

Sage SCFusion DLL Application Notes

D. Gedeon
Gedeon Associates

February 10, 2025

Contents

1	DLL Overview	1
1.1	Installation	3
1.2	Exception Handling — Pop-Up Messages vs Silent	3
1.3	AccessForm Window	3
1.4	Solve and Optimize Status Dialogs	4
1.5	Calling Convention	4
1.6	Variable Types	4
1.7	Stream Identifiers	5
1.8	Interface Files	5
1.8.1	Data Types	6
1.8.2	Delphi Example	8
1.8.3	C# Example	8
1.8.4	C Example	8
1.9	Application Examples	8
2	Function Reference	10
2.1	Sage Version	10
2.2	Functions for Pop-Up Dialog Control	11
2.3	Functions for Sage Option Settings	13
2.4	Functions for Model-Class I/O	18
2.5	Functions for Visual Settings	21
2.6	Functions for Getting and Setting Variable Values	22
2.7	Functions for Model Processing	30
2.8	Functions for Managing Model Components and Variables	32
2.9	Functions for Managing Model Connections	36

1 DLL Overview

The purpose of the Sage DLL (dynamic link library) is to provide a way for another program running under the Windows operating system to access Sage functionality to open existing Sage model files, modify inputs, read outputs, etc.

Originally (1995) Sage was strictly a graphical Windows application (GUI). A frequent request among users was for a way to run Sage automatically from another computer program rather than by a human operator. That led to the creation of the first Sage DLL, released under version 3 (2004). Until recently there were three separate DLLs for each of the three model classes (Stirling, Pulse-Tube and Low-T Cooler).

Fast forward 20 years to Sage version 13 and the three model classes have merged into a single SCFusion model class (Stirling Cycle Fusion) encompassing the functionality of all three previous model classes. 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):

Win32 DLL	Win64 DLL	Opens Model Class	File Extension
SCFusionLib.dll	SCFusionLib64.dll	SCFusion	.scfn

The SCFusion DLLs cannot directly open model files previously saved under the Stirling, Pulse-Tube and Low-T Cooler model classes (file extensions .stl, .ptb, .ltc). To work with those files you must first open the file with the normal SCFusion Windows application then save it under the new .scfn file extension.

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. 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.

New in Version 13 The identifiers used to store model-component and variable class instances in file streams changed from integer constants (maintained in separate stream-identifier *.sid files), to the actual class names used in Sage's Delphi source code. As a practical matter this means that some arguments and returned values changed from type integer to `PAnsiChar` (pointer to null-terminated string of ANSI characters) in the following functions.

`sageCreateChildModel`

`sageGetSeedSidAt`

`sageGetVarSid`

`sageGetModelSid`

sageGetNegSynapse

sageGetPosSynapse

sageGetConnectorSid

To support this change, new function SaveModelClassInfo allows you to create an exhaustive list of model-component class names for reference.

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 Explorer. Files are installed in the DLL subdirectory under the Sage installation directory (e.g. c:\Program Files\Gedeon\Sage13\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

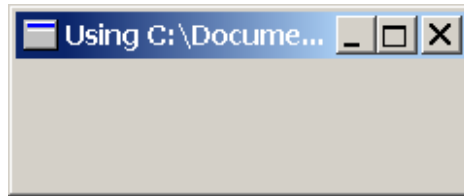
Sage DLLs always handle software exceptions that occur within the DLL. When an exception occurs, some DLL functions pop-up a message dialog window that notifies you of the problem. This temporarily halts program execution until you manually click the OK button. Other DLL functions — those beginning with 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?
<code>register</code>	Left-to-right	Routine	Yes
<code>pascal</code>	Left-to-right	Routine	No
<code>cdecl</code>	Right-to-left	Caller	No
<code>stdcall</code>	Right-to-left	Routine	No
<code>safecall</code>	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

As of version 13 Sage uses the actual class names defined in its Delphi source code to identify instances of that class that are 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 class-name identifiers you may need are those for model components, connectors and variables. The class names for model components are accessible with the `sageGetModelSid` or `sageGetSeedSidAt` functions. Function `SaveModelClassInfo` also allows you to create an exhaustive list of model-component class names for reference. Model component class names also appear in the hints that appear when hovering over the tool buttons in the child-model creation palette or the icons in the edit window of the Sage GUI. Class names for connectors appear in the hints that appear when hovering over the inter-component connection arrows of the Sage GUI. Class names for variables are accessible with the `sageGetVarSid` function.

1.8 Interface Files

In order for your program to link to the Sage DLL you 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. Three sample interface files are provided with the DLL distribution.

sample SCFusionLib interface files	
SCFusionInterface.pas	Embarcadero Delphi (Pascal) format
SCFusionLibInterface.cs	Microsoft Visual Studio (C#) format
SCFusionLib.h	C format

The first interface in Embarcadero Delphi format has been thoroughly tested. If you are using the DLL within the Delphi environment you can directly reference SCFusionInterface.pas in your program's uses statement and be done with it. The second interface in Microsoft Visual-Studio format has been tested for a limited set of function calls. The third file based on generic C data types may require some customization, depending on the application referencing it.

1.8.1 Data Types

Different programming languages have evolved with different names for the same data types. In the DLL interface files, type names are used for function returned values and arguments passed to those functions, not necessarily the same names.

Function return types

Delphi Pascal	Visual-Studio C#	C, C++, ...
(procedure)	void	void
Double	double	double
Extended	double	long double
PAnsiChar	IntPtr	char*
NativeInt	IntPtr	void*
THandle	IntPtr	void*
Integer	int	int
Boolean	bool	bool

In the C# language IntPtr is the name for a generic pointer to an unknown value — an integer with the same size as a pointer. The C and C++ languages denote such pointers by void*. In the Sage DLL generic pointers can be Windows handles or reference data-class instances of variables or model-components.

Arguments passed by value (read-only)

Delphi Pascal	Visual-Studio C#	C, C++, ...
Double	double	double
Extended	double	long double
PAnsiChar	string	char*
NativeInt	IntPtr	void*
THandle	IntPtr	void*
Integer	int	int
Boolean	bool	bool

Arguments passed by reference (read-write) In Delphi Pascal such arguments are designated by the `var` keyword. In the other languages they are designated by different names. It only makes sense to pass non-pointer values by reference.

Delphi Pascal	Visual-Studio C#	C, C++, ...
var ... Double	ref double	double*
var ... Extended	ref double	long double*
var ... Integer	ref int	int*
var ... Boolean	ref bool	bool*

Floating point formats Calling DLL functions involving Double or Extended types requires some care. The floating point format for the calling application may differ from the format in the Sage DLL. In some cases the Win32 SCFusionLib dll will work and the Win64 SCFusionLig64 will not, and vice-versa, according to the following table.

	Win 32	Win 64
Delphi Double	64 bit	64 bit
C-style double	64 bit	64 bit
Delphi Extended	80 bit	64 bit
C-style long double*	80 bit	80 bit

* may also be 64 or 128 bit, depending on compiler

Evidently there may be some issues calling functions involving the Extended type. In that case you may have to avoid calling those functions.

Your choice of whether to use the Win32 or Win64 DLL may also be determined by which of those DLL formats the application you are using supports. Function calling conventions differ for Win32 and Win64 applications.

As far as the internal calculations within the DLL go, the Win32 DLL is actually more precise since it uses 80 bit rather than 64 bit floating-point precision. Higher precision matters when evaluating finite-difference derivatives and can make the difference between the Sage solver converging or not converging. So, if possible use the Win32 DLL.

1.8.2 Delphi Example

In 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 SCFusionInterface;

interface

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

implementation

procedure sageSetRealVal; external 'SCFusionLib.dll';

end.
```

1.8.3 C# Example

The same minimalist interface in C# format would consist of a single declaration block in an interface file that might look like this:

```
[DllImport(DLLname)]
public static extern void sageSetRealVal( IntPtr AAccessForm,
    IntPtr AV,
    double AValue,
    ref int Status);
```

1.8.4 C Example

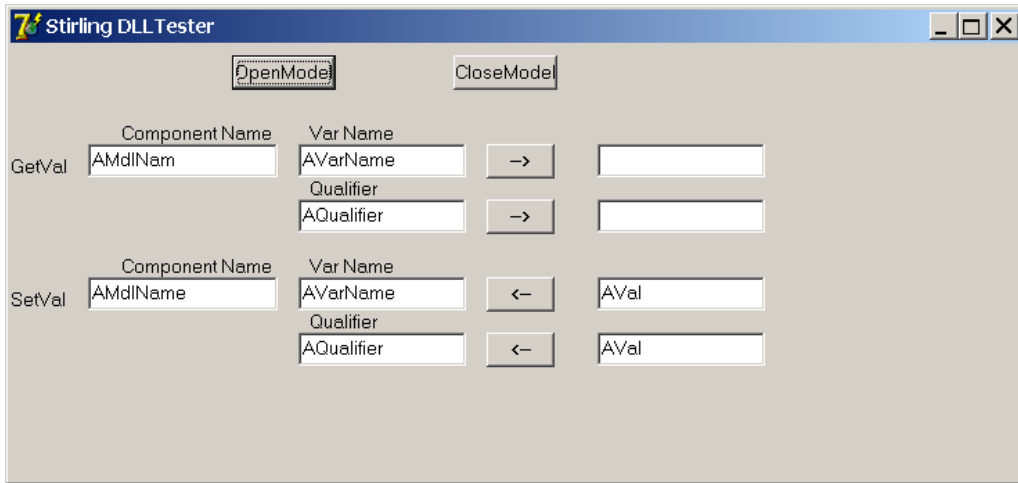
Similarly in C format it would look like this:

```
__declspec(dllexport)
void __stdcall sageSetRealVal(void* AAccessForm,
    void* AV,
    double AValue,
    int* Status);
```

1.9 Application Examples

Sample applications `TestSCFusionLib.exe` and `SageDLLDemo.exe` demonstrate how to use the Sage DLL (*see* Testing the DLL). These application were compiled from projects under the Embarcadero Delphi programming environment

(Pascal language) and Microsoft Visual Studio environment (C# language). Embarcadero Delphi is also the native environment for the SCFusion DLL itself, as well as the SCFusion GUI (graphical user interface). The source code for these applications reside in the files TestSCFusionForm.pas and SageDLLdemoForm.cs. Both the test applications have a simple Windows interface. The interface for TestSCFusionLib derives from the original TestStirlingLib interface 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 you to select the SCFusion 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 TestSCFusionLib.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 (TestSCFusionForm.pas) manually to see additional comments and examples of how the DLL functions might be called to do various things.

2 Function Reference

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 occurred 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:
none

Returns:
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:
none

Returns:
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
none	Boolean

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

SetIgnoreRangeError

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
none	Boolean

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

SetIgnoreDemagError

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
none	Boolean

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

SetIgnoreOverfilledError

Arguments:
AIgnoreOverfilledError: Boolean

Returns:
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:
none

Returns:
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:
none

Returns:
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

<i>Arguments:</i>	<i>Returns:</i>
<code>AErrScale</code> : 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

<i>Arguments:</i>	<i>Returns:</i>
none	Integer

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

SetMaxTerribIter

<i>Arguments:</i>	<i>Returns:</i>
<code>AMaxTerribIter</code> : 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

<i>Arguments:</i>	<i>Returns:</i>
none	Integer

Returns the “`MaxTerribIter`” value. See `SetMaxTerribIter`.

SetMaxTotalIter

<i>Arguments:</i>	<i>Returns:</i>
<code>AMaxTotalIter</code> : Integer	none

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

GetMaxTotalIter

Arguments:
none

Returns:
Integer

Returns the “MaxTotalIter” value. See SetMaxTotalIter.

SetOptMaxIter

Arguments:
AOptMaxIter: Integer

Returns:
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:
none

Returns:
Integer

Returns the “OptMaxIter” value. See SetOptMaxIter

SetOptMaxStepSolveAttempts

Arguments:
AOptMaxStepSolveAttempts: Integer

Returns:
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:
none

Returns:
Integer

Returns the “OptMaxStepSolveAttempts” value. See SetOptMaxStepSolveAttempts

SetDFScale

Arguments:
ADFScale: Integer

Returns:
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:
none

Returns:
Integer

Returns the relative step change limit index. See SetDFScale.

SetDisplayEnumDetail

Arguments:
ADisplayEnumDetail: Integer

Returns:
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 the root model, containing a large number of numeric property values (the details).

GetDisplayEnumDetail

Arguments:
none

Returns:
Integer

Returns the enumerated variable display detail level. See SetDisplayEnumDetail.

SetDisplaySigFigs

Arguments:
ADisplaySigFigs: Integer

Returns:
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

<i>Arguments:</i>	<i>Returns:</i>
none	Integer

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

SetGridInterpOrder

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
none	Integer

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

SetGasFileName

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
none	PAnsiChar

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

SetSolidFileName

Arguments:
ASolidFileName: PAnsiChar

Returns:
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:
none

Returns:
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 SCFusion interface or create a model entirely using DLL functions.

OpenModel

Arguments:
AHandle: THandle
ASageFileName: PAnsiChar

Returns:
NativeInt

Opens a Sage model file (.scfn file extension) 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.scfn). This name may be hard-wired into the application or provided as the returned result of a file-open dialog. The sample program TestSCFusionLib.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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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. Creates or overwrites two files, the main model file (e.g. *.scfn) and an initialization file (e.g. *.scin) holding information about current dimensions in effect.

sageSaveModel

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
ASageFileName: PAnsiChar	
var Status: Integer	

A version of SaveModel with silent exception handling. Created so that an exception encountered while writing either the main model file or the initialization file will not halt execution of the calling program.

SaveListing

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
ALogFileName: PAnsiChar	

Saves user-defined inputs or variables tagged with the "log" attribute to the file ALogFileName, which must be a fully-qualified file name including directory path. Variables tagged with the log attribute appear in mapping or optimization log files. They are so tagged by checking the "write to log file" checkbox during their definition in the standard Sage GUI. The resulting file is an ASCII format tab-delimited text file that may be opened with any text editor or a spreadsheet like Excel.

SaveCADVars

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
ALogFileName: PAnsiChar	

Similar to SaveTaggedVars except for user-defined inputs or variables tagged with the "CAD" attribute. They are tagged with this attribute by checking the "CAD variable" checkbox during their definition in the standard Sage GUI. The CAD attribute provides a way to select inputs or outputs that serve as driving dimensions in solid models during a subsequent CAD design process. The resulting file is an ASCII format tab-delimited text file that may be opened with any text editor or a spreadsheet like Excel.

SaveModelClassInfo

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
ATxtFileName: PAnsiChar	

New in version 13. Saves an exhaustive list of class-name identifier strings and default display name of all available model-component classes to the file ATxtFileName, which must be a fully-qualified file name including directory path. This list intended for use with other functions (e.g. sageCreateChild-Model) that require class name identifier strings passed as arguments. The resulting file is an ASCII format tab-delimited text file that may be opened with any text editor or a spreadsheet like Excel.

SaveSolutionGrid

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none

Makes the AccessForm window invisible.

ShowAccessForm

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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:

AAccessForm: NativeInt

Returns:

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:

AAccessForm: NativeInt

AMdlName: PAnsiChar

AVarName: PAnsiChar

AValue: Double

Returns:

none

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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
AV: NativeInt	
AValue: Double	
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>Double</code>
<code>AV: NativeInt</code>	
<code>var Status: Integer</code>	

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

sageSetIntegerVal

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>none</code>
<code>AV: NativeInt</code>	
<code>AValue: Integer</code>	
<code>var Status: Integer</code>	

Sets the value of the integer variable `AV` to `AValue`.

sageGetIntegerVal

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>Integer</code>
<code>AV: NativeInt</code>	
<code>var Status: Integer</code>	

Returns the value of the Integer variable `AV`.

SetRealPart

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>none</code>
<code>AMdlName: PAnsiChar</code>	
<code>AVarName: PAnsiChar</code>	
<code>AQualifier: PAnsiChar</code>	
<code>AValue: Double</code>	

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 θ_n of the n -th harmonic. Rather than directly setting `amp.n = r_n` and `arg.n = θ_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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	Double
AMdlName: PAnsiChar	
AVarName: PAnsiChar	
AQualifier: PAnsiChar	

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

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>Double</code>
<code>AV: NativeInt</code>	
<code>AQualifier: PAnsiChar</code>	
<code>var Status: Integer</code>	

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

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>none</code>
<code>APairsVar: NativeInt</code>	
<code>ACount: Integer</code>	
<code>var Status: Integer</code>	

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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
AFSeriesVar: NativeInt	
ACount: Integer	
var Status: Integer	

Re-initializes the number of terms within Fourier-series input variable AFSeriesVar. 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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AMdlName: PAnsiChar	
AVarName: PAnsiChar	

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

sageGetGasName

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AMdlName: PAnsiChar	
AVarName: PAnsiChar	

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

sageGetSolidName

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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 extent 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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	Boolean

Returns true if solution converged successfully.

IsOptimized

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	Boolean

Returns true if optimization converged successfully.

2.8 Functions for Managing Model Components and Variables

sageGetRoot

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	NativeInt
var Status: Integer	

Returns pointer to root model of AAccessForm

sageCreateChildModel

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	NativeInt
AParentModel: NativeInt	
AsidChild: PAnsiChar	
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 class name identifier string 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.

Note `AsidChild` type changed from `Integer` to `PAnsiChar` in Sage version 13.

sageDeleteModel

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>none</code>
<code>ADoomed: NativeInt</code>	
<code>var Status: Integer</code>	

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

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>NativeInt</code>
<code>AMdlName: PAnsiChar</code>	
<code>var Status: Integer</code>	

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

sageGetVarByName

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>NativeInt</code>
<code>AMdl: NativeInt</code>	
<code>AVarName: PAnsiChar</code>	
<code>var Status: Integer</code>	

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

sageGetChildModelCount

<i>Arguments:</i>	<i>Returns:</i>
<code>AAccessForm: NativeInt</code>	<code>Integer</code>
<code>AParentModel: NativeInt</code>	
<code>var Status: Integer</code>	

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

sageGetChildModelAt

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	Integer
AParentModel: NativeInt	
var Status: Integer	

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

sageGetSeedCountAt

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AParentModel: NativeInt	
APodIndex: Integer	
ASeedIndex: Integer	
var Status: Integer	

Returns the class-name 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.

Note Return type changed from Integer to PAnsiChar in Sage version 13.

sageSetModelName

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
AModel: NativeInt	
AName: PAnsiChar	
var Status: Integer	

Sets the name of Amodel to AName

sageGetModelName

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AModel: NativeInt	
var Status: Integer	

Returns the name of AModel.

sageGetModelSid

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AModel: NativeInt	
var Status: Integer	

Returns the class-name stream identifier of AModel. Each model component type has a unique class name. (*see* Stream Identifiers).

Note Return type changed from Integer to PAnsiChar in Sage version 13.

sageGetVarCount

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	Integer
AParentModel: NativeInt	
var Status: Integer	

Returns the number variables within AParentModel.

sageGetVarAt

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AVar: NativeInt	
var Status: Integer	

Returns the name identifier of AVar.

sageGetVarSid

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AVar: NativeInt	
var Status: Integer	

Returns the class-name stream identifier of AVar. Each type of variable TRealVar, TParseVar (user var), TIntegerVar, TFSeriesVar, etc., has a unique class name. (*see* Stream Identifiers).

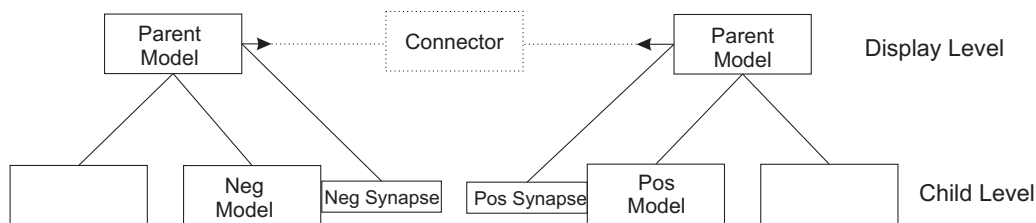
Note Return type changed from Integer to PAnsiChar in Sage version 13.

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: PAnsiChar
 AOccur: Integer
 var Status: Integer

Returns:

NativeInt

Returns a handle to a Neg facing synapse (TSynapse instance) for AModel, whose ValidCnctType class-name stream identifier is AsidCnct. (*see* Stream Identifiers). 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.

Note AsidCnct type changed from Integer to PAnsiChar in Sage version 13.

sageGetPosSynapse

Arguments:

AAccessForm: NativeInt
 AModel: NativeInt
 AsidCnct: PAnsiChar
 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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	Boolean
ASynapse1: NativeInt	
ASynapse2: NativeInt	
var Status: Integer	

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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
AParentModel: NativeInt	
ASynapse1: NativeInt	
ASynapse2: NativeInt	
var Status: Integer	

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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	none
ASynapse1: NativeInt	
ASynapse2: NativeInt	
var Status: Integer	

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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
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

<i>Arguments:</i>	<i>Returns:</i>
AAccessForm: NativeInt	PAnsiChar
AConnector: NativeInt	
var Status: Integer	

Returns the class-name stream identifier of AConnector.

Note Return type changed from Integer to PAnsiChar in Sage version 13.

sageGetConnectedNegMdl

<i>Arguments:</i>	<i>Returns:</i>
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:

AAccessForm: NativeInt

AConnector: NativeInt

var Status: Integer

Returns:

NativeInt

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.

Index

CloseModel, 17
GetCaptionAccessForm, 19
GetDFScale, 13
GetDisplayEnumDetail, 14
GetDisplaySigFigs, 14
GetErrScale, 11
GetExtendedPart, 24
GetExtendedVal, 21
GetGasFileName, 15
GetGasName, 26
GetGridInterpOrder, 15
GetIgnoreDemagError, 10
GetIgnoreExprError, 8
GetIgnoreOverfilledError, 10
GetIgnoreRangeError, 9
GetMaxTerriblter, 12
GetMaxTotalIter, 12
GetOptMaxIter, 12
GetOptMaxStepSolveAttempts, 13
GetRealPart, 24
GetRealVal, 20
GetSolidFileName, 15
GetSolidName, 27
GetUserVarVal, 22
GetVersion, 8
HideAccessForm, 18
IsOptimized, 29
IsSolved, 29
OpenModel, 16
OptimizeModel, 28
ReinitializeModel, 28
SaveCADVars, 18
SaveListing, 17
SaveModel, 17
SaveSolutionGrid, 18
SaveTaggedVars, 18
SetCaptionAccessForm, 19
SetDFScale, 13
SetDefaultDims, 11
SetDisplayEnumDetail, 13
SetDisplaySigFigs, 14
SetErrScale, 11
SetExtendedPart, 24
SetExtendedVal, 21
SetGasFileName, 15
SetGasName, 26
SetGridInterpOrder, 14
SetIgnoreDemagError, 9
SetIgnoreExprError, 8
SetIgnoreOverfilledError, 10
SetIgnoreRangeError, 9
SetMaxTerriblter, 11
SetMaxTotalIter, 12
SetOptMaxIter, 12
SetOptMaxStepSolveAttempts, 12
SetRealPart, 23
SetRealVal, 20
SetSolidFileName, 15
SetSolidName, 27
SetUserVar, 21
ShowAccessForm, 19
sageConnectSynapses, 35
sageCreateChildModel, 30
sageDeleteModel, 30
sageDisconnectSynapses, 35
sageGetChildModelAt, 31
sageGetChildModelCount, 31
sageGetConnectedNegMdl, 37
sageGetConnectedPosMdl, 37
sageGetConnectorAt, 36
sageGetConnectorCount, 36
sageGetConnectorSid, 37
sageGetExtendedPart, 24
sageGetExtendedVal, 21
sageGetFSeriesCount, 25
sageGetGasName, 26
sageGetIntegerVal, 22
sageGetLastErrorMsg, 7
sageGetModelByName, 30
sageGetModelName, 32
sageGetModelSid, 32
sageGetNegSynapse, 34
sageGetPairsCount, 25
sageGetPosSynapse, 35
sageGetRealPart, 24
sageGetRealVal, 21

sageGetRoot, 29
sageGetSeedCountAt, 31
sageGetSeedPodCount, 31
sageGetSeedSidAt, 32
sageGetSolidName, 27
sageGetSynapseConnectedTo, 36
sageGetUserVarVal, 22
sageGetVarAt, 33
sageGetVarByName, 30
sageGetVarCount, 33
sageGetVarName, 33
sageGetVarSid, 33
sageNewModel, 16
sageOpenModel, 16
sageOptimizeModel, 29
sageSaveModel, 17
sageSetExtendedPart, 24
sageSetExtendedVal, 21
sageSetFSeriesCount, 25
sageSetGasName, 26
sageSetIntegerVal, 22
sageSetModelName, 32
sageSetPairsCount, 25
sageSetRealPart, 23
sageSetRealVal, 20
sageSetSolidName, 27
sageSetUserVar, 22
sageSolveModel, 28
sageSynapsesCompatible, 35