 are reasonable. This function has no affect on Sage's internal numerical calculations nor on DLL functions like GetRealVal that retrieve floating-point variable values. Sage employs extended precision variables for its internal calculations but final listing outputs are generally the result of hundreds of calculations with cumulative round-off and finite-difference truncation error, the result of which is to reduce precision by several orders of magnitude.

**GetDisplaySigFigs**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the number of significant figures for floating-point outputs. See SetDisplaySigFigs.

**SetGridInterpOrder**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AGridInterpOrder: Integer | |

Sets the computational grid x-interpolation order (axial direction) for non-solved grid variables in the model within AAccessForm. The value must be 0 for "linear" interpolation or 1 for "cubic" interpolation. Other values will be truncated to the nearest valid value. Because it affects a model's solved values this setting is saved in the initialization file corresponding to the model data file and restored on re-loading the model. The grid interpolation order must not be set during processing.

**GetGridInterpOrder**

| *Arguments:* | *Returns:* |
|---|---|
| none | Integer |

Returns the computational grid x-interpolation order. See SetGridInterpOrder.

**SetGasFileName**

| *Arguments:* | *Returns:* |
|---|---|
| AGasFileName: PAnsiChar | none |

Sets the name of the database file from which gas property data will be read. AGasFileName must be a fully-qualified file name (including directory path).

**GetGasFileName**

| *Arguments:* | *Returns:* |
|---|---|
| none | PAnsiChar |

Returns the fully-qualified file name from which gas property data will be read. See SetGasFileName.

**SetSolidFileName**

| *Arguments:* | *Returns:* |
|---|---|
| ASolidFileName: PAnsiChar | none |

Sets the name of the database file from which solid property data will be read. ASolidFileName must be a fully-qualified file name (including directory path).

**GetSolidFileName**

| *Arguments:* | *Returns:* |
|---|---|
| none | PAnsiChar |

Returns the fully-qualified file name from which solid property data will be read. See SetSolidFileName.

## 2.4 Functions for Model-Class I/O

These functions deal with the opening and closing of an exist Sage model file or creating a new model data structure from scratch. You may start from a Sage model file previously created with the usual Sage application interface (Stirling, Ptube, LowTCooler) or create a model entirely using DLL functions.

**OpenModel**

| *Arguments:* | *Returns:* |
|---|---|
| AHandle: THandle | NativeInt |
| ASageFileName: PAnsiChar | |

Opens a Sage model file (.stl, .ptb, .ltc file extension, depending on DLL) for subsequent use. Generally the first function of the DLL to be called. OpenModel requires two arguments AHandle and ASageFileName. ASageFileName is the fully-qualified file name of the model file to be opened (example: `c:\ADirectory\GenericFPSE.stl`) . This name may be hard-wired into the application or provided as the returned result of a file-open dialog. The sample program TestStirlingLib.exe uses the latter approach. AHandle is the Windows handle (pointer) of the calling application (Delphi example: Application.Handle) and has the effect of the calling application owning the Sage access form that will pop up so that both respond to Windows commands (e.g. minimized) together. Passing 0 for the AHandle argument appears to work without any adverse affects. This would be the thing to do if your application has no visual presence (window) associated with it. The returned result of the call to OpenModel is a pointer to the Sage access form that provides the functionality for most of the other functions of the DLL. The calling application must save this pointer (e.g. in a variable AccessForm) generally to be passed as the first argument to the other functions of the DLL.

**sageOpenModel**

| *Arguments:* | *Returns:* |
|---|---|
| AHandle: THandle | NativeInt |
| ASageFileName: PAnsiChar | |
| var Status: Integer | |

Like OpenModel except with silent exception handling. Returns Status = 0 if successful else Status = 1 with an error message returned by the sageGetLastErrorMsg function.

**sageNewModel**

| *Arguments:* | *Returns:* |
|---|---|
| AHandle: THandle | NativeInt |
| var Status: Integer | |

Alternative to sageOpenModel which may be called first to create an access form with a new root model component without any child model components. AHandle has the same purpose as in sageOpenModel. Same silent exception handling as sageOpenModel.

**CloseModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |

Closes the model file previously opened. Generally the last function of the DLL to be called. CloseModel requires one argument AAccessForm, which is the returned result of the OpenModel function, as explained above. CloseModel releases the memory used for the Sage access form. Closing the access form by clicking on the x button at the top removes the visual window associated with the access form but does not affect its software data structures. The access-form functionality remains intact until calling CloseModel.

**SaveModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| ASageFileName: PAnsiChar | |

Saves the current model state in the Sage access form to the file ASageFileName, which must be a fully-qualified file name including directory path.

**SaveListing**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AListingFileName: PAnsiChar | |

Saves the listing for the current model state in the Sage access form to the file AListingFileName, which must be a fully-qualified file name including directory path. The resulting file is an ASCII format text file which may be opened with any text editor.

**SaveTaggedVars**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| ALogFileName: PAnsiChar | |

Saves the tagged user-defined variables for the current model in the Sage access form to the file ALogFileName, which must be a fully-qualified file name including directory path. User-defined variables are "tagged" by checking the "write to log file" checkbox during their definition in the standard Sage GUI. The resulting file is an ASCII format text file which may be opened with any text editor.

**SaveSolutionGrid**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AModel: NativeInt | |
| AGridFileName: PAnsiChar | |

Saves solution grid of model component AModel to the file AGridFileName, which must be a fully-qualified file name including directory path. Includes grids of all child components plus their connected connectors.

## 2.5 Functions for Visual Settings

**HideAccessForm**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |

Makes the AccessForm window invisible.

**ShowAccessForm**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |

Makes the AccessForm window visible. The AccessForm window is normally visible (after calling the OpenModel or sageOpenModel functions) but may have been hidden by the HideAccessForm function.

**SetCaptionAccessForm**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| ACaption: PAnsiChar | |

Changes the caption at the top of the AccessForm window to ACaption. Use this function if you want to change the default caption, which is "using AFileName", where AFileName is the file name supplied as an argument to the OpenModel or sageOpenModel functions.

**GetCaptionAccessForm**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |

Returns the caption at the top of the AccessForm window.

## 2.6 Functions for Getting and Setting Variable Values

There are several "set" and "get" routines that can be called any time the Sage access form is open (between calls to OpenModel and CloseModel) to get or set values for Sage variables. It is only possible to set input variables. However it is possible to get any type of variable, input or output. When getting an output variable the model is first solved, if it is not already in a solved state. Once solved, the model remains in a solved state during subsequent get calls so that no further solve processing is required. The model state changes from "solved" to "not solved" as a result of setting an input variable. After that the next "Get" call will initiate the solving process, and so forth.

As of Sage version 5.5 (12-07) the DLL implements a quit-after-effort policy under which the solve process gives up when further effort toward convergence seems hopeless (*see* function sageSolveModel). In this case those functions that implement silent exception handling (functions with *sage* prefix) return Status = 1 and an appropriate message for retrieval by the sageGetLastErrorMsg function, giving the calling program the option to deal with the problem. Those functions that do not implement silent exception handling just pop-up a message dialog and return control to the calling program after the user presses the OK button, along with a zero result for the solved variable. Prior to version 5.5 the DLL solve process would continue iterating forever unless the user pressed the stop button in the solution status dialog.

Some functions in this section select the variable to "set" or "get" from its name identifier and the name identifier of the model component in which it resides. Other functions require a direct pointer to the variable. Variable

pointers may be obtained using the functions documented below (*see* Functions for Managing Model Components and Variables).

**SetRealVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |
| AValue: Double | |

Sets a real-valued (floating point) input to the value passed in the AValue parameter. Operates using the dimensional units currently selected in unit SiUnits (default SI units unless reset by LoadState or model-options dialog). Parameters AMdlName and AVarName are the model-component and identifier names of the variable to be changed within the Sage model. For example 'displacer', 'Xamp'. These are the same names that appear in the Sage GUI display window or listing. The model-component name can be easily customized using the normal Sage application interface. Argument AAccessForm is the returned result of the OpenModel call.

**sageSetRealVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AV: NativeInt | |
| AValue: Double | |
| x var Status: Integer | |

Like SetRealVal except selects the Sage variable whose value will be set by pointer rather than by name and has silent exception handling. Sets the value of the real variable AV to AValue in current dimensional units.

**GetRealVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Double |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |

Returns the value of a real-valued variable in current dimensional units. The parameters have the same meaning as in SetRealVal. Works with built-in inputs and outputs as well as user-defined variables.

**sageGetRealVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Double |
| AV: NativeInt | |
| var Status: Integer | |

Like GetRealVal except selects the Sage variable whose value will be returned by pointer rather than by name and has silent exception handling. Returns the value of the real variable AV in current dimensional units.

**SetExtendedVal**
**sageSetExtendedVal**
**GetExtendedVal**
**sageGetExtendedVal**

Under Win32 DLLs these functions differ from SetRealVal, sageSetRealVal, GetRealVal, sageGetRealVal because the AValue argument or returned result is in extended precision (Extended type) instead of double precision (Double type). Extended-precision values are accurate to about 18 significant figures (in decimal notation) while double precision values are accurate to about 15 figures. Internally, Sage calculates everything in extended precision but the only time it is important to do so is for finite-difference derivative approximations. If your application needs to do the same then these functions are available. Otherwise the double-precision equivalents are probably sufficiently accurate.

Under Win64 DLLs these functions are equivalent to SetRealVal, sageSetRealVal, GetRealVal, sageGetRealVal because the Extended and Double types are the same.

**SetUserVar**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |
| AParseString: PAnsiChar | |

Sets the defining expression for a user variable to the value encoded in AParseString. Arguments AAccessForm, AMdlName and AVarName have the same meanings as in SetRealVal.

**sageSetUserVar**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AV: NativeInt | |
| AParseString: PAnsiChar | |
| var Status: Integer | |

Like SetUserVar except selects the Sage variable whose value will be set by pointer rather than by name and has silent exception handling.

**GetUserVarVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Double |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |

Returns the real value (result of evaluating the defining expression) of the user-defined variable named AVarName in the model named AMdlName. GetRealVal does the same thing and also works for built-in real-valued variables.

**sageGetUserVarVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Double |
| AV: NativeInt | |
| var Status: Integer | |

Like GetUserVarVal except selects the Sage variable whose value will be returned by pointer rather than by name and has silent exception handling.

**sageSetIntegerVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AV: NativeInt | |
| AValue: Integer | |
| var Status: Integer | |

Sets the value of the integer variable AV to AValue.

**sageGetIntegerVal**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AV: NativeInt | |
| var Status: Integer | |

Returns the value of the Integer variable AV.

**SetRealPart**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |
| AQualifier: PAnsiChar | |
| AValue: Double | |

Sets a real-part of a multi-valued input variable (e.g. Fourier series). Arguments AAccessForm, AMdlName and AVarName have the same meanings as in SetRealVal. Argument AQualifier is the qualifier string used to select the appropriate part to set. For example 'Mean' or 'Amp.1' or 'FData.2', depending on the type of variable. Function arguments such as '(0.5)' are not recognized in qualifier strings. Operates using the dimensional units currently selected in unit SiUnits (default SI units unless reset by LoadState or model-options dialog). Although SetRealVal is more direct for a simple single-valued variable, this procedure will also work. In that case the AQualifier string must be passed as nil.

**Setting Fourier series components** The recommended way to set values for Fourier series harmonics is by assigning the sine and cosine coefficients, rather than amplitude and phase. Say you know the amplitude $r_n$ and phase $\theta_n$ of the $n$-th harmonic. Rather than directly setting amp.n = $r_n$ and arg.n = $\theta_n$ it is more reliable to set cosine and sine components using the identities

$$\begin{aligned} \text{cos.n} &\equiv r_n cos(\theta_n) \\ \text{sin.n} &\equiv -r_n sin(\theta_n) \end{aligned}$$

The reason is that the actual data Sage stores for Fourier Series objects are the $a_n$ and $b_n$ arrays of cosine and sine coefficients. Amplitude and phase are calculated on the fly when needed. Setting amplitude and phase involves a conversion and there are two problems. The first is that phase is indeterminate when amplitude is zero. So setting arg.n when the current amp.n is zero results in the values $a_n$ and $b_n$ both set to zero which effectively erases the phase information. So you should set amplitude before phase to avoid this problem. The second is the case when amp.n is negative. Setting amp.n prior to setting arg.n results in the phase getting correctly shifted by 180 degrees but that information is lost on setting arg.n. So for this case you should set phase prior to amplitude. Rather than remember these possibilities it is easy to set cosine and sine coefficients directly, as above, which is always reliable.

**sageSetRealPart**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AV: NativeInt | |
| AQualifier: PAnsiChar | |
| AValue: Double | |
| var Status: Integer | |

Like SetRealPart except selects the Sage variable whose value will be set by pointer rather than by name and has silent exception handling.

**GetRealPart**

*Arguments:*
AAccessForm: NativeInt
AMdlName: PAnsiChar
AVarName: PAnsiChar
AQualifier: PAnsiChar

*Returns:*
Double

Returns a real-part of a multi-valued variable in current dimensional units. The parameters have the same meaning as in SetRealPart except that AQualifier is interpreted more generally. It is not limited to being a simple identifier (like 'Mean') or identifier + subqualifier (like 'Amp.1'). AQualifier can also include the argument list needed for referencing certain properties of gas or solid variables (like 'Vsound(300)'). It can also include arguments that are themselves expressions (like 'Vsound(Gas.T0)'). In fact the only requirement is that the string 'AVarName.AQualifier' is a valid expression of the sort that might be used in a user-defined variable.

**sageGetRealPart**

*Arguments:*
AAccessForm: NativeInt
AV: NativeInt
AQualifier: PAnsiChar
var Status: Integer

*Returns:*
Double

Like GetRealPart except selects the Sage variable whose value will be returned by pointer rather than by name and has silent exception handling.

**SetExtendedPart**
**sageSetExtendedPart**
**GetExtendedPart**
**sageGetExtendedPart**

Like SetRealPart, sageSetRealPart, GetRealPart, sageGetRealPart except the AValue argument or returned result is in extended precision (Extended type) instead of double precision (Double type). This is only relevant for Win32 DLLs. For Win64 DLLs the Extended and Double types are identical.

**sageSetPairsCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| APairsVar: NativeInt | |
| ACount: Integer | |
| var Status: Integer | |

Re-initializes the number of data pairs within data-pairs or cubic-spline variable APairsVar. Any pre-existing data values are lost. Why would you want to do this? In the GUI a user adds interpolation pairs to an input like "Tinit" (temperature-distribution interpolation pairs) by clicking the "add" button in the input specification dialog. The GUI replaces the old TPairs structure with a new one containing the correct number of data pairs. sageSetPairsCount allows you to do this from the DLL. It is up to you to save any old data values and re-assign them into the new pairs as needed using the Get/SetRealPart functions.

**sageGetPairsCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| APairsVar: NativeInt | |
| var Status: Integer | |

Returns the number of data pairs within data-pairs or cubic-spline variable APairsVar. Use sageGetRealPart and sageSetRealPart to get or set independent and dependent data values using qualifiers 'TData.n', FData.n', for n = 1 to Count. For cubic splines beware that TData values must be in strictly increasing order and are sometimes restricted to the range [0, 1].

**sageSetFSeriesCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AFSeriesVar: NativeInt | |
| ACount: Integer | |
| var Status: Integer | |

Re-initializes the number of terms within Fourier-series input variable AFSeries-Var. Any pre-existing term data values are lost. This is useful as the first step in entering an input Fourier Series containing more terms than the current value. See comments under the similar function sageSetPairsCount.

**sageGetFSeriesCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AFSeriesVar: NativeInt | |
| var Status: Integer | |

Returns the number of terms within Fourier-series variable AFSeriesVar. Use sageGetRealPart and sageSetRealPart to get or set terms using qualifiers like 'mean', 'cos.n', 'sin.n', 'amp.n', 'arg.n', for n = 1 to Count

**SetGasName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |
| AGasName: PAnsiChar | |

Sets the value of the gas variable named AVarName in the model named AMdlName to the gas with the name AGasName in the database file established by the SetGasFileName procedure.

**sageSetGasName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AV: NativeInt | |
| AGasName: PAnsiChar | |
| var Status: Integer | |

Like SetGasName except selects the gas variable whose name value will be set by pointer rather than by name and has silent exception handling.

**GetGasName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |

Returns the gas identifier name for the gas variable named AVarName in the model named AMdlName.

**sageGetGasName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |
| AV: NativeInt | |
| var Status: Integer | |

Like GetGasName except selects the gas variable whose name value will be returned by pointer rather than by name and has silent exception handling.

**SetSolidName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |
| ASolidName: PAnsiChar | |

Sets the value of the solid variable named AVarName in the model named AMdlName to the solid with the name ASolidName in the database file established by the SetSolidFileName procedure.

**sageSetSolidName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AV: NativeInt | |
| ASolidName: PAnsiChar | |
| var Status: Integer | |

Like SetSolidName except selects the solid variable whose name value will be set by pointer rather than by name and has silent exception handling.

**GetSolidName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |
| AMdlName: PAnsiChar | |
| AVarName: PAnsiChar | |

Returns the solid identifier name for the solid variable named AVarName in the model named AMdlName.

**sageGetSolidName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |
| AV: NativeInt | |
| var Status: Integer | |

Like GetSolidName except selects the solid variable whose name value will be returned by pointer rather than by name and has silent exception handling.

## 2.7 Functions for Model Processing

These functions can be used to solve or optimize the model. Normally solving is an automatic part of getting a variable value but there is a sageSolveModel function available anyway, in case you need it.

**ReinitializeModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |

Reinitializes the model in AAccessForm. This is useful if you have just changed a variable value, such as the working gas, which throws an existing solution into a state from which the solver cannot recover.

**sageSolveModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| var Status: Integer | |

Solves the model (invokes the Sage solver to iterate implicit model variables until convergence). Solving is done automatically, when necessary, as part of the sageGet...Val functions (sageGetRealVal, sageGetRealPart, etc.) Calling this procedure will update the solution prior to calling any sageGet...Val functions.

The DLL determines the error tolerance for solution convergence automatically, subject to a scale factor defined by the current value of ErrScale (*see* functions Set/GetErrScale).

As of Sage version 5.5 (12-07) the solve process gives up when further effort toward convergence seems hopeless, as determined by the current values of MaxTerribIter and MaxTotalIter (*see* functions Set/GetMaxTerribIter and Set/GetMaxTotalIter). In this case, or for any other exception, this function returns Status = 1 and logs the appropriate error message for retrieval by the sageGetLastErrorMsg function, giving the calling program the option to deal with the problem.

**OptimizeModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| ALogFileName: PAnsiChar | |
| AIdString: PAnsiChar | |

Optimize the model in AAccessForm. ALog filename is a fully-qualified file name of the file that will contain the optimization log information in ASCII text format. If a file by that name already exists then it will be overwritten. AIdstring will appear in the log-file header and also in the caption of the optimization status dialog.

The optimization process returns control to the calling program after convergence. Whether or not convergence is achieved depends to some extend on the solve processes that occur as part of optimization, which are subject to the current values of ErrScale, MaxTerribIter and MaxTotalIter (*see* above). The

optimization process terminates when the number of iterations exceeds the constant MaxOptIter (*see* functions Set/GetMaxOptIter). In this event a message dialog pops up and control returns to the calling program after the user presses the stop button (however, see companion function below).

**sageOptimizeModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| ALogFileName: PAnsiChar | |
| AIdString: PAnsiChar | |
| var Status: Integer | |

Like the OptimizeModel function except silent exception handling. If the optimization terminates due to exceeding the maximum number of iterations or for any other reason, this function returns Status = 1 and logs the appropriate error message for retrieval by the sageGetLastErrorMsg function, giving the calling program the option to deal with the problem.

**IsSolved**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Boolean |

Returns true if solution converged successfully.

**IsOptimized**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Boolean |

Returns true if optimization converged successfully.

## 2.8 Functions for Managing Model Components and Variables

**sageGetRoot**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| var Status: Integer | |

Returns pointer to root model of AAccessForm

**sageCreateChildModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AParentModel: NativeInt | |
| AsidChild: Integer | |
| ADisplayPointX: Integer | |
| ADisplayPointY: Integer | |
| var Status: Integer | |

Creates a child model of AParentModel and returns a pointer to it. AParentModel is a pointer that may be the result of a previous call to CreateChildModel or obtained from functions GetRoot or GetChildModel. AsidChild is the registered stream identifier for the child model class to be created (*see* Stream Identifiers). ADisplayPointX and Y are the screen coordinates at which the model component will be displayed if later opened in the usual Sage GUI.

**sageDeleteModel**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| ADoomed: NativeInt | |
| var Status: Integer | |

Deletes model component ADoomed if possible. ADoomed must be a user-created model component (not built in) and not externally connected to other model components, although it may have internal connections among child model components.

**sageGetModelByName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AMdlName: PAnsiChar | |
| var Status: Integer | |

Returns a pointer to the model component named AMdlName in AAccessForm.

**sageGetVarByName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AMdl: NativeInt | |
| AVarName: PAnsiChar | |
| var Status: Integer | |

Returns a pointer to the variable named AVarName in model component AMdl. in AAccessForm

**sageGetChildModelCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AParentModel: NativeInt | |
| var Status: Integer | |

Returns the number of child model components within AParentModel. Only counts first-generation children.

**sageGetChildModelAt**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AParentModel: NativeInt | |
| AIndex: Integer | |
| var Status: Integer | |

Returns a pointer to the child model components at AIndex of AParentModel's child-model collection. Applies only to first-generation child models. Indexing is zero-based. Valid indices are AIndex = 0..Count-1, where Count is the value returned by sageGetChildModelCount. Returns zero if AIndex is out of range.

**sageGetSeedPodCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AParentModel: NativeInt | |
| var Status: Integer | |

Returns the number of SeedPods (tab pages) in the child-model creation palette of AParentModel.

**sageGetSeedCountAt**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AParentModel: NativeInt | |
| APodIndex: Integer | |
| var Status: Integer | |

Returns the number of seeds in the SeedPod at APodIndex of AParentModel's child creation palette. Indexing is zero-based. Valid indices are APodIndex = 0..Count-1, where Count is the value returned by sageGetSeedPodCount.

**sageGetSeedSidAt**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AParentModel: NativeInt | |
| APodIndex: Integer | |
| ASeedIndex: Integer | |
| var Status: Integer | |

Returns the stream identifier of the seed at ASeedIndex of SeedPod APodIndex of AParentModel's child creation palette. Indexing is zero-based. Valid indices are APodIndex = 0..PodCount-1, where PodCount is the value returned by sageGetSeedPodCount and ASeedIndex = 0..SeedCount-1, where SeedCount is the value returned by sageGetSeedCount. A child model component of this type can be then created using the sageCreateChildModel function.

**sageSetModelName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | none |
| AModel: NativeInt | |
| AName: PAnsiChar | |
| var Status: Integer | |

Sets the name of Amodel to AName

**sageGetModelName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |
| AModel: NativeInt | |
| var Status: Integer | |

Returns the name of AModel.

**sageGetModelSid**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AModel: NativeInt | |
| var Status: Integer | |

Returns the stream identifier integer (sid) of AModel. Each model component type has a unique stream identifier defined in the model-class sid file (*see* Stream Identifiers).

**sageGetVarCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AParentModel: NativeInt | |
| var Status: Integer | |

Returns the number variables within AParentModel.

**sageGetVarAt**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AParentModel: NativeInt | |
| AIndex: Integer | |
| var Status: Integer | |

Returns a pointer to variable at AIndex of AParentModel's variable collection. Valid indices are AIndex = 0..Count-1, where Count is the value returned by sageGetVarCount. Returns zero if AIndex is out of range.

**sageGetVarName**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | PAnsiChar |
| AVar: NativeInt | |
| var Status: Integer | |

Returns the name identifier of AVar.

**sageGetVarSid**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AVar: NativeInt | |
| var Status: Integer | |

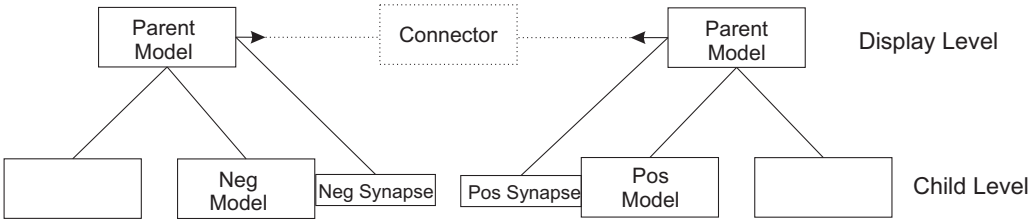
Returns the stream identifier integer (sid) of AVar. Each type of variable TRealVar, TParseVar (user var), TIntegerVar, TFSeriesVar, etc., has a unique stream identifier defined in the file sage.sid

## 2.9 Functions for Managing Model Connections

In order to understand the functions of this section you have to understand Sage terminology for the connections between model components. If you are familiar with the Sage GUI you understand that model components appear as icons in the edit window and connections between them appear as numbered arrows

pointing to the right (positive direction) or left (negative direction). Matching numbers denote connections.

These numbered arrows correspond to what the Sage DLL calls *synapses*, because they are the link through which information flows between model components. The actual *connectors* are the invisible objects that perform the mathematical duties required to connect two synapses together. At least in Sage parlance. Think of a connector as residing between two model components, a negative model component and a positive model component. The connection synapse may belong to those negative or positive models or to child model components, in which case the synapses are displayed at the parent model level in the GUI, as illustrated below. Even though it is possible to physically re-arrange connected model components in any relative orientation you wish in the GUI, it is necessary to think of them arranged in this standard orientation for the function terminology to make sense.




**sageGetNegSynapse**

*Arguments:*
AAccessForm: NativeInt
AModel: NativeInt
AsidCnct: Integer
AOccur: Integer
var Status: Integer

*Returns:*
NativeInt

Returns a handle to a Neg facing synapse (TSynapse instance) for AModel, whose ValidCnctType stream identifier is Asid_Cnct. Each connector type has a unique stream identifier defined in file Cnctobj.sid. In case of multiple such instances (e.g. TFlowReverser), AOccur = 1 selects the first, AOccur = 2 the second, and so forth. Neg facing means has a negative (left) pointing arrow in the visual display.

**sageGetPosSynapse**

*Arguments:*
AAccessForm: NativeInt
AModel: NativeInt
AsidCnct: Integer
AOccur: Integer
var Status: Integer

*Returns:*
NativeInt

Similar to sageGetNegSynapse except for pos facing synapse. Pos facing means has a positive (right) pointing arrow in the visual display.

**sageSynapsesCompatible**

*Arguments:*
AAccessForm: NativeInt
ASynapse1: NativeInt
ASynapse2: NativeInt
var Status: Integer

*Returns:*
Boolean

Returns true if ASynapse1 and ASynapse2 are compatible with each other for purposes of a pending connection or disconnection. Otherwise returns false. Returns Status = 1 if the synapses are not compatible with the reason why posted as the LastErrorMsg.

**sageConnectSynapses**

*Arguments:*
AAccessForm: NativeInt
AParentModel: NativeInt
ASynapse1: NativeInt
ASynapse2: NativeInt
var Status: Integer

*Returns:*
none

Connects together two oppositely oriented synapses ASynapse1 and ASynapse2 under common parent AParentModel. In the Sage visual interface AParentModel is the component within which the connector arrows will appear. Returns Status = 0 if successful. Returns Status = 1 if the connection cannot not be made with the reason why posted as the LastErrorMsg. An invisible TConnector instance is created in the process.

**sageDisconnectSynapses**

*Arguments:*
AAccessForm: NativeInt
ASynapse1: NativeInt
ASynapse2: NativeInt
var Status: Integer

*Returns:*
none

Disconnects synapses ASynapse1 and ASynapse2. Returns Status = 0 if successful. Otherwise returns Status = 1 with the reason why posted as the LastErrorMsg. An invisible TConnector instance is destroyed in the process.

**sageGetSynapseConnectedTo**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AModel: NativeInt | |
| AConnector: NativeInt | |
| var Status: Integer | |

Returns a pointer to the synapse of AModel connected to connector AConnector. Returns zero if no such synapse exists. One of a series of functions useful for tracing connections within existing model structures. For example, you might use sageGetConnectorAt to get AConnector, then sageGetConnectedNegMdl or sageGetConnectedPosMdl to get AModel, then this function to return the synapse involved in the connection.

**sageGetConnectorCount**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AParentModel: NativeInt | |
| var Status: Integer | |

Returns the number of active connections existing within AParentModel. In the GUI, these are the invisible components "between" the numbered arrows in AParentModel's edit window.

**sageGetConnectorAt**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AParentModel: NativeInt | |
| AIndex: Integer | |
| var Status: Integer | |

Returns a pointer to the connector at AIndex of AParentModel's child-model collection. Applies only to first-generation connectors. Indexing is zero-based. Valid indices are AIndex = 0..Count-1, where Count is the value returned by sageGetConnectorCount. Returns zero if AIndex is out of range.

**sageGetConnectorSid**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | Integer |
| AConnector: NativeInt | |
| var Status: Integer | |

Returns the stream identifier integer (sid) of AConnector

**sageGetConnectedNegMdl**

| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AConnector: NativeInt | |
| var Status: Integer | |

Returns a pointer to the model connected on the "negative" side of Aconnector. Of the pair of connected model components, this is the one with the positive directed (right pointing) arrow in the GUI.

**sageGetConnectedPosMdl**

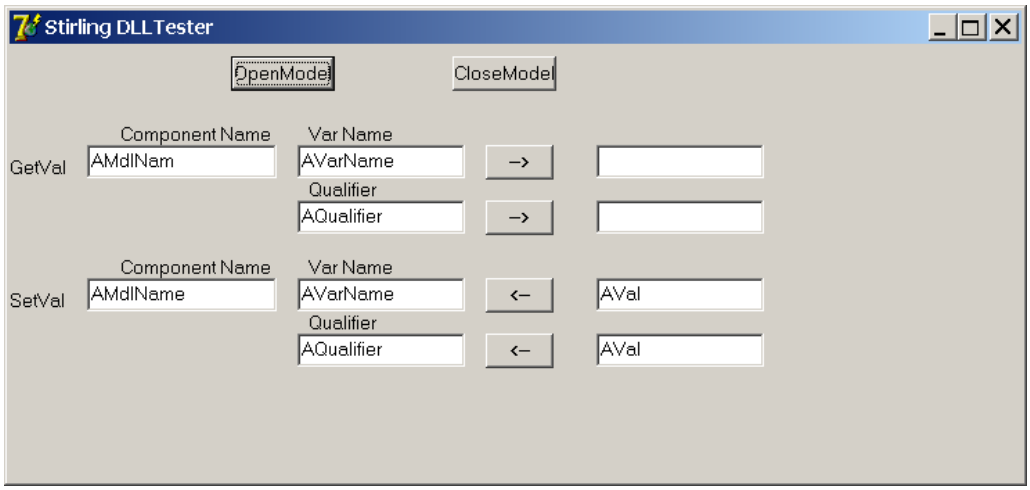
| *Arguments:* | *Returns:* |
|---|---|
| AAccessForm: NativeInt | NativeInt |
| AConnector: NativeInt | |
| var Status: Integer | |

Returns a pointer to the model connected on the "positive" side of Aconnector. Of the pair of connected model components, this is the one with the negative directed (left pointing) arrow in the GUI.

# 3 Testing the DLL

Simple applications TestStirlingLib.exe, TestPtubeLib.exe and TestLTCooler-Lib.exe are included in the DLL distribution to demonstrate that it is indeed possible to use Sage DLLs. The source code for these applications reside in the files TestStirlingLib.dpr, TestStlForm.pas and so forth. All of the test applications have a simple Windows interface that derives from the original TestStir-

lingLib that looked like this:




The two buttons at the top invoke the OpenModel and CloseModel functions. After clicking the OpenModel button a file-open dialog pops up which allows the user to select the stirling model file. After selecting the input file its name is passed to the DLL OpenModel function which then creates a Sage access form with a visual presence as illustrated above.

The arrow buttons allow you to "get" or "set" real-valued variable values (ones that are built-into Sage, not user-defined variables). First you must specify the model-component name, variable name and qualifier (if any) by typing into the boxes so labeled. In the case of setting a value (clicking on left-pointing arrows) you must also type in the value to be assigned. Such as 1.0E-2, 0.01, etc. When you are getting the value (clicking on right-pointing arrows) the value will appear in the box to the right of the arrow, after a the Sage solution is updated. When Sage is in solving mode, the usual Sage "status" dialog pops up to give you feedback on what Sage is doing. If there is a convergence problem during the solution some sort of dialog box will pop up to let you know about it, just as it would under the normal Sage user interface. It is not necessary to manually click the OK button when the solver converges to resume running the test application.

The latest TestStirlingLib.exe has many more testing options than are shown in the above illustration. It has evolved more buttons for testing new functions as they have been added to the DLL. The result of clicking these new buttons is not always obvious and program execution is best monitored with the debugger within the Delphi programming environment so you can follow program execution. Those without access to the Delphi debugger might want to inspect the source code (TestStlForm.pas) manually to see additional comments and examples of how the DLL functions might be called to do various things.

# Index