
SAF User Manual

Release 2.0.3

LLNL

Mar 12, 2021

Contents

1	SAF User Manuals	3
1.1	Sets and Fields (SAF) API Reference Manual for SAF 2.0.3	3
1.2	Sets and Fields (SAF) Examples and Use Cases	185
1.3	SAF Support Library (SSlib) Programming Reference Manual for SAF 2.0.0 and later	215
	Index	323

SAF Software is available on [github](#)

1.1 Sets and Fields (SAF) API Reference Manual for SAF 2.0.3

Acknowledgements

1.1.1 Table of Contents

Permuted Index

Table 1.1: Permuted Index

Concept	Key	Reference
Turn off	aborts	<i>saf_setProps_DontAbort</i>
Specify read-only database	access	<i>saf_setProps_ReadOnly</i>
Finalize	access to the library	<i>saf_final</i>
	Add members to a collection	<i>saf_extend_collection</i>
Declare a new	algebraic type	<i>saf_declare_algebraic</i>
Describe an	algebraic type	<i>saf_describe_algebraic</i>
Find one	algebraic type	<i>saf_find_one_algebraic</i>
Common	algebraic types	<i>SAF_ALGTYPE</i>
Find	algebraic types	<i>saf_find_algebraics</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
More meaningful	alias for SAF_TOTALITY	<i>SAF_WHOLE_FIELD</i>
NULL	aliases	<i>SAF_VoidPtr</i>
Begin a block of error handling code for	all errors	<i>SAF_CATCH_ALL</i>
String	allocation modes	<i>SAF_StrMode</i>
Set string	allocation style	<i>saf_setProps_StrMode</i>
Set the pool size for string return value	allocations	<i>saf_setProps_StrPoolSize</i>
Find	alternate index specs by matching criteria	<i>saf_find_alternate_indexspecs</i>
Read an	alternate index specs from disk	<i>saf_read_alternate_indexspecs</i>
Write an	alternate index specs to disk	<i>saf_write_alternate_indexspecs</i>
Get a description of an	alternate indexing spec	<i>saf_describe_alternate_indexspec</i>
Declare an	Alternative Index Specification	<i>saf_declare_alternate_indexspec</i>
The quantity	Amount	<i>SAF_QAMOUNT</i>
Version	Annotation	<i>SAF_VERSION_ANNOT</i>
Check if	any stat errors occurred	<i>saf_getInfo_staterror</i>
Trace SAF	API calls and times	<i>SAF_TRACING</i>
Not	applicable	<i>SAF_NOT_APPLICABLE_IN</i>
Find the not	applicable unit	<i>saf_find_unit_not_applicable</i>
	Apply a logarithmic scale to a unit	<i>saf_log_unit</i>
An	arbitrary named quantity	<i>SAF_QNAME</i>
Make a C-automatic	array of cat handles	<i>SAF_Cat</i>
Make a C-automatic	array of set handles	<i>SAF_Set</i>
	Array size	<i>SAF_NELMTS</i>
Control	Assertion Checking	<i>SAF_ASSERT_DISABLE</i>
	Associates a unit of measure with a specific quantity	<i>saf_quantify_unit</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Associating a role with a collection category	<i>SAF_RoleConstants</i>
	Attach an attribute to a state group	<i>saf_put_state_grp_att</i>
	Attach an attribute to a state template	<i>saf_put_state_tmpl_att</i>
	Attach an attribute to a suite	<i>saf_put_suite_att</i>
Get an attribute	attached to a state group	<i>saf_get_state_grp_att</i>
Get an attribute	attached to a state template	<i>saf_get_state_tmpl_att</i>
Get an attribute	attached to a suite	<i>saf_get_suite_att</i>
Create or update a non-sharable	attribute	<i>saf_put_attribute</i>
Read a non-sharable	attribute	<i>saf_get_attribute</i>
Get an	attribute attached to a state group	<i>saf_get_state_grp_att</i>
Get an	attribute attached to a state template	<i>saf_get_state_tmpl_att</i>
Get an	attribute attached to a suite	<i>saf_get_suite_att</i>
Get an	attribute from a field	<i>saf_get_field_att</i>
Get an	attribute from a set	<i>saf_get_set_att</i>
Reserved	attribute name keys	<i>SAF_ATT</i>
Put an	attribute to a set	<i>saf_put_set_att</i>
Attach an	attribute to a state group	<i>saf_put_state_grp_att</i>
Attach an	attribute to a state template	<i>saf_put_state_tmpl_att</i>
Attach an	attribute to a suite	<i>saf_put_suite_att</i>
Get an	attribute with a cat	<i>saf_get_cat_att</i>
Put an	attribute with a cat	<i>saf_put_cat_att</i>
Put an	attribute with a field	<i>saf_put_field_att</i>
Get an	attribute with a field template	<i>saf_get_field_tmpl_att</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Put an	attribute with a field template	<i>saf_put_field_tmpl_att</i>
Make a field handle a C	automatic variable	SAF_Field
Make a field template handle a C	automatic variable	SAF_FieldTmpl
Make a relation handle a C	automatic variable	SAF_Rel
Make a state handle a C	automatic variable	SAF_StateGrp
Make a state template handle a C	automatic variable	SAF_StateTmpl
Make a suite handle a C	automatic variable	SAF_Suite
Database information not	available	<i>SAF_NOT_SET_DB</i>
Synchronization	barrier	<i>SAF_BARRIER</i>
Find	bases	<i>saf_find_bases</i>
Declare a new	basis type	<i>saf_declare_basis</i>
Describe a	basis type	<i>saf_describe_basis</i>
Find one	basis type	<i>saf_find_one_basis</i>
	Basis types	<i>SAF_BasisConstants</i>
Queries whether data has	been written	<i>saf_data_has_been_written_t</i>
	Begin a block of error handling code	<i>SAF_CATCH_ERR</i>
	Begin a block of error handling code for all errors	<i>SAF_CATCH_ALL</i>
	Begin a the CATCH part of a TRY/CATCH block	<i>SAF_CATCH</i>
	Begin a TRY/CATCH block	<i>SAF_TRY_BEGIN</i>
Begin a the CATCH part of a TRY/CATCH	block	<i>SAF_CATCH</i>
Begin a TRY/CATCH	block	<i>SAF_TRY_BEGIN</i>
End a TRY/CATCH	block	<i>SAF_TRY_END</i>
Begin a	block of error handling code	<i>SAF_CATCH_ERR</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Begin a	block of error handling code for all errors	<i>SAF_CATCH_ALL</i>
	Boundary set tri-state	<i>SAF_BoundMode</i>
Conveniently specify a	boundary subset	<i>SAF_BOUNDARY</i>
Conveniently specify an embedded	boundary subset	<i>SAF_EMBEDBND</i>
Initialize the	built-in object registry	<i>_saf_gen_stdtypes</i>
	Built-in registry name	<i>SAF_REGISTRY_SAVE</i>
Make a field handle a	C automatic variable	<i>SAF_Field</i>
Make a field template handle a	C automatic variable	<i>SAF_FieldTmpl</i>
Make a relation handle a	C automatic variable	<i>SAF_Rel</i>
Make a state handle a	C automatic variable	<i>SAF_StateGrp</i>
Make a state template handle a	C automatic variable	<i>SAF_StateTmpl</i>
Make a suite handle a	C automatic variable	<i>SAF_Suite</i>
Make a	C-automatic array of cat handles	<i>SAF_Cat</i>
Make a	C-automatic array of set handles	<i>SAF_Set</i>
Specify a	callback for error conditions	<i>saf_setProps_ErrFunc</i>
The rank of the	calling process	<i>SAF_RANK</i>
Trace SAF API	calls and times	<i>SAF_TRACING</i>
Declare a field as a	candidate coordinate field	<i>saf_declare_coords</i>
Get an attribute with a	cat	<i>saf_get_cat_att</i>
Put an attribute with a	cat	<i>saf_put_cat_att</i>
Make a C-automatic array of	cat handles	<i>SAF_Cat</i>
Begin a the	CATCH part of a TRY/CATCH block	<i>SAF_CATCH</i>
Find collection	categories	<i>saf_find_categories</i>
Associating a role with a collection	category	<i>SAF_RoleConstants</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Declare a collection	category	<i>saf_declare_category</i>
Get a description of a collection	category	<i>saf_describe_category</i>
Query quantity	characteristics	<i>saf_describe_quantity</i>
Query unit	characteristics	<i>saf_describe_unit</i>
	Check if any stat errors occurred	<i>saf_getInfo_staterror</i>
	Check if path is a SAF database	<i>saf_getInfo_isSAFdatabase</i>
	Check if path is an HDF5 file	<i>saf_getInfo_isHDFfile</i>
Control Assertion	Checking	<i>SAF_ASSERT_DISABLE</i>
Control Postcondition	Checking	<i>SAF_POSTCOND_DISABLE</i>
Control Precondition	Checking	<i>SAF_PRECOND_DISABLE</i>
	Clobber an existing database on open	<i>saf_setProps_Clobber</i>
	Close a database	<i>saf_close_database</i>
Begin a block of error handling	code	<i>SAF_CATCH_ERR</i>
Begin a block of error handling	code for all errors	<i>SAF_CATCH_ALL</i>
Return	codes	<i>SAF_return_t</i>
Error	codes returned by the library	<i>SAF_error_t</i>
Add members to a	collection	<i>saf_extend_collection</i>
Declare a	collection	<i>saf_declare_collection</i>
Describe a	collection	<i>saf_describe_collection</i>
Find	collection categories	<i>saf_find_categories</i>
Associating a role with a	collection category	<i>SAF_RoleConstants</i>
Declare a	collection category	<i>saf_declare_category</i>
Get a description of a	collection category	<i>saf_describe_category</i>
Declare a new	collection role	<i>saf_declare_role</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Find one	collection role	<i>saf_find_one_role</i>
Compare two	collections	<i>saf_same_collections</i>
Find	collections	<i>saf_find_collections</i>
	Common algebraic types	<i>SAF_ALGTYPE</i>
Specify MPI database	communicator	<i>saf_setProps_DbComm</i>
The size of the	communicator	<i>SAF_SIZE</i>
Set the MPI	communicator for the library	<i>saf_setProps_LibComm</i>
	Compare two collections	<i>saf_same_collections</i>
Find the parent of a	component field	<i>_saf_find_parent_field</i>
Field	component interleave modes	<i>SAF_Interleave</i>
Specify a callback for error	conditions	<i>saf_setProps_ErrFunc</i>
Conveniently specify a	constant field	<i>SAF_CONSTANT</i>
Update database	contents	<i>saf_update_database</i>
	Control Assertion Checking	<i>SAF_ASSERT_DISABLE</i>
	Control Postcondition Checking	<i>SAF_POSTCOND_DISABLE</i>
	Control Precondition Checking	<i>SAF_PRECOND_DISABLE</i>
	Control Reporting of Error Messages	<i>SAF_ERROR_REPORTING</i>
	Convenience function for finding a quantity	<i>saf_find_one_quantity</i>
	Convenience function for finding a unit	<i>saf_find_one_unit</i>
	Conveniently specify a boundary subset	<i>SAF_BOUNDARY</i>
	Conveniently specify a constant field	<i>SAF_CONSTANT</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Conveniently specify a decomposition-centered field	<i>SAF_DECOMP</i>
	Conveniently specify a node-centered field	<i>SAF_NODAL</i>
	Conveniently specify a typical subset	<i>SAF_COMMON</i>
	Conveniently specify a zone-centered field	<i>SAF_ZONAL</i>
	Conveniently specify an embedded boundary subset	<i>SAF_EMBEDBND</i>
	Conveniently specify an general subset	<i>SAF_GENERAL</i>
	Convert a single value	<i>_saf_convert</i>
Declare a field as a candidate	coordinate field	<i>saf_declare_coords</i>
Find	coordinate fields	<i>saf_find_coords</i>
Find default	coordinate fields	<i>saf_find_default_coords</i>
Declare default	coordinates of a given set	<i>saf_declare_default_coords</i>
	Copy a string	<i>_saf_strdup</i>
	Create a new database property list with default values	<i>saf_createProps_database</i>
	Create a new library property list with default values	<i>saf_createProps_lib</i>
	Create an memory-resident database	<i>saf_setProps_MemoryResident</i>
	Create or update a non-sharable attribute	<i>saf_put_attribute</i>
Find alternate index specs by matching	criteria	<i>saf_find_alternate_indexspecs</i>
Find set by matching	criteria	<i>saf_find_matching_sets</i>
The quantity	Current	<i>SAF_QCURRENT</i>
Does field have	data	<i>saf_data_has_been_written_t</i>
Read topological relation	data	<i>saf_read_topo_relation</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Write topological relation	data	<i>saf_write_topo_relation</i>
Read the	data for a field	<i>saf_read_field</i>
Write the	data for a field	<i>saf_write_field</i>
Read the	data for a subset relation	<i>saf_read_subset_relation</i>
Queries whether	data has been written	<i>saf_data_has_been_written_t</i>
Reuse	data in a subset relation	<i>saf_use_written_subset_relati</i>
Check if path is a SAF	database	<i>saf_getInfo_isSAFdatabase</i>
Close a	database	<i>saf_close_database</i>
Create an memory-resident	database	<i>saf_setProps_MemoryResiden</i>
Open a	database	<i>saf_open_database</i>
Specify location of SAF's standard types	database	<i>SAF_REGISTRIES</i>
Specify read-only	database access	<i>saf_setProps_ReadOnly</i>
Specify MPI	database communicator	<i>saf_setProps_DbComm</i>
Update	database contents	<i>saf_update_database</i>
	Database information not available	<i>SAF_NOT_SET_DB</i>
Clobber an existing	database on open	<i>saf_setProps_Clobber</i>
Free	database property list	<i>saf_freeProps_database</i>
Create a new	database property list with default values	<i>saf_createProps_database</i>
Get	datatype and size for a field	<i>saf_get_count_and_type_for_</i>
Get	datatype and size for a subset relation	<i>saf_get_count_and_type_for_</i>
Get	datatype and size for a topological relation	<i>saf_get_count_and_type_for_</i>
Determine if	datatype is primitive	<i>_saf_is_primitive_type</i>
Predefined scalar	datatypes	<i>SAF_type_t</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Declare a collection	<i>saf_declare_collection</i>
	Declare a collection category	<i>saf_declare_category</i>
	Declare a field	<i>saf_declare_field</i>
	Declare a field as a candidate coordinate field	<i>saf_declare_coords</i>
	Declare a field template	<i>saf_declare_field_tmpl</i>
	Declare a new algebraic type	<i>saf_declare_algebraic</i>
	Declare a new basis type	<i>saf_declare_basis</i>
	Declare a new collection role	<i>saf_declare_role</i>
	Declare a new evaluation type	<i>saf_declare_evaluation</i>
	Declare a new object	<i>saf_declare_relrep</i>
	Declare a new quantity	<i>saf_declare_quantity</i>
	Declare a new unit	<i>saf_declare_unit</i>
	Declare a set	<i>saf_declare_set</i>
	Declare a state group	<i>saf_declare_state_group</i>
	Declare a state template	<i>saf_declare_state_tmpl</i>
	Declare a subset relation	<i>saf_declare_subset_relation</i>
	Declare a suite	<i>saf_declare_suite</i>
	Declare a topological relation	<i>saf_declare_topo_relation</i>
	Declare an Alternative Index Specification	<i>saf_declare_alternate_indexspec</i>
	Declare default coordinates of a given set	<i>saf_declare_default_coords</i>
The self	decomposition of a set	<i>SAF_SELF</i>
	Decomposition tri-state	<i>SAF_DecompMode</i>
Conveniently specify a	decomposition-centered field	<i>SAF_DECOMP</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Find	default coordinate fields	<i>saf_find_default_coords</i>
Declare	default coordinates of a given set	<i>saf_declare_default_coords</i>
	Default properties	<i>SAF_DEFAULT_DBPROPS</i>
Create a new database property list with	default values	<i>saf_createProps_database</i>
Create a new library property list with	default values	<i>saf_createProps_lib</i>
Divide a quantity into a quantity	definition	<i>saf_divide_quantity</i>
Divide a unit into a unit	definition	<i>saf_divide_unit</i>
Multiply a quantity into a quantity	definition	<i>saf_multiply_quantity</i>
Multiply a unit into a unit	definition	<i>saf_multiply_unit</i>
	Describe a basis type	<i>saf_describe_basis</i>
	Describe a collection	<i>saf_describe_collection</i>
	Describe a role	<i>saf_describe_role</i>
	Describe an algebraic type	<i>saf_describe_algebraic</i>
	Describe an evaluation type	<i>saf_describe_evaluation</i>
	Describe an object	<i>saf_describe_relrep</i>
Obtain a set	description	<i>saf_describe_set</i>
Get a	description of a collection category	<i>saf_describe_category</i>
Get a	description of a field	<i>saf_describe_field</i>
Get a	description of a field template	<i>saf_describe_field_tmpl</i>
Get a	description of a state group	<i>saf_describe_state_group</i>
Get a	description of a state template	<i>saf_describe_state_tmpl</i>
Get a	description of a subset relation	<i>saf_describe_subset_relation</i>
Get a	description of a suite	<i>saf_describe_suite</i>
Get a	description of an alternate indexing spec	<i>saf_describe_alternate_index</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Get	description of topological relation	<i>saf_describe_topo_relation</i>
Set the	destination form of a field	<i>saf_target_field</i>
Set the	destination form of a subset relation	<i>saf_target_subset_relation</i>
Set the	destination form of a topological relation	<i>saf_target_topo_relation</i>
	Determine if a handle is a valid handle	<i>SAF_VALID</i>
	Determine if datatype is primitive	<i>_saf_is_primitive_type</i>
	Determine if two handles refer to the same object	<i>SAF_EQUIV</i>
Topological	dimensions	<i>SAF_TopoDim</i>
Read an alternate index specs from	disk	<i>saf_read_alternate_indexspec</i>
Write an alternate index specs to	disk	<i>saf_write_alternate_indexspec</i>
	Divide a quantity into a quantity definition	<i>saf_divide_quantity</i>
	Divide a unit into a unit definition	<i>saf_divide_unit</i>
	Does field have data	<i>saf_data_has_been_written_t</i>
Conveniently specify an	embedded boundary subset	<i>SAF_EMBEDBND</i>
	End a TRY/CATCH block	<i>SAF_TRY_END</i>
	Error codes returned by the library	<i>SAF_error_t</i>
Specify a callback for	error conditions	<i>saf_setProps_ErrFunc</i>
Begin a block of	error handling code	<i>SAF_CATCH_ERR</i>
Begin a block of	error handling code for all errors	<i>SAF_CATCH_ALL</i>
Set the library	error handling mode	<i>saf_setProps_ErrorMode</i>
Set the	error logging mode	<i>saf_setProps_ErrorLogging</i>
Get stat	error message	<i>saf_getInfo_errmsg</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Control Reporting of	Error Messages	<i>SAF_ERROR_REPORTING</i>
	Error return modes	<i>SAF_ErrMode</i>
Return a pointer to an	error string	<i>saf_error_str</i>
Begin a block of error handling code for all	errors	<i>SAF_CATCH_ALL</i>
Check if any stat	errors ocured	<i>saf_getInfo_staterror</i>
Declare a new	evaluation type	<i>saf_declare_evaluation</i>
Describe an	evaluation type	<i>saf_describe_evaluation</i>
Find one	evaluation type	<i>saf_find_one_evaluation</i>
	Evaluation Types	<i>SAF_EvalConstants</i>
Find	evaluation types	<i>saf_find_evaluations</i>
	Exchange handles	<i>saf_allgather_handles</i>
	Exclusive OR operator	<i>SAF_XOR</i>
Clobber an	existing database on open	<i>saf_setProps_Clobber</i>
	Extendable set tri-state	<i>SAF_ExtendMode</i>
Conveniently specify a constant	field	<i>SAF_CONSTANT</i>
Conveniently specify a decomposition-centered	field	<i>SAF_DECOMP</i>
Conveniently specify a node-centered	field	<i>SAF_NODAL</i>
Conveniently specify a zone-centered	field	<i>SAF_ZONAL</i>
Declare a	field	<i>saf_declare_field</i>
Declare a field as a candidate coordinate	field	<i>saf_declare_coords</i>
Find the parent of a component	field	<i>_saf_find_parent_field</i>
Get a description of a	field	<i>saf_describe_field</i>
Get an attribute from a	field	<i>saf_get_field_att</i>
Get datatype and size for a	field	<i>saf_get_count_and_type_for_</i>
Put an attribute with a	field	<i>saf_put_field_att</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Read the data for a	field	<i>saf_read_field</i>
Set the destination form of a	field	<i>saf_target_field</i>
Write the data for a	field	<i>saf_write_field</i>
Declare a	field as a candidate coordinate field	<i>saf_declare_coords</i>
	Field component interleave modes	<i>SAF_Interleave</i>
The null	field handle	<i>SAF_NULL_FIELD</i>
Make a	field handle a C automatic variable	<i>SAF_Field</i>
Does	field have data	<i>saf_data_has_been_written_t</i>
Is	field stored on self	<i>saf_is_self_stored_field</i>
Declare a	field template	<i>saf_declare_field_tmpl</i>
Get a description of a	field template	<i>saf_describe_field_tmpl</i>
Get an attribute with a	field template	<i>saf_get_field_tmpl_att</i>
Put an attribute with a	field template	<i>saf_put_field_tmpl_att</i>
The null	field template handle	<i>SAF_NULL_FTMPL</i>
Make a	field template handle a C automatic variable	<i>SAF_FieldTmpl</i>
Find	field templates	<i>saf_find_field_tmpls</i>
Find	fields	<i>saf_find_fields</i>
Find coordinate	fields	<i>saf_find_coords</i>
Find default coordinate	fields	<i>saf_find_default_coords</i>
Check if path is an HDF5	file	<i>saf_getInfo_isHDFfile</i>
Specify registry	file	<i>saf_setProps_Registry</i>
	Finalize access to the library	<i>saf_final</i>
	Find a state template	<i>saf_find_state_tmpl</i>
	Find algebraic types	<i>saf_find_algebraics</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Find alternate index specs by matching criteria	<i>saf_find_alternate_indexspecs</i>
	Find bases	<i>saf_find_bases</i>
	Find collection categories	<i>saf_find_categories</i>
	Find collections	<i>saf_find_collections</i>
	Find coordinate fields	<i>saf_find_coords</i>
	Find default coordinate fields	<i>saf_find_default_coords</i>
	Find evaluation types	<i>saf_find_evaluations</i>
	Find field templates	<i>saf_find_field_tmpls</i>
	Find fields	<i>saf_find_fields</i>
Set	find modes	<i>SAF_FindSetMode</i>
	Find one algebraic type	<i>saf_find_one_algebraic</i>
	Find one basis type	<i>saf_find_one_basis</i>
	Find one collection role	<i>saf_find_one_role</i>
	Find one evaluation type	<i>saf_find_one_evaluation</i>
	Find one object	<i>saf_find_one_relrep</i>
	Find quantities	<i>saf_find_quantities</i>
	Find relation representation types	<i>saf_find_relreps</i>
	Find roles	<i>saf_find_roles</i>
	Find set by matching criteria	<i>saf_find_matching_sets</i>
	Find sets by traversing the subset inclusion lattice	<i>saf_find_sets</i>
	Find state groups	<i>saf_find_state_groups</i>
	Find subset relations	<i>saf_find_subset_relations</i>
	Find suites	<i>saf_find_suites</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Find the not applicable unit	<i>saf_find_unit_not_applicable</i>
	Find the parent of a component field	<i>_saf_find_parent_field</i>
	Find topological relations	<i>saf_find_topo_relations</i>
	Find units	<i>saf_find_units</i>
Convenience function for	finding a quantity	<i>saf_find_one_quantity</i>
Convenience function for	finding a unit	<i>saf_find_one_unit</i>
Set the destination	form of a field	<i>saf_target_field</i>
Set the destination	form of a subset relation	<i>saf_target_subset_relation</i>
Set the destination	form of a topological relation	<i>saf_target_topo_relation</i>
	Free database property list	<i>saf_freeProps_database</i>
	Free library property list	<i>saf_freeProps_lib</i>
	Free SAF_PathInfo	<i>saf_freeInfo_path</i>
Convenience	function for finding a quantity	<i>saf_find_one_quantity</i>
Convenience	function for finding a unit	<i>saf_find_one_unit</i>
Conveniently specify an	general subset	<i>SAF_GENERAL</i>
	Get a description of a collection category	<i>saf_describe_category</i>
	Get a description of a field	<i>saf_describe_field</i>
	Get a description of a field template	<i>saf_describe_field_tmpl</i>
	Get a description of a state group	<i>saf_describe_state_group</i>
	Get a description of a state template	<i>saf_describe_state_tmpl</i>
	Get a description of a subset relation	<i>saf_describe_subset_relation</i>
	Get a description of a suite	<i>saf_describe_suite</i>
	Get a description of an alternate indexing spec	<i>saf_describe_alternate_index</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Get an attribute attached to a state group	<i>saf_get_state_grp_att</i>
	Get an attribute attached to a state template	<i>saf_get_state_tmpl_att</i>
	Get an attribute attached to a suite	<i>saf_get_suite_att</i>
	Get an attribute from a field	<i>saf_get_field_att</i>
	Get an attribute from a set	<i>saf_get_set_att</i>
	Get an attribute with a cat	<i>saf_get_cat_att</i>
	Get an attribute with a field template	<i>saf_get_field_tmpl_att</i>
	Get datatype and size for a field	<i>saf_get_count_and_type_for_</i>
	Get datatype and size for a subset relation	<i>saf_get_count_and_type_for_</i>
	Get datatype and size for a topological relation	<i>saf_get_count_and_type_for_</i>
	Get description of topological relation	<i>saf_describe_topo_relation</i>
	Get stat error message	<i>saf_getInfo_errmsg</i>
	Get the HDF5 version	<i>saf_getInfo_hdfversion</i>
	Get the MPI library version	<i>saf_getInfo_mpverson</i>
	Get the SAF library version	<i>saf_getInfo_libversion</i>
Declare default coordinates of a	given set	<i>saf_declare_default_coords</i>
	Grab HDF5 I/O library	<i>saf_grab_hdf5</i>
Attach an attribute to a state	group	<i>saf_put_state_grp_att</i>
Declare a state	group	<i>saf_declare_state_group</i>
Get a description of a state	group	<i>saf_describe_state_group</i>
Get an attribute attached to a state	group	<i>saf_get_state_grp_att</i>
The null state	group handle	<i>SAF_NULL_STATE_GRP</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Find state	groups	<i>saf_find_state_groups</i>
Determine if a handle is a valid	handle	<i>SAF_VALID</i>
The null field	handle	<i>SAF_NULL_FIELD</i>
The null field template	handle	<i>SAF_NULL_FTMPL</i>
The null relation	handle	<i>SAF_NULL_REL</i>
The null set	handle	<i>SAF_NULL_SET</i>
The null state group	handle	<i>SAF_NULL_STATE_GRP</i>
The null state template	handle	<i>SAF_NULL_STMPL</i>
The null suite	handle	<i>SAF_NULL_SUITE</i>
The universe set	handle	<i>SAF_UNIVERSE</i>
Make a field	handle a C automatic variable	SAF_Field
Make a field template	handle a C automatic variable	SAF_FieldTmpl
Make a relation	handle a C automatic variable	SAF_Rel
Make a state	handle a C automatic variable	SAF_StateGrp
Make a state template	handle a C automatic variable	SAF_StateTmpl
Make a suite	handle a C automatic variable	SAF_Suite
Determine if a	handle is a valid handle	<i>SAF_VALID</i>
Exchange	handles	<i>saf_allgather_handles</i>
Make a C-automatic array of cat	handles	SAF_Cat
Make a C-automatic array of set	handles	SAF_Set
Determine if two	handles refer to the same object	<i>SAF_EQUIV</i>
Begin a block of error	handling code	<i>SAF_CATCH_ERR</i>
Begin a block of error	handling code for all errors	<i>SAF_CATCH_ALL</i>
Set the library error	handling mode	<i>saf_setProps_ErrorMode</i>
Queries whether data	has been written	<i>saf_data_has_been_written_t</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Does field	have data	<i>saf_data_has_been_written_t</i>
Check if path is an	HDF5 file	<i>saf_getInfo_isHDFfile</i>
Grab	HDF5 I/O library	<i>saf_grab_hdf5</i>
Ungrab	HDF5 I/O library	<i>saf_ungrab_hdf5</i>
Get the	HDF5 version	<i>saf_getInfo_hdfversion</i>
Grab HDF5	I/O library	<i>saf_grab_hdf5</i>
Ungrab HDF5	I/O library	<i>saf_ungrab_hdf5</i>
Determine	if a handle is a valid handle	<i>SAF_VALID</i>
Check	if any stat errors occurred	<i>saf_getInfo_staterror</i>
Determine	if datatype is primitive	<i>_saf_is_primitive_type</i>
Check	if path is a SAF database	<i>saf_getInfo_isSAFdatabase</i>
Check	if path is an HDF5 file	<i>saf_getInfo_isHDFfile</i>
Determine	if two handles refer to the same object	<i>SAF_EQUIV</i>
Not	implemented	<i>SAF_NOT_IMPL</i>
Find sets by traversing the subset	inclusion lattice	<i>saf_find_sets</i>
Subset	inclusion lattice roles	<i>SAF_SilRole</i>
Declare an Alternative	Index Specification	<i>saf_declare_alternate_indexspec</i>
Find alternate	index specs by matching criteria	<i>saf_find_alternate_indexspec</i>
Read an alternate	index specs from disk	<i>saf_read_alternate_indexspec</i>
Write an alternate	index specs to disk	<i>saf_write_alternate_indexspec</i>
	Indexing scheme	<i>SAF_IndexSchema</i>
	Indexing scheme	<i>SAF_FORDER</i>
	Indexing scheme	<i>SAF_CORDER</i>
	Indexing scheme	<i>SAF_IDC</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Indexing scheme	<i>SAF_NA_INDEXSPEC</i>
	Indexing scheme	<i>SAF_2DC</i>
	Indexing scheme	<i>SAF_3DC</i>
	Indexing scheme	<i>SAF_1DF</i>
	Indexing scheme	<i>SAF_2DF</i>
	Indexing scheme	<i>SAF_3DF</i>
Get a description of an alternate	indexing spec	<i>saf_describe_alternate_index</i>
Load	information from the specified path	<i>saf_readInfo_path</i>
Database	information not available	<i>SAF_NOT_SET_DB</i>
	Initialize the built-in object registry	<i>_saf_gen_stdtypes</i>
	Initialize the library	<i>saf_init</i>
Field component	interleave modes	<i>SAF_Interleave</i>
Divide a quantity	into a quantity definition	<i>saf_divide_quantity</i>
Multiply a quantity	into a quantity definition	<i>saf_multiply_quantity</i>
Divide a unit	into a unit definition	<i>saf_divide_unit</i>
Multiply a unit	into a unit definition	<i>saf_multiply_unit</i>
Check if path	is a SAF database	<i>saf_getInfo_isSAFdatabase</i>
Determine if a handle	is a valid handle	<i>SAF_VALID</i>
Check if path	is an HDF5 file	<i>saf_getInfo_isHDFfile</i>
	Is field stored on self	<i>saf_is_self_stored_field</i>
Determine if datatype	is primitive	<i>_saf_is_primitive_type</i>
	Is topological relation stored on self	<i>saf_is_self_stored_topo_rel</i>
Reserved attribute name	keys	<i>SAF_ATT</i>
Find sets by traversing the subset inclusion	lattice	<i>saf_find_sets</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Subset inclusion	lattice roles	<i>SAF_SilRole</i>
The quantity	Length	<i>SAF_QLENGTH</i>
Error codes returned by the	library	<i>SAF_error_t</i>
Finalize access to the	library	<i>saf_final</i>
Grab HDF5 I/O	library	<i>saf_grab_hdf5</i>
Initialize the	library	<i>saf_init</i>
Set the MPI communicator for the	library	<i>saf_setProps_LibComm</i>
Ungrab HDF5 I/O	library	<i>saf_ungrab_hdf5</i>
Set the	library error handling mode	<i>saf_setProps_ErrorMode</i>
	Library properties	<i>SAF_DEFAULT_LIBPROPS</i>
Free	library property list	<i>saf_freeProps_lib</i>
Create a new	library property list with default values	<i>saf_createProps_lib</i>
Get the MPI	library version	<i>saf_getInfo_mpiversion</i>
Get the SAF	library version	<i>saf_getInfo_libversion</i>
The quantity	Light	<i>SAF_QLIGHT</i>
Free database property	list	<i>saf_freeProps_database</i>
Free library property	list	<i>saf_freeProps_lib</i>
Create a new database property	list with default values	<i>saf_createProps_database</i>
Create a new library property	list with default values	<i>saf_createProps_lib</i>
	Load information from the specified path	<i>saf_readInfo_path</i>
Specify	location of SAF's standard types database	<i>SAF_REGISTRIES</i>
Apply a	logarithmic scale to a unit	<i>saf_log_unit</i>
Set the error	logging mode	<i>saf_setProps_ErrorLogging</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Major version number	<i>SAF_VERSION_MAJOR</i>
	Make a C-automatic array of cat handles	SAF_Cat
	Make a C-automatic array of set handles	SAF_Set
	Make a field handle a C automatic variable	SAF_Field
	Make a field template handle a C automatic variable	SAF_FieldTmpl
	Make a relation handle a C automatic variable	SAF_Rel
	Make a state handle a C automatic variable	SAF_StateGrp
	Make a state template handle a C automatic variable	SAF_StateTmpl
	Make a suite handle a C automatic variable	SAF_Suite
The quantity	Mass	<i>SAF_QMASS</i>
Find alternate index specs by	matching criteria	<i>saf_find_alternate_indexspecs</i>
Find set by	matching criteria	<i>saf_find_matching_sets</i>
More	meaningful alias for SAF_TOTALITY	<i>SAF_WHOLE_FIELD</i>
Associates a unit of	measure with a specific quantity	<i>saf_quantify_unit</i>
Add	members to a collection	<i>saf_extend_collection</i>
Create an	memory-resident database	<i>saf_setProps_MemoryResident</i>
Get stat error	message	<i>saf_getInfo_errmsg</i>
Control Reporting of Error	Messages	<i>SAF_ERROR_REPORTING</i>
	Minor version number	<i>SAF_VERSION_MINOR</i>
Set the error logging	mode	<i>saf_setProps_ErrorLogging</i>
Set the library error handling	mode	<i>saf_setProps_ErrorMode</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Top	mode tri-state	<i>SAF_TopMode</i>
Error return	modes	<i>SAF_ErrMode</i>
Field component interleave	modes	<i>SAF_Interleave</i>
Set find	modes	<i>SAF_FindSetMode</i>
String allocation	modes	<i>SAF_StrMode</i>
	More meaningful alias for SAF_TOTALITY	<i>SAF_WHOLE_FIELD</i>
Set the	MPI communicator for the library	<i>saf_setProps_LibComm</i>
Specify	MPI database communicator	<i>saf_setProps_DbComm</i>
Get the	MPI library version	<i>saf_getInfo_mpiversion</i>
	Multiply a quantity into a quantity definition	<i>saf_multiply_quantity</i>
	Multiply a unit into a unit definition	<i>saf_multiply_unit</i>
Built-in registry	name	<i>SAF_REGISTRY_SAVE</i>
Reserved attribute	name keys	<i>SAF_ATT</i>
An arbitrary	named quantity	<i>SAF_QNAME</i>
Declare a	new algebraic type	<i>saf_declare_algebraic</i>
Declare a	new basis type	<i>saf_declare_basis</i>
Declare a	new collection role	<i>saf_declare_role</i>
Create a	new database property list with default values	<i>saf_createProps_database</i>
Declare a	new evaluation type	<i>saf_declare_evaluation</i>
Create a	new library property list with default values	<i>saf_createProps_lib</i>
Declare a	new object	<i>saf_declare_relrep</i>
Declare a	new quantity	<i>saf_declare_quantity</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Declare a	new unit	<i>saf_declare_unit</i>
Conveniently specify a	node-centered field	<i>SAF_NODAL</i>
Create or update a	non-sharable attribute	<i>saf_put_attribute</i>
Read a	non-sharable attribute	<i>saf_get_attribute</i>
	Not applicable	<i>SAF_NOT_APPLICABLE_IN</i>
Find the	not applicable unit	<i>saf_find_unit_not_applicable</i>
Database information	not available	<i>SAF_NOT_SET_DB</i>
	Not implemented	<i>SAF_NOT_IMPL</i>
	NULL aliases	<i>SAF_VoidPtr</i>
The	null field handle	<i>SAF_NULL_FIELD</i>
The	null field template handle	<i>SAF_NULL_FTAMPL</i>
The	null relation handle	<i>SAF_NULL_REL</i>
The	null set handle	<i>SAF_NULL_SET</i>
The	null state group handle	<i>SAF_NULL_STATE_GRP</i>
The	null state template handle	<i>SAF_NULL_STAMPL</i>
The	null suite handle	<i>SAF_NULL_SUITE</i>
Major version	number	<i>SAF_VERSION_MAJOR</i>
Minor version	number	<i>SAF_VERSION_MINOR</i>
Release	number	<i>SAF_VERSION_RELEASE</i>
Returns string representation of version	number	<i>saf_version_string</i>
Declare a new	object	<i>saf_declare_relrep</i>
Describe an	object	<i>saf_describe_relrep</i>
Determine if two handles refer to the same	object	<i>SAF_EQUIV</i>
Find one	object	<i>saf_find_one_relrep</i>
Initialize the built-in	object registry	<i>_saf_gen_stdtypes</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Obtain a set description	<i>saf_describe_set</i>
	Obtain permissions of path	<i>saf_getInfo_permissions</i>
Check if any stat errors	occured	<i>saf_getInfo_staterror</i>
Turn	off aborts	<i>saf_setProps_DontAbort</i>
Translate unit by an	offset	<i>saf_offset_unit</i>
Clobber an existing database	on open	<i>saf_setProps_Clobber</i>
Is field stored	on self	<i>saf_is_self_stored_field</i>
Is topological relation stored	on self	<i>saf_is_self_stored_topo_relat</i>
Find	one algebraic type	<i>saf_find_one_algebraic</i>
Find	one basis type	<i>saf_find_one_basis</i>
Find	one collection role	<i>saf_find_one_role</i>
Find	one evaluation type	<i>saf_find_one_evaluation</i>
Find	one object	<i>saf_find_one_relrep</i>
Clobber an existing database on	open	<i>saf_setProps_Clobber</i>
	Open a database	<i>saf_open_database</i>
Exclusive OR	operator	<i>SAF_XOR</i>
Write	out a state	<i>saf_write_state</i>
Find the	parent of a component field	<i>_saf_find_parent_field</i>
Begin a the CATCH	part of a TRY/CATCH block	<i>SAF_CATCH</i>
Load information from the specified	path	<i>saf_readInfo_path</i>
Obtain permissions of	path	<i>saf_getInfo_permissions</i>
Check if	path is a SAF database	<i>saf_getInfo_isSAFdatabase</i>
Check if	path is an HDF5 file	<i>saf_getInfo_isHDFfile</i>
Obtain	permissions of path	<i>saf_getInfo_permissions</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Return a	pointer to an error string	<i>saf_error_str</i>
Set the	pool size for string return value allocations	<i>saf_setProps_StrPoolSize</i>
Control	Postcondition Checking	<i>SAF_POSTCOND_DISABLE</i>
Control	Precondition Checking	<i>SAF_PRECOND_DISABLE</i>
	Predefined scalar datatypes	<i>SAF_type_t</i>
Determine if datatype is	primitive	<i>_saf_is_primitive_type</i>
The rank of the calling	process	<i>SAF_RANK</i>
Default	properties	<i>SAF_DEFAULT_DBPROPS</i>
Library	properties	<i>SAF_DEFAULT_LIBPROPS</i>
Free database	property list	<i>saf_freeProps_database</i>
Free library	property list	<i>saf_freeProps_lib</i>
Create a new database	property list with default values	<i>saf_createProps_database</i>
Create a new library	property list with default values	<i>saf_createProps_lib</i>
	Put an attribute to a set	<i>saf_put_set_att</i>
	Put an attribute with a cat	<i>saf_put_cat_att</i>
	Put an attribute with a field	<i>saf_put_field_att</i>
	Put an attribute with a field template	<i>saf_put_field_tmpl_att</i>
Find	quantities	<i>saf_find_quantities</i>
An arbitrary named	quantity	<i>SAF_QNAME</i>
Associates a unit of measure with a specific	quantity	<i>saf_quantify_unit</i>
Convenience function for finding a	quantity	<i>saf_find_one_quantity</i>
Declare a new	quantity	<i>saf_declare_quantity</i>
The	quantity Amount	<i>SAF_QAMOUNT</i>
Query	quantity characteristics	<i>saf_describe_quantity</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
The	quantity Current	<i>SAF_QCURRENT</i>
Divide a quantity into a	quantity definition	<i>saf_divide_quantity</i>
Multiply a quantity into a	quantity definition	<i>saf_multiply_quantity</i>
Divide a	quantity into a quantity definition	<i>saf_divide_quantity</i>
Multiply a	quantity into a quantity definition	<i>saf_multiply_quantity</i>
The	quantity Length	<i>SAF_QLENGTH</i>
The	quantity Light	<i>SAF_QLIGHT</i>
The	quantity Mass	<i>SAF_QMASS</i>
The	quantity Temperature	<i>SAF_QTEMP</i>
The	quantity Time	<i>SAF_QTIME</i>
	Queries whether data has been written	<i>saf_data_has_been_written_t</i>
	Query quantity characteristics	<i>saf_describe_quantity</i>
	Query unit characteristics	<i>saf_describe_unit</i>
The	rank of the calling process	<i>SAF_RANK</i>
	Read a non-sharable attribute	<i>saf_get_attribute</i>
	Read an alternate index specs from disk	<i>saf_read_alternate_indexspec</i>
	Read the data for a field	<i>saf_read_field</i>
	Read the data for a subset relation	<i>saf_read_subset_relation</i>
	Read topological relation data	<i>saf_read_topo_relation</i>
Specify	read-only database access	<i>saf_setProps_ReadOnly</i>
Determine if two handles	refer to the same object	<i>SAF_EQUIV</i>
Initialize the built-in object	registry	<i>_saf_gen_stdtypes</i>
Specify	registry file	<i>saf_setProps_Registry</i>
Built-in	registry name	<i>SAF_REGISTRY_SAVE</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Declare a subset	relation	<i>saf_declare_subset_relation</i>
Declare a topological	relation	<i>saf_declare_topo_relation</i>
Get a description of a subset	relation	<i>saf_describe_subset_relation</i>
Get datatype and size for a subset	relation	<i>saf_get_count_and_type_for_</i>
Get datatype and size for a topological	relation	<i>saf_get_count_and_type_for_</i>
Get description of topological	relation	<i>saf_describe_topo_relation</i>
Read the data for a subset	relation	<i>saf_read_subset_relation</i>
Reuse data in a subset	relation	<i>saf_use_written_subset_relati</i>
Set the destination form of a subset	relation	<i>saf_target_subset_relation</i>
Set the destination form of a topological	relation	<i>saf_target_topo_relation</i>
Write a subset	relation	<i>saf_write_subset_relation</i>
Read topological	relation data	<i>saf_read_topo_relation</i>
Write topological	relation data	<i>saf_write_topo_relation</i>
The null	relation handle	<i>SAF_NULL_REL</i>
Make a	relation handle a C automatic variable	SAF_Rel
	Relation representation types	<i>SAF_TopoRelRep</i>
Find	relation representation types	<i>saf_find_relreps</i>
Subset	relation representation types	<i>SAF_SubsetRelRep</i>
Is topological	relation stored on self	<i>saf_is_self_stored_topo_relat</i>
Find subset	relations	<i>saf_find_subset_relations</i>
Find topological	relations	<i>saf_find_topo_relations</i>
	Release number	<i>SAF_VERSION_RELEASE</i>
Control	Reporting of Error Messages	<i>SAF_ERROR_REPORTING</i>
Returns string	representation of version number	<i>saf_version_string</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Find relation	representation types	<i>saf_find_relreps</i>
Relation	representation types	<i>SAF_TopoRelRep</i>
Subset relation	representation types	<i>SAF_SubsetRelRep</i>
	Reserved attribute name keys	<i>SAF_ATT</i>
	Retrieve a state	<i>saf_read_state</i>
	Return a pointer to an error string	<i>saf_error_str</i>
	Return codes	<i>SAF_return_t</i>
Error	return modes	<i>SAF_ErrMode</i>
Set the pool size for string	return value allocations	<i>saf_setProps_StrPoolSize</i>
Error codes	returned by the library	<i>SAF_error_t</i>
	Returns string representation of version number	<i>saf_version_string</i>
	Reuse data in a subset relation	<i>saf_use_written_subset_relattr</i>
Declare a new collection	role	<i>saf_declare_role</i>
Describe a	role	<i>saf_describe_role</i>
Find one collection	role	<i>saf_find_one_role</i>
Associating a	role with a collection category	<i>SAF_RoleConstants</i>
Find	roles	<i>saf_find_roles</i>
Subset inclusion lattice	roles	<i>SAF_SilRole</i>
Trace	SAF API calls and times	<i>SAF_TRACING</i>
Check if path is a	SAF database	<i>saf_getInfo_isSAFdatabase</i>
Get the	SAF library version	<i>saf_getInfo_libversion</i>
Specify location of	SAF's standard types database	<i>SAF_REGISTRIES</i>
Free	SAF_PathInfo	<i>saf_freeInfo_path</i>
More meaningful alias for	SAF_TOTALITY	<i>SAF_WHOLE_FIELD</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Determine if two handles refer to the	same object	<i>SAF_EQUIV</i>
Predefined	scalar datatypes	<i>SAF_type_t</i>
Apply a logarithmic	scale to a unit	<i>saf_log_unit</i>
Indexing	scheme	<i>SAF_IndexSchema</i>
Indexing	scheme	<i>SAF_FORDER</i>
Indexing	scheme	<i>SAF_CORDER</i>
Indexing	scheme	<i>SAF_IDC</i>
Indexing	scheme	<i>SAF_NA_INDEXSPEC</i>
Indexing	scheme	<i>SAF_2DC</i>
Indexing	scheme	<i>SAF_3DC</i>
Indexing	scheme	<i>SAF_1DF</i>
Indexing	scheme	<i>SAF_2DF</i>
Indexing	scheme	<i>SAF_3DF</i>
Wildcards for	searching	SAF
Is field stored on	self	<i>saf_is_self_stored_field</i>
Is topological relation stored on	self	<i>saf_is_self_stored_topo_rel</i>
The	self decomposition of a set	<i>SAF_SELF</i>
	Serial/Parallel-dependent variable	<i>SAF_PARALLEL_VAR</i>
Declare a	set	<i>saf_declare_set</i>
Declare default coordinates of a given	set	<i>saf_declare_default_coords</i>
Get an attribute from a	set	<i>saf_get_set_att</i>
Put an attribute to a	set	<i>saf_put_set_att</i>
The self decomposition of a	set	<i>SAF_SELF</i>
Find	set by matching criteria	<i>saf_find_matching_sets</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Obtain a	set description	<i>saf_describe_set</i>
	Set find modes	<i>SAF_FindSetMode</i>
The null	set handle	<i>SAF_NULL_SET</i>
The universe	set handle	<i>SAF_UNIVERSE</i>
Make a C-automatic array of	set handles	<i>SAF_Set</i>
	Set string allocation style	<i>saf_setProps_StrMode</i>
	Set the destination form of a field	<i>saf_target_field</i>
	Set the destination form of a subset relation	<i>saf_target_subset_relation</i>
	Set the destination form of a topological relation	<i>saf_target_topo_relation</i>
	Set the error logging mode	<i>saf_setProps_ErrorLogging</i>
	Set the library error handling mode	<i>saf_setProps_ErrorMode</i>
	Set the MPI communicator for the library	<i>saf_setProps_LibComm</i>
	Set the pool size for string return value allocations	<i>saf_setProps_StrPoolSize</i>
Boundary	set tri-state	<i>SAF_BoundMode</i>
Extendable	set tri-state	<i>SAF_ExtendMode</i>
Find	sets by traversing the subset inclusion lattice	<i>saf_find_sets</i>
Convert a	single value	<i>_saf_convert</i>
Array	size	<i>SAF_NELMTS</i>
Get datatype and	size for a field	<i>saf_get_count_and_type_for_</i>
Get datatype and	size for a subset relation	<i>saf_get_count_and_type_for_</i>
Get datatype and	size for a topological relation	<i>saf_get_count_and_type_for_</i>
Set the pool	size for string return value allocations	<i>saf_setProps_StrPoolSize</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
The	size of the communicator	<i>SAF_SIZE</i>
Get a description of an alternate indexing	spec	<i>saf_describe_alternate_index</i>
Associates a unit of measure with a	specific quantity	<i>saf_quantify_unit</i>
Declare an Alternative Index	Specification	<i>saf_declare_alternate_index</i>
Load information from the	specified path	<i>saf_readInfo_path</i>
Conveniently	specify a boundary subset	<i>SAF_BOUNDARY</i>
	Specify a callback for error conditions	<i>saf_setProps_ErrFunc</i>
Conveniently	specify a constant field	<i>SAF_CONSTANT</i>
Conveniently	specify a decomposition-centered field	<i>SAF_DECOMP</i>
Conveniently	specify a node-centered field	<i>SAF_NODAL</i>
Conveniently	specify a typical subset	<i>SAF_COMMON</i>
Conveniently	specify a zone-centered field	<i>SAF_ZONAL</i>
Conveniently	specify an embedded boundary subset	<i>SAF_EMBEDBND</i>
Conveniently	specify an general subset	<i>SAF_GENERAL</i>
	Specify location of SAF's standard types database	<i>SAF_REGISTRIES</i>
	Specify MPI database communicator	<i>saf_setProps_DbComm</i>
	Specify read-only database access	<i>saf_setProps_ReadOnly</i>
	Specify registry file	<i>saf_setProps_Registry</i>
Find alternate index	specs by matching criteria	<i>saf_find_alternate_indexspec</i>
Read an alternate index	specs from disk	<i>saf_read_alternate_indexspec</i>
Write an alternate index	specs to disk	<i>saf_write_alternate_indexspec</i>
	Standard tri-state values	<i>SAF_TriState</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Specify location of SAF's	standard types database	<i>SAF_REGISTRIES</i>
Get	stat error message	<i>saf_getInfo_errmsg</i>
Check if any	stat errors occurred	<i>saf_getInfo_staterror</i>
Retrieve a	state	<i>saf_read_state</i>
Write out a	state	<i>saf_write_state</i>
Attach an attribute to a	state group	<i>saf_put_state_grp_att</i>
Declare a	state group	<i>saf_declare_state_group</i>
Get a description of a	state group	<i>saf_describe_state_group</i>
Get an attribute attached to a	state group	<i>saf_get_state_grp_att</i>
The null	state group handle	<i>SAF_NULL_STATE_GRP</i>
Find	state groups	<i>saf_find_state_groups</i>
Make a	state handle a C automatic variable	<i>SAF_StateGrp</i>
Attach an attribute to a	state template	<i>saf_put_state_tmpl_att</i>
Declare a	state template	<i>saf_declare_state_tmpl</i>
Find a	state template	<i>saf_find_state_tmpl</i>
Get a description of a	state template	<i>saf_describe_state_tmpl</i>
Get an attribute attached to a	state template	<i>saf_get_state_tmpl_att</i>
The null	state template handle	<i>SAF_NULL_STMPL</i>
Make a	state template handle a C automatic variable	<i>SAF_StateTmpl</i>
Is field	stored on self	<i>saf_is_self_stored_field</i>
Is topological relation	stored on self	<i>saf_is_self_stored_topo_rel</i>
Copy a	string	<i>_saf_strdup</i>
Return a pointer to an error	string	<i>saf_error_str</i>
	String allocation modes	<i>SAF_StrMode</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Set	string allocation style	<i>saf_setProps_StrMode</i>
Returns	string representation of version number	<i>saf_version_string</i>
Set the pool size for	string return value allocations	<i>saf_setProps_StrPoolSize</i>
Set string allocation	style	<i>saf_setProps_StrMode</i>
Conveniently specify a boundary	subset	<i>SAF_BOUNDARY</i>
Conveniently specify a typical	subset	<i>SAF_COMMON</i>
Conveniently specify an embedded boundary	subset	<i>SAF_EMBEDBND</i>
Conveniently specify an general	subset	<i>SAF_GENERAL</i>
Find sets by traversing the	subset inclusion lattice	<i>saf_find_sets</i>
	Subset inclusion lattice roles	<i>SAF_SilRole</i>
Declare a	subset relation	<i>saf_declare_subset_relation</i>
Get a description of a	subset relation	<i>saf_describe_subset_relation</i>
Get datatype and size for a	subset relation	<i>saf_get_count_and_type_for_</i>
Read the data for a	subset relation	<i>saf_read_subset_relation</i>
Reuse data in a	subset relation	<i>saf_use_written_subset_relati</i>
Set the destination form of a	subset relation	<i>saf_target_subset_relation</i>
Write a	subset relation	<i>saf_write_subset_relation</i>
	Subset relation representation types	<i>SAF_SubsetRelRep</i>
Find	subset relations	<i>saf_find_subset_relations</i>
Attach an attribute to a	suite	<i>saf_put_suite_att</i>
Declare a	suite	<i>saf_declare_suite</i>
Get a description of a	suite	<i>saf_describe_suite</i>
Get an attribute attached to a	suite	<i>saf_get_suite_att</i>
The null	suite handle	<i>SAF_NULL_SUITE</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Make a	suite handle a C automatic variable	SAF_Suite
Find	suites	<i>saf_find_suites</i>
	Synchronization barrier	<i>SAF_BARRIER</i>
The quantity	Temperature	<i>SAF_QTEMP</i>
Attach an attribute to a state	template	<i>saf_put_state_tmpl_att</i>
Declare a field	template	<i>saf_declare_field_tmpl</i>
Declare a state	template	<i>saf_declare_state_tmpl</i>
Find a state	template	<i>saf_find_state_tmpl</i>
Get a description of a field	template	<i>saf_describe_field_tmpl</i>
Get a description of a state	template	<i>saf_describe_state_tmpl</i>
Get an attribute attached to a state	template	<i>saf_get_state_tmpl_att</i>
Get an attribute with a field	template	<i>saf_get_field_tmpl_att</i>
Put an attribute with a field	template	<i>saf_put_field_tmpl_att</i>
The null field	template handle	<i>SAF_NULL_FTMPL</i>
The null state	template handle	<i>SAF_NULL_STMPL</i>
Make a field	template handle a C automatic variable	SAF_FieldTmpl
Make a state	template handle a C automatic variable	SAF_StateTmpl
Find field	templates	<i>saf_find_field_tmpls</i>
The quantity	Time	<i>SAF_QTIME</i>
Trace SAF API calls and	times	<i>SAF_TRACING</i>
	Top mode tri-state	<i>SAF_TopMode</i>
	Topological dimensions	<i>SAF_TopoDim</i>
Declare a	topological relation	<i>saf_declare_topo_relation</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Get datatype and size for a	topological relation	<i>saf_get_count_and_type_for_</i>
Get description of	topological relation	<i>saf_describe_topo_relation</i>
Set the destination form of a	topological relation	<i>saf_target_topo_relation</i>
Read	topological relation data	<i>saf_read_topo_relation</i>
Write	topological relation data	<i>saf_write_topo_relation</i>
Is	topological relation stored on self	<i>saf_is_self_stored_topo_relat</i>
Find	topological relations	<i>saf_find_topo_relations</i>
	Trace SAF API calls and times	<i>SAF_TRACING</i>
	Translate unit by an offset	<i>saf_offset_unit</i>
Find sets by	traversing the subset inclusion lattice	<i>saf_find_sets</i>
Boundary set	tri-state	<i>SAF_BoundMode</i>
Decomposition	tri-state	<i>SAF_DecompMode</i>
Extendable set	tri-state	<i>SAF_ExtendMode</i>
Top mode	tri-state	<i>SAF_TopMode</i>
Standard	tri-state values	<i>SAF_TriState</i>
Begin a	TRY/CATCH block	<i>SAF_TRY_BEGIN</i>
Begin a the CATCH part of a	TRY/CATCH block	<i>SAF_CATCH</i>
End a	TRY/CATCH block	<i>SAF_TRY_END</i>
	Turn off aborts	<i>saf_setProps_DontAbort</i>
Compare	two collections	<i>saf_same_collections</i>
Determine if	two handles refer to the same object	<i>SAF_EQUIV</i>
Declare a new algebraic	type	<i>saf_declare_algebraic</i>
Declare a new basis	type	<i>saf_declare_basis</i>
Declare a new evaluation	type	<i>saf_declare_evaluation</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Describe a basis	type	<i>saf_describe_basis</i>
Describe an algebraic	type	<i>saf_describe_algebraic</i>
Describe an evaluation	type	<i>saf_describe_evaluation</i>
Find one algebraic	type	<i>saf_find_one_algebraic</i>
Find one basis	type	<i>saf_find_one_basis</i>
Find one evaluation	type	<i>saf_find_one_evaluation</i>
Basis	types	<i>SAF_BasisConstants</i>
Common algebraic	types	<i>SAF_ALGTYPE</i>
Evaluation	Types	<i>SAF_EvalConstants</i>
Find algebraic	types	<i>saf_find_algebraics</i>
Find evaluation	types	<i>saf_find_evaluations</i>
Find relation representation	types	<i>saf_find_relreps</i>
Relation representation	types	<i>SAF_TopoRelRep</i>
Subset relation representation	types	<i>SAF_SubsetRelRep</i>
Specify location of SAF's standard	types database	<i>SAF_REGISTRIES</i>
Conveniently specify a	typical subset	<i>SAF_COMMON</i>
	Ungrab HDF5 I/O library	<i>saf_ungrab_hdf5</i>
Apply a logarithmic scale to a	unit	<i>saf_log_unit</i>
Convenience function for finding a	unit	<i>saf_find_one_unit</i>
Declare a new	unit	<i>saf_declare_unit</i>
Find the not applicable	unit	<i>saf_find_unit_not_applicable</i>
Translate	unit by an offset	<i>saf_offset_unit</i>
Query	unit characteristics	<i>saf_describe_unit</i>
Divide a unit into a	unit definition	<i>saf_divide_unit</i>
Multiply a unit into a	unit definition	<i>saf_multiply_unit</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
Divide a	unit into a unit definition	<i>saf_divide_unit</i>
Multiply a	unit into a unit definition	<i>saf_multiply_unit</i>
Associates a	unit of measure with a specific quantity	<i>saf_quantify_unit</i>
Find	units	<i>saf_find_units</i>
The	universe set handle	<i>SAF_UNIVERSE</i>
Create or	update a non-sharable attribute	<i>saf_put_attribute</i>
	Update database contents	<i>saf_update_database</i>
Determine if a handle is a	valid handle	<i>SAF_VALID</i>
Convert a single	value	<i>_saf_convert</i>
Set the pool size for string return	value allocations	<i>saf_setProps_StrPoolSize</i>
Create a new database property list with default	values	<i>saf_createProps_database</i>
Create a new library property list with default	values	<i>saf_createProps_lib</i>
Standard tri-state	values	<i>SAF_TriState</i>
Make a field handle a C automatic	variable	<i>SAF_Field</i>
Make a field template handle a C automatic	variable	<i>SAF_FieldTmpl</i>
Make a relation handle a C automatic	variable	<i>SAF_Rel</i>
Make a state handle a C automatic	variable	<i>SAF_StateGrp</i>
Make a state template handle a C automatic	variable	<i>SAF_StateTmpl</i>
Make a suite handle a C automatic	variable	<i>SAF_Suite</i>
Serial/Parallel-dependent	variable	<i>SAF_PARALLEL_VAR</i>
Version-dependent	variable	<i>SAF_VERSION_VAR</i>
Get the HDF5	version	<i>saf_getInfo_hdfversion</i>
Get the MPI library	version	<i>saf_getInfo_mpvversion</i>
Get the SAF library	version	<i>saf_getInfo_libversion</i>

Continued on next page

Table 1.1 – continued from previous page

Concept	Key	Reference
	Version Annotation	<i>SAF_VERSION_ANNOT</i>
Major	version number	<i>SAF_VERSION_MAJOR</i>
Minor	version number	<i>SAF_VERSION_MINOR</i>
Returns string representation of	version number	<i>saf_version_string</i>
	Version-dependent variable	<i>SAF_VERSION_VAR</i>
Queries	whether data has been written	<i>saf_data_has_been_written_t</i>
	Wildcards for searching	SAF
	Write a subset relation	<i>saf_write_subset_relation</i>
	Write an alternate index specs to disk	<i>saf_write_alternate_indexspe</i>
	Write out a state	<i>saf_write_state</i>
	Write the data for a field	<i>saf_write_field</i>
	Write topological relation data	<i>saf_write_topo_relation</i>
Queries whether data has been	written	<i>saf_data_has_been_written_t</i>
Conveniently specify a	zone-centered field	<i>SAF_ZONAL</i>

Concept Index

Introduction

This is the Sets and Fields (SAF pronounced “safe”) Application Programming Interface (API) programmer’s reference manual. This manual is organized into *Chapters* where each chapter covers a different, top-level, set of functions (e.g. object and its supporting methods) SAF supports.

There is a decent introduction to the SAF data model in this paper, github.com/markcmiller86/SAF/blob/master/src/safapi/docs/mi

Various API design ideas were taken from this paper, github.com/markcmiller86/SAF/blob/master/src/safapi/docs/necdc_2004_paper.pdf

SAF is designed first and foremost to support scalable I/O of shareable, scientific data.

The key words in this statement are *scalable* and *shareable*.

Scalable means that SAF is designed to operate with high performance from single processor, workstation class machines, to large scale, parallel computing platforms such as are in use in the ASCI program. In turn, this also demands that SAF be portable across a variety of computing platforms. Currently, SAF operates in serial and parallel on Dec, Sun, Linux, IBM-SP2, Intel TeraFlops, SGI-O2k (single box). SAF is also supported in serial on Windows. A good measure of SAF’s performance and portability is derived from its use of industry standard software components such as HDF5 (support.hdfgroup.org/HDF5/doc/index.html) and MPI (www.mpi-forum.org). However, scalable I/O is just one of SAF’s primary goals. Making data *shareable* is another.

Shareable means that if one application uses SAF to write its data, other **wholly independent** applications can easily read **and interpret** that data. Of course, it is not all that impressive if one application can simply read a bunch of **bytes** that another has written. Thus, the key to understanding what *shareable* means is the **and interpret** part. SAF is designed to make it easy for one scientific computing application to interpret another's data. Even more so, SAF is designed to enable this interpretation across a diverse and continually expanding gamut of scientific computing applications. In a nutshell, SAF lays the foundation for very large scale integration of scientific software.

The organizations involved in the development of SAF have plenty of experience with integration on smaller scales with products like netCDF, HDF, PATRAN, SEACAS, Silo and Exodus II. These technologies offer applications a menu of objects; some data structures (e.g. array, list, tree) and/or some mesh objects (e.g. structured-mesh, ucd-mesh, side-sets, etc.). For application developers who use these products, the act of sharing their data is one of browsing the menu. If they are lucky, they will find an object that matches their data and use it. If they are unlucky, they will have to modify their data to put it into a form that matches one of the objects on the menu.

Thus, former approaches to making shareable data suffer from either requiring all clients to use the same data structures and/or objects to represent their data or by resulting in an ever expanding set of incrementally different data structures and/or objects to support each client's slightly different needs. The result is that these products can and have been highly successful within a **small** group of applications who either...

- a) buy into the small menu of objects they do support, or
- b) don't require support for very many new objects (e.g. changes to the supporting library), or
- c) don't expect very many other applications to understand their data

In other words, previous approaches have succeeded in integration on the small scale but hold little promise for integration on the large scale.

The key to integration and sharing of data on the large scale is to find a small set of primitive, yet mathematically meaningful, building blocks out of which descriptions for many different kinds of scientific data can be constructed. In this approach, each new and slightly different kind of data requires the application of the same building blocks to form a slightly different *assembly*. Since every assembly is just a different application of the same building blocks, each is fully supported by existing software. In fact, every assembly of building blocks is simply a *model* for an instance of some scientific data. This is precisely how SAF is designed to operate. For application developers using SAF, the act of sharing their data is one of literally *modeling* their data; not browsing a menu. This modeling is analogous to the user of a CAD/CAM tool when applying constructive solid geometry (CSG) primitives to build an engineering model for some physical part. In a nutshell, the act of sharing data with SAF is one of *scientific data modeling*.

This requires a revolution in the way scientific computing application developers think about their data. The details of bits and bytes, arrays and lists are pushed to the background. These concepts are still essential but less so than the modeling primitives used to characterize scientific data. These modeling primitives are firmly rooted in the mathematics underlying most, if not all, scientific computing applications. By and large, this means the model primitives will embody the mathematical and physical notions of *fields* defined on *base - spaces* or sets.

The term *field* is used to describe any phenomenon that can be mathematically represented, at least locally, as a function over some, often continuous, base-space or domain. The term *base - space* is used to describe an infinite point set, often continuous, with a topological dimension over which fields are defined. Thus, SAF provides three key modeling primitives; fields, sets, and relations between these entities. Fields may represent real physical phenomena such as pressure, stress and velocity. Fields may be related to other fields by integral, derivative or algebraic equations. Fields are defined on sets. Sets may represent real physical objects such parts in an assembly, materials and slide interfaces. And, sets may be related to other sets by set-algebraic equations involving union, intersection and difference.

A full description of modeling principles upon which SAF is based is outside this scope of this programmer's reference manual. User quality tutorials of this material will be forthcoming as SAF evolves. However, the reader should pause for a moment and confirm in his own mind just how general the notions of field and set are in describing scientific data. The columns of an Excel spreadsheet are fields. A time history is a field. The coordinates of a mesh is a field. A plot dump is a whole bunch of related fields. An image is a field. A video is a field. A load curve is a field. Likewise

for sets. An individual node or zone is a set. A processor domain is a set. An element block is a set. A slide line or surface is a set. A part in an assembly is a set. And so on.

Understanding and applying set, field and relation primitives to model scientific data represents a revolutionary departure from previous, menu based approaches. [SAF](#) represents a first cut at a portable, parallel, high performance application programming interface for modeling scientific data. Over the course of development of [SAF](#), the organizations involved have seen the value in applying this technology in several directions...

- a) A publish/subscribe scenario for exchanging data between scientific computing clients, in-situ.
- b) End-user tools for performing set operations and *restricting* fields to subsets of the base space to take a closer look at portions of tera-scale data.
- c) Operators which *transform* data during exchange between clients such as changing the processor decomposition, evaluation method, node-order over elements, units, precision, etc. on a field.
- d) Data consistency checkers which confirm a given bunch of scientific data does indeed conform to the mathematical and physical description that has been ascribed to it by its model. For example, that a volume or mass fraction field is indeed between 0.0 and 1.0, everywhere in its base-space.
- e) MPI-like parallel communication routines pitched in terms of *sets* and *fields* rather than data structures.

And many others.

While each of these areas shows promise, our first goal has been to demonstrate that we can apply this technology to do the same job we previously achieved with mesh-object I/O libraries like Silo and Exodus II. In other words, our first and foremost goal is to demonstrate that we can read and write shareable scientific data files with good performance. Such a capability is fundamental to the success of any organization involved in scientific computing. If we cannot demonstrate that, there is little point in trying to address these other areas of interest.

Environment

A number of environment variables affect the operation of [SAF](#) such as error detections and reporting as well as where predefined types are obtained.

Members

Control Assertion Checking

`SAF_ASSERT_DISABLE` is a symbol defined in `init.c`.

Synopsis:

SAF_ASSERT_DISABLE

Description: There are three environment variables that control, independently, the level or pre-, post- and assert-condition checking done by [SAF](#). They are...

```
SAF_ASSERT_DISABLE
```

```
SAF_PRECOND_DISABLE
```

```
SAF_POSTCOND_DISABLE
```

These three environment variables control the level of assertion, pre-condition and post-condition checking, respectively, that [SAF](#) does during execution. Each is a string valued environment variable with possible values "none", "high", "medium", and "all". For example, a value of "none" for `SAF_ASSERT_DISABLE` means that

none of the assertion checking is disabled. A value of “high” means that all high cost assertions are disabled but medium and low cost assertions are still enabled. A value “medium” means that all high and medium cost assertions are disabled but low cost assertions are still enabled. A value of “all” means that all assertions are disabled. Likewise, `SAF__PRECOND_DISABLE` controls pre-condition checking and `SAF__POSTCOND_DISABLE` controls post-condition checking.

The cost of an assertion, pre-condition or post-condition is specified in terms relative to the [SAF](#) function in which the condition is checked. This means that a simple test for a null pointer in a very simple [SAF](#) function, such as [saf_setProps_LibComm](#), is considered high cost while in a [saf_declare_field](#) it is considered low cost. This is so because the test for a null pointer in [saf_setProps_LibComm](#) relative to the other work [saf_setProps_LibComm](#) does is high cost while that same test in [saf_declare_field](#) is relatively low cost.

In addition to controlling [SAF](#)’s assertion, pre-condition and post-condition checking, these environment variables also control similar checks that go on in the lower layers of [SAF](#). These lower layers do not have high, medium and low check costs. Instead, they can either be turned on or off. The checks are performed if `SAF__ASSERT_DISABLE` is “none”, and not performed otherwise.

Assertion, pre-condition and post-condition checking has a marked effect on performance. To obtain maximum performance, all checks should be turned off using

```
1 setenv SAF_ASSERT_DISABLE all
2 setenv SAF_PRECOND_DISABLE all
3 setenv SAF_POSTCOND_DISABLE all
```

or

```
1 env SAF_ASSERT_DISABLE=all \
2     SAF_PRECOND_DISABLE=all \
3     SAF_POSTCOND_DISABLE=all a.out ...
```

For each of these environment variables, if it does not exist, [SAF](#) will set the default values depending on whether the library was compiled for production or development. For a production compile, the default values for all three environment settings are “all” meaning that the error checking for assertion, pre-condition and post-condition checking is set for maximal performance. For a development compile, the default setting for all three is “none” meaning it is set for maximal error checking.

See Also:

- [saf_declare_field](#): 16.12: *Declare a field*
- [saf_setProps_LibComm](#): 5.7: *Set the MPI communicator for the library*
- [Environment](#): Introduction for current chapter

Control Reporting of Error Messages

`SAF__ERROR_REPORTING` is a symbol defined in `init.c`.

Synopsis:

SAF__ERROR_REPORTING

Description: This is a string valued environment variable that may be set to one of the following values...

```
1 none
```

Means no error reporting. [SAF](#) does not print error messages. This is the default for a production compile but may be overridden at anytime by use of this environment variable.


```
stderr
```

Means SAF sends its error messages to the stderr stream. This is the default for a serial, development compile. See below for the default for a parallel, development compile. If this mode is selected in parallel, SAF will prepend each message with the rank of the MPI task in the communicator used to initialize SAF (see *saf_init*) to each line of output in this file.

```
file: /name/ // no white space
```

Where *name* is a file name, this means SAF will open a stream by this name and send its error messages to this stream. In parallel, SAF will prepend the rank of the task in the communicator used to initialize SAF (see *saf_init*) to each line of output to this file. However, the order of task's output to this file is indeterminate.

```
procfile: /prefix/,/fmt/,/suffix/ // no white space
```

where *prefix* and *suffix* are parts of a name and *fmt* is a printf style integer format designation for including the task number in the name. For example, in 'procfile:saf_,%03d,.log', the prefix is 'saf_', task number format designation is '%03d' and the suffix is '.log'. If this mode is selected in a serial run, the task number format designation will be ignored. A minor issue with this form of error logging is that it generates one file for each task. If you have a 1,000 task run, you get 1000 files. However, it does keep each task's outputs separate, unlike the preceding mode. However, the following mode gets around this problem by generating only a single log file and forcing each proc to write to only a given segment of the file.

See Also:

- *saf_init*: 4.3: Initialize the library
- *Environment*: Introduction for current chapter

Control Postcondition Checking

SAF_POSTCOND_DISABLE is a symbol defined in init.c.

Synopsis:

SAF_POSTCOND_DISABLE

Description: See SAF__ASSERT_DISABLE

See Also:

- *Environment*: Introduction for current chapter

Control Precondition Checking

SAF_PRECOND_DISABLE is a symbol defined in init.c.

Synopsis:

SAF_PRECOND_DISABLE

Description: See SAF__ASSERT_DISABLE

See Also:

- *Environment*: Introduction for current chapter

Specify location of SAF's standard types database

SAF_REGISTRIES is a symbol defined in init.c.

Synopsis:

SAF_REGISTRIES

Description: This is a string valued environment variable that holds colon (':') separated list of pathnames files from where **SAF** will obtain predefined type definitions. If this variable is not set, **SAF** will build an use a transient database containing a minimal set of pre-defined types. In typical usage, this variable need not be set.

By default, **SAF** will generate a minimal, memory resident registry that is destroyed when saf is finalized. This permits **SAF** to operate in such a way that it does not need to access some registry file on disk somewhere to properly initialize. However, other, disk resident registry files can be opened if either the env. variable, SAF__REGISTRIES, is set and/or the client has specified a specific registry with the initialization properties.

Files specified by the SAF__REGISTRIES env. variable are first in the list followed by those specified by the initialization properties. In this way, if SAF__REGISTRIES is specified, the definitions of symbols there take precedence over those that may also exist in the file(s) specified by the initialization properties.

If SAF__REGISTRIES is set to the string "implicit", it will look for a file named *Registry.saf* in the *usual places*, namely in the current working directory, then in the user's home directory and finally in the **SAF** installation directory.

Errors will be reported for registries specified explicitly by environment variables and/or calls to saf_SetProps_Registry, but not for the implicit locations. A warning will be issued if no registry can be found at all.

Finally, if SAF__REGISTRIES is set to the string "none", then no registries will be opened, not even the minimal, memory resident one. If SAF__REGISTRIES is set to the string "default" then only the minimal registry will be opened. That is, you can force **SAF** to ignore all registries specified by a client through the initialization properties by setting SAF__REGISTRIES to "default".

Note that the order in which filenames are specified is important. When **SAF** needs to look up a pre-defined datatype, it searches its known registries in the order in which they were specified in SAF__REGISTRIES, then those specified with *saf_setProps_Registry*. **SAF** returns the first matching referenced type.

See Also:

- *Environment*: Introduction for current chapter

Built-in registry name

SAF_REGISTRY_SAVE is a symbol defined in init.c.

Synopsis:

SAF_REGISTRY_SAVE

Description: Normally **SAF** will create a minimal object registry that exists only in memory. However, if this environment variable is set to the name of a file, then **SAF** will save the built-in object registry in that file. This is intended only to be used for debugging to make sure that the object registry file's contents are what is expected.

See Also:

- *Environment*: Introduction for current chapter

Trace SAF API calls and times

SAF_TRACING is a symbol defined in init.c.

Synopsis:

SAF_TRACING

Description: This is a string valued environment variable used to control API call tracing in SAF. It may be set to any one of the values described below. Note that API call tracing is logged to the same file specified by SAF__ERROR_REPORTING. However, if SAF__ERROR_REPORTING is set to “none” and SAF__TRACING is not also “none”, SAF will log its API tracing to stderr. Currently, SAF only logs entrances to SAF API calls, not exits.

Note: Since the bulk of SAF’s API is collective, only processor 0 actually prints any trace information.

```
1 none
```

This is the default. It means that no API tracing will be generated.

```
1 times
```

This setting will record the cumulative amount of time spent in saf_read_xxx calls and saf_write_xxx calls as compared to the total time between calls to saf_init and saf_final. The times recorded are wall clock seconds. Entrances to functions WILL NOT be reported. However, during saf_final, the cumulative timers for time spent in reads and writes will be reported.

```
1 public
```

Public API calls will be logged to whatever file SAF is also reporting errors to.

```
1 public,times
```

Same as “public” but SAF will also output wall clock times since the last API call was entered. SAF will report the delta since the last call and the absolute time, starting from 0. The times reported are WALL CLOCK seconds, not CPU seconds. Thus, if there are other activities causing SAF to run more slowly then it will be reflected in the times SAF reports.

```
1 public,private
```

Both public and private API calls are logged.

```
1 public,private,times
```

Both public and private API calls are logged along with timing information.

Finally, if SAF__TRACING is set to a valid value other than “none”, SAF will also invoke HDF5’s tracing facilities. However, HDF5’s tracing facilities WILL NOT take effect unless the environment var H5_DEBUG is also defined in the environment. Thus, HDF5’s tracing can be turned on/off separately by setting or unsetting the H5_DEBUG environment variable.

See Also:

- *saf_final*: 4.2: Finalize access to the library
- *saf_init*: 4.3: Initialize the library
- *Environment*: Introduction for current chapter

Error Handling

SAF can be used either in an exception-catching programming paradigm or test the return code programming paradigm. SAF will either throw exceptions or return error codes depending on a library property (see *saf_setProps_ErrorMode*).

See **Environment** section where environment variables affecting error checking, etc. are discussed.

In addition, the client can control if and how error messages are reported and whether certain kinds of errors are detected. In every function, SAF does work to detect problematic conditions. However, the cost of this detection work is weighted relative to the real work of the function using a low, medium and high weighting scheme.

There are several macros available to use in an exception handling programming style rather than a test the return value style. The basic structure of an exception handling style is...

```
1  SAF_TRY_BEGIN {           // begin an exception/catch block
2      ...                   // put your saf calls here
3  } SAF_CATCH {             // begin the catch block
4      // catch a particular error
5      SAF_CATCH_ERR(SAF_WRITE_ERROR) {
6          ...               // specific error handling here
7      }
8      SAF_CATCH_ALL {       // catch any error here
9          ...               // generic error handling here
10     }
11 } SAF_TRY_END;            // end the exception/catch block
```

Members

Begin a the CATCH part of a TRY/CATCH block

SAF_CATCH is a symbol defined in saf.h.

Synopsis:

SAF_CATCH

Description: Use this macro to demarcate the beginning of the error catching portion of a TRY/CATCH block

See Also:

- *Error Handling*: Introduction for current chapter

Begin a block of error handling code for all errors

SAF_CATCH_ALL is a symbol defined in saf.h.

Synopsis:

SAF_CATCH_ALL

Description: Use this macro to demarcate the beginning of a block of code that catches all errors in the error catching portion of a TRY/CATCH block

See Also:

- *Error Handling*: Introduction for current chapter

Begin a block of error handling code

`SAF_CATCH_ERR` is a macro defined in `saf.h`.

Synopsis:

`SAF_CATCH_ERR` (`err`)

Description: Use this macro to demarcate the beginning of a block of code that catches a specific error, `err`, in the error catching portion of a `TRY/CATCH` block

Issues: I am not sure I have confirmed that the catching does, in fact, fall through from a specific catch to a next specific catch or to the `ALL` case?

If we changed the `SAF__CATCH_ERR` macro to test the bit(s) of the argument, rather than equality we'd be able to catch several specific errors in that block.

See Also:

- *Error Handling*: Introduction for current chapter

Begin a TRY/CATCH block

`SAF_TRY_BEGIN` is a symbol defined in `saf.h`.

Synopsis:

`SAF_TRY_BEGIN`

Description: Use this macro to demarcate the beginning of a block of code in which exceptions shall be caught.

Issues: We should be clear about what happens in this block if the library properties are set to `SAF__ERRMODE_RETURN` rather than `SAF__ERRMODE_THROW`.

Should we add code here to check to make sure *saf_init* is called first. I think that would make some sense.

See Also:

- *saf_init*: 4.3: Initialize the library
- *Error Handling*: Introduction for current chapter

End a TRY/CATCH block

`SAF_TRY_END` is a symbol defined in `saf.h`.

Synopsis:

`SAF_TRY_END`

Description: Use this macro to end a `TRY/CATCH` block

See Also:

- *Error Handling*: Introduction for current chapter

Error codes returned by the library

`SAF_error_t` is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

`SAF_FATAL_ERROR`: Any fatal error.

`SAF_MEMORY_ERROR`: A memory-related error.

`SAF_FILE_ERROR`: File-related errors.

`SAF_CONTEXT_ERROR`: Context errors.

`SAF_LOOKUP_ERROR`: Name lookup errors.

`SAF_MAPPING_ERROR`: Mapping errors.

`SAF_WRITE_ERROR`: File write errors.

`SAF_DEBUG_ERROR`: Debugging messages.

`SAF_CONSTRAINT_ERROR`: Failed constraints.

`SAF_PARAMETER_ERROR`: Function parameter errors.

`SAF_COMMUNICATION_ERROR`: MPI-related errors.

`SAF_READ_ERROR`: File read errors.

`SAF_NOTIMPL_ERROR`: Functionality has not been implemented.

`SAF_BADHNDL_ERROR`: Object handle errors.

`SAF_MISC_ERROR`: Miscellaneous errors.

`SAF_SIZE_ERROR`: Size-related errors.

`SAF_PMODE_ERROR`: Errors in the parallel mode argument to a function.

`SAF_ASSERTION_ERROR`: Failed assertions.

`SAF_PRECONDITION_ERROR`: Failed preconditions.

`SAF_POSTCONDITION_ERROR`: Failed postconditions.

`SAF_GENERIC_ERROR`: Generic errors.

`SAF_SSLIB_ERROR`: SSlib related errors

Description: These C preprocessor symbols define an integer bitmask where each bit represents an error condition.

See Also:

- *Error Handling*: Introduction for current chapter

Return a pointer to an error string

`saf_error_str` is a function defined in `error.c`.

Synopsis:

`char * saf_error_str (void)`

Description: `saf_error_str` returns a pointer to a string containing the most recent error message.

See Also:

- *Error Handling*: Introduction for current chapter

Library Initialization

To interact with **SAF**, the client must call *saf_init*. To end interaction with **SAF**, the client must call *saf_final*.

In parallel, **SAF** will not call `MPI_Init` or `MPI_Finalize` on behalf of the client. It is the client's responsibility to initialize and finalize MPI. MPI should be initialized before calling *saf_init* and finalized after calling *saf_final*.

The only **SAF** functions that can be called outside of an enclosing *saf_init/saf_final* pair are functions to create and set library properties (see **Library Properties**).

SAF provides a link-time library and include file consistency check that will generate an undefined reference link-time error for a symbol whose name is of the form "SAF__version_X_Y_Z", if the library and include files are not consistent.

Members

Initialize the built-in object registry

`_saf_gen_stdtypes` is a function defined in `init.c`.

Synopsis:

```
static herr_t _saf_gen_stdtypes (ss_file_t *stdtypes)
```

Description: The built-in object registry is a **SAF** file that contains definitions for objects that are frequently used. This function also caches some of those objects in global variables.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the library communicator.

Issues: Since other files probably point into the built-in registry and they do so by specifying a table row number, we have to be sure that we always create the built-in registry the same way. It is important that we don't move objects around in the tables over the life of the file—only add new objects to the end of the table.

See Also:

- *Library Initialization*: Introduction for current chapter

Finalize access to the library

`saf_final` is a function defined in `init.c`.

Synopsis:

```
void saf_final (void)
```

Description: A call to `saf_final` terminates the client's interaction with the **SAF** library. Any open databases and supplemental files are closed and all memory allocated by the library is freed. Calling this function when the library is already in a finalized state has no effect. This function should not be called before the library has been initialized.

This call is mainly just a wrapper for a call to `_saf_final` so that we can distinguish between a situation in which *saf_final* is called by `exit` and one in which the client made the call explicitly.

Parallel Notes: This function must be called collectively across all processes in the library's communicator, which was set in the *saf_init* call. Furthermore, the client should not call `MPI_Finalize` prior to calling `saf_final`.

SAF does try to detect this condition and report its occurrence before aborting. However, on some platforms, this is simply not possible and the client might silently hang with no indication as to the cause.

See Also:

- *saf_init*: 4.3: *Initialize the library*
- *Library Initialization*: Introduction for current chapter

Initialize the library

`saf_init` is a macro defined in `SAFinit.h`.

Synopsis:

`saf_init` (PROPERTIES)

Description: The `saf_init` function must be called by the client to initialize the client's interaction with the library and should be called before any other SAF-API function except functions that set the properties to be passed in the `saf_init` call.

Calling `saf_init` when the library is initialized has absolutely no effect, even when a new, different list of properties is specified.

The counterpart of `saf_init` is *saf_final*, which releases all resources held by the library. The `saf_init` function must be called after *saf_final* if the client desires to interact with the library again.

Since all SAF clients are required to call `saf_init`, we've chosen to wrap that function in a macro which also makes a reference to a global variable whose name is derived from the SAF version number. This variable is declared in the SAF library so that if an application is compiled with SAF header files which have a different version than the SAF library a link-time error will result. A version mismatch will result in an error similar to *undefined reference to "SAF__version_1_4_0"* from the linker.

Return Value: The constant `SAF__SUCCESS` is returned for success; errors are returned as other values or by exception, depending on the setting of the error handling property in `PROPERTIES` (the default is to return an error number).

Parallel Notes: In parallel, `saf_init` is collective and must be called by all processes in the library's communicator, which is `MPI_COMM_WORLD` by default. All processes must initialize the library with the same property values, although each may pass its own `PROPERTIES` argument.

If a new communicator is specified in the `PROPERTIES` argument then it will become the communicator for any database which doesn't override this communicator. It is the maximal communicator in the sense that no database can be opened on a set of processors which is not a subset of those in the communicator declared in the properties passed here.

Issues: Verify that the set of processors which must participate in this call is either `MPI_COMM_WORLD` or the communicator passed in the `PROPERTIES`.

We might want to communicate to confirm that all procs pass the same properties.

See the private function, `_saf_init`, for the *real* implementation of this function

Calling `saf_init` after *saf_final* is currently not supported.

See Also:

- *saf_final*: 4.2: *Finalize access to the library*
- *Library Initialization*: Introduction for current chapter

Library Properties

There are a number of properties that affect the behavior of the library. For a general description of how properties are used (See **Properties**).

The functions to set library properties are the only functions that may be called prior to calling *saf_init*.

Members

Create a new library property list with default values

saf_createProps_lib is a function defined in libprops.c.

Synopsis:

SAF_LibProps * **saf_createProps_lib** (void)

Description: This function creates a library property list which can be passed to the *saf_init* function. All properties in this list are set to their default values...

```
1 ErrFunc = NULL;
```

```
1 ErrMsgMode = <ignored>;
```

```
1 ErrLoggingMode = none;
```

```
1 ErrorMode = SAF_ERRMODE_RETURN;
```

```
1 LibComm = MPI_COMM_WORLD;
```

```
1 StrMode = SAF_STRMODE_LIB;
```

```
1 StrPoolSize = 4096;
```

Return Value: A handle to a new library properties list initialized to default values. NULL on failure or an exception is thrown depending on the error handling library property currently in effect (See **Properties**).

Parallel Notes: This function must be called collectively by all processors in MPI_COMM_WORLD.

Issues: Since this function is called before *saf_init*, the only communicator we can use to work correctly is MPI_COMM_WORLD. However, if we allowed the client to pass a communicator here, then we could avoid that.

See Also:

- *saf_init*: 4.3: Initialize the library
- *Library Properties*: Introduction for current chapter

Free library property list

saf_freeProps_lib is a function defined in libprops.c.

Synopsis:

SAF_LibProps * **saf_freeProps_lib** (SAF_LibProps **properties*)

Description: Releases resources inside the library property list and frees the property list that was allocated in *saf_createProps_lib*.

Return Value: Always returns null.

Parallel Notes: Independent

Issues: Releasing the resources used by the property list was never implemented and so is not implemented here yet either.

See Also:

- *saf_createProps_lib*: 5.1: *Create a new library property list with default values*
- *Library Properties*: Introduction for current chapter

Turn off aborts

`saf_setProps_DontAbort` is a function defined in `libprops.c`.

Synopsis:

int **saf_setProps_DontAbort** (SAF_LibProps **properties*)

Formal Arguments:

- `properties`: The library property list which will be modified by this function

Description: In certain cases, the library will abort when it encounters an error condition. This function turns that behavior off.

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)

Parallel Notes: This function can be called independently. Nonetheless, properties passed to *saf_init* must be consistent on all processors.

See Also:

- *Library Properties*: Introduction for current chapter

Specify a callback for error conditions

`saf_setProps_ErrFunc` is a function defined in `libprops.c`.

Synopsis:

int **saf_setProps_ErrFunc** (SAF_LibProps **properties*, SAF_ErrFunc *func*)

Formal Arguments:

- `properties`: The library property list which will be modified by this function (See **Properties**).
- `func`: The callback to invoke when an error occurs.

Description: The specified function (or no function if `func` is the null pointer) will be called when the library is recovering from an error condition. The default is that no callback is invoked.

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: Not implemented yet (always returns `SAF__NOTIMPL_ERROR`).

See Also:

- *Library Properties*: Introduction for current chapter

Set the error logging mode

`saf_setProps_ErrorLogging` is a function defined in `libprops.c`.

Synopsis:

```
int saf_setProps_ErrorLogging (SAF_LibProps *properties, const char *mode)
```

Formal Arguments:

- `properties`: The library property list which will be modified by this function (See **Properties**).
- `mode`: The error logging mode.

Description: This library property controls how the library reports errors. See section on environment variables where the environment variable `SAF__ERROR_REPORTING` is described.

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Library Properties*: Introduction for current chapter

Set the library error handling mode

`saf_setProps_ErrorMode` is a function defined in `libprops.c`.

Synopsis:

```
int saf_setProps_ErrorMode (SAF_LibProps *properties, SAF_ErrMode mode)
```

Formal Arguments:

- `properties`: The library property list which will be modified by this function (See **Properties**).
- `mode`: The new error handling mode. Valid values are `SAF__ERRMODE_RETURN` (the default) and `SAF__ERRMODE_THROW`.

Description: The library normally handles error conditions by causing the erring function to return a non-zero error number. However, the error mode can be set to `SAF__ERRMODE_THROW` which causes the erring function to throw an exception instead.

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently.

Known Bugs: This function sometimes returns an error instead of throwing an exception when the library error mode is `SAF__ERRMODE_THROW`.

See Also:

- *Library Properties*: Introduction for current chapter

Set the MPI communicator for the library

`saf_setProps_LibComm` is a function defined in `libprops.c`.

Synopsis:

```
int saf_setProps_LibComm (SAF_LibProps *properties, MPI_Comm communicator)
```

Formal Arguments:

- `properties`: The library property list which will be modified by this function (See **Properties**).
- `communicator`: The new MPI communicator.

Description: This function sets the MPI communicator in the specified library property list to `communicator` (the default is `MPI_COMM_WORLD`). After this property list is used to initialize the library by calling `saf_init`, it will become the communicator for all collective calls. However, a database can override this communicator in the `saf_open_database` call.

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently. It is not defined in a non-parallel version of the library.

Issues: Should this function even be defined if the library is not compiled for parallel. My reasoning is that it would only be called if the client is compiled for parallel and therefore it only makes sense to link the application if the SAF-API is also compiled for parallel. Getting a link error is probably better than a runtime error for two reasons: the error comes earlier (what if the application did a day of number crunching before trying I/O), and we can guarantee that it's an error (what if the client failed to check return values).

Known Bugs: This function sometimes returns an error instead of throwing an exception when the library error mode is `SAF__ERRMODE_THROW`.

See Also:

- `saf_init`: 4.3: *Initialize the library*
- `saf_open_database`: 7.3: *Open a database*
- *Library Properties*: Introduction for current chapter

Specify registry file

`saf_setProps_Registry` is a function defined in `libprops.c`.

Synopsis:

```
int saf_setProps_Registry (SAF_LibProps *properties, const char *name)
```

Formal Arguments:

- `properties`: Library properties (See **Properties**)

- `name`: Name of object registry file

Description: The registry consists of one or more SAF databases which will be consulted when an object query cannot be satisfied from the primary database. For instance, if the client performed a [saf_find_one_unit](#) to obtain a handle for something called “millimeter” and the find operation could not find any matching definition in the specified database, then each SAF registry database will be queried until a definition can be found or all registered registry files have been exhausted.

The library consults registries in the following order: First all files specified with the `SAF__REGISTRIES` environment variable (if the variable is set to the word *none* then no registries are consulted). The environment variable can specify multiple registries by separating them from one another with colons. Second, all files registered with ‘`saf_setProps_Registry`’ are searched in the order they were specified. Third, a file by the name of `Registry.saf` in the current working directory, then the home directory (as specified by the environment variable ‘`HOME`’). Last, SAF will check for a file named `Registry.saf` in the data installation directory specified during the saf configuration with the `-datadir` switch (defaults to ‘`/usr/local/share`’).

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)
- `name` is required to be non-empty. (low-cost)

Return Value: A non-negative value on success. Otherwise this function either returns a negative error number or throws an exception, depending on the value of the library’s error handling property.

Issues: The library does not attempt to open the registry file until a database is opened. Therefore, specifying an invalid file name here will not result in an error until the call to [saf_open_database](#) is made.

See Also:

- [saf_find_one_unit](#): 21.4: Convenience function for finding a unit
- [saf_open_database](#): 7.3: Open a database
- [Library Properties](#): Introduction for current chapter

Set string allocation style

`saf_setProps_StrMode` is a function defined in `libprops.c`.

Synopsis:

```
int saf_setProps_StrMode (SAF_LibProps *properties, SAF_StrMode mode)
```

Formal Arguments:

- `properties`: The library property list which will be modified by this function (See [*Properties*](#)).
- `mode`: The string allocation mode, one of `SAF__STRMODE_LIB`, `SAF__STRMODE_CLIENT`, or `SAF__STRMODE_POOL`.

Description: By default, the library allocates memory for returned strings and the client is expected to free that memory when it is no longer needed. However, by calling `saf_setProps_StrMode` the client can set library properties which change this mode of operation. Possible values are `SAF__STRMODE_LIB`, the default; `SAF__STRMODE_CLIENT`, which means that the client will always allocate space for the string return value (and free it also); and `SAF__STRMODE_POOL`, which means that the library will allocate space for the string return values from a recirculating pool, freeing the memory which has been least recently allocated. The [saf_setProps_StrPoolSize](#) function can be used to change the default size of the pool. (See [*Returned Strings*](#)).

Preconditions:

- `properties` must be a valid library properties handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently.

See Also:

- *saf_setProps_StrPoolSize*: 5.10: *Set the pool size for string return value allocations*
- *Library Properties*: Introduction for current chapter

Set the pool size for string return value allocations

`saf_setProps_StrPoolSize` is a function defined in `libprops.c`.

Synopsis:

int **saf_setProps_StrPoolSize** (SAF_LibProps **properties*, int *size*)

Formal Arguments:

- *properties*: The library property list which will be modified by this function (See **Properties**).
- *size*: The new pool size.

Description: If the library string mode property is `SAF__STRMODE_POOL` then string return values are allocated by the library from a pool with some number of entries (4096 by default). When all entries of the pool have been allocated for return values then the library begins freeing the least recently allocated entry.

Preconditions:

- *properties* must be a valid library properties handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently.

Known Bugs: This function sometimes returns an error instead of throwing an exception when the library error mode is `SAF__ERRMODE_THROW`.

The pool and its entries are not freed by *saf_final*.

See Also:

- *saf_final*: 4.2: *Finalize access to the library*
- *Library Properties*: Introduction for current chapter

Path Info

These functions are designed to permit a **SAF** client to obtain information about a named file without actually requiring the client to successfully open the file with a call to *saf_open_database*. This is particularly useful for clients that need to be responsive to different versions of **SAF** databases.

All the functions in this part of the interface return information about the file identified in the *saf_readInfo_path* call and how that file was generated. The typical usage of these functions is to first obtain information about the given file with a call to *saf_readInfo_path*. Then, use the various query functions here to obtain specific information about the file name (path) passed into *saf_readInfo_path* and finally to free up any resources with a call to *saf_freeInfo_path*.

Members

Free SAF_PathInfo

`saf_freeInfo_path` is a function defined in `info.c`.

Synopsis:

```
void saf_freeInfo_path (SAF_PathInfo info)
```

Formal Arguments:

- `info`: a `SAF_PathInfo` object obtained from a *`saf_readInfo_path`* call.

Description: This function is used to free a `SAF_PathInfo` object.

Preconditions:

- `info` must be non-NULL. (low-cost)

Parallel Notes: Independent semantics only

See Also:

- *`saf_readInfo_path`*: 6.10: *Load information from the specified path*
- *Path Info*: Introduction for current chapter

Get stat error message

`saf_getInfo_errmsg` is a function defined in `info.c`.

Synopsis:

```
const char * saf_getInfo_errmsg (const SAF_PathInfo info)
```

Formal Arguments:

- `info`: [IN] database info object obtained from a *`saf_readInfo_path`* call.

Description: This function returns error message associated with any stat errors on the path specified in `saf_getInfo_path`. Caller should **not** free the returned string.

Parallel Notes: Independent semantics only

See Also:

- *`saf_readInfo_path`*: 6.10: *Load information from the specified path*
- *Path Info*: Introduction for current chapter

Get the HDF5 version

`saf_getInfo_hdfversion` is a function defined in `info.c`.

Synopsis:

```
void saf_getInfo_hdfversion (const SAF_PathInfo info, int *major, int *minor, int *patch, char *annot)
```

Formal Arguments:

- `info`: [IN] database info object obtained from a *`saf_readInfo_path`* call.

- `major`: [OUT] major version number. Ignored if NULL.
- `minor`: [OUT] minor version number. Ignored if NULL.
- `patch`: [OUT] patch (aka “release”) version number. Ignored if NULL.
- `annot`: [OUT] annotation string of at most 8 chars including null. Caller allocates. Ignored if NULL.

Description: This function is used to obtain HDF5 library version information from a given path queried with `saf_getInfo_path`.

Preconditions:

- `info` must be non-NULL. (low-cost)

Parallel Notes: Independent semantics only

See Also:

- [*saf_readInfo_path*](#): 6.10: *Load information from the specified path*
- [*Path Info*](#): Introduction for current chapter

Check if path is an HDF5 file

`saf_getInfo_isHDFfile` is a function defined in `info.c`.

Synopsis:

```
int saf_getInfo_isHDFfile (const SAF_PathInfo info)
```

Formal Arguments:

- `info`: [IN] database info object obtained from a [*saf_readInfo_path*](#) call.

Description: This function returns true if the path queried with `saf_getInfo_path` is an HDF5 file. That is, if `H5Fopen` can succeed on it.

Parallel Notes: Independent semantics only

See Also:

- [*saf_readInfo_path*](#): 6.10: *Load information from the specified path*
- [*Path Info*](#): Introduction for current chapter

Check if path is a SAF database

`saf_getInfo_isSAFdatabase` is a function defined in `info.c`.

Synopsis:

```
int saf_getInfo_isSAFdatabase (const SAF_PathInfo info)
```

Formal Arguments:

- `info`: [IN] database info object obtained from a [*saf_readInfo_path*](#) call.

Description: This function returns true if the path queried with `saf_getInfo_path` is a [*SAF*](#) database (of any version).

Parallel Notes: Independent semantics only

See Also:

- *saf_readInfo_path*: 6.10: Load information from the specified path
- *Path Info*: Introduction for current chapter

Get the SAF library version

`saf_getInfo_libversion` is a function defined in `info.c`.

Synopsis:

```
void saf_getInfo_libversion (const SAF_PathInfo info, int *major, int *minor, int *patch, char *annot)
```

Formal Arguments:

- `info`: [IN] database info object obtained from a *saf_readInfo_path* call.
- `major`: [OUT] major version number. Ignored if NULL.
- `minor`: [OUT] minor version number. Ignored if NULL.
- `patch`: [OUT] patch (aka “release”) version number. Ignored if NULL.
- `annot`: [OUT] annotation string of at most 8 chars including null. Caller allocates. Ignored if NULL.

Description: This function is used to obtain SAF library version information from a given path queried with `saf_getInfo_path`.

Preconditions:

- `info` must be non-NULL. (low-cost)

Parallel Notes: Independent semantics only

See Also:

- *saf_readInfo_path*: 6.10: Load information from the specified path
- *Path Info*: Introduction for current chapter

Get the MPI library version

`saf_getInfo_mpiversion` is a function defined in `info.c`.

Synopsis:

```
void saf_getInfo_mpiversion (const SAF_PathInfo info, int *major, int *minor, int *patch, char *annot)
```

Formal Arguments:

- `info`: [IN] database info object obtained from a *saf_readInfo_path* call.
- `major`: [OUT] major version number. Ignored if NULL.
- `minor`: [OUT] minor version number. Ignored if NULL.
- `patch`: [OUT] patch (aka “release”) version number. Ignored if NULL.
- `annot`: [OUT] annotation string of at most 8 chars including null. Caller allocates. Ignored if NULL.

Description: This function is used to obtain MPI library version information from a given path queried with `saf_getInfo_path`.

Preconditions:

- `info` must be non-NULL. (low-cost)

Parallel Notes: Independent semantics only

See Also:

- [*saf_readInfo_path*](#): 6.10: *Load information from the specified path*
- [*Path Info*](#): Introduction for current chapter

Obtain permissions of path

`saf_getInfo_permissions` is a function defined in `info.c`.

Synopsis:

int **saf_getInfo_permissions** (const SAF_PathInfo *info*)

Formal Arguments:

- `info`: [IN] database info object obtained from a [*saf_readInfo_path*](#) call.

Description: This function returns the permissions of the path queried with `saf_getInfo_path`.

Parallel Notes: Independent semantics only

See Also:

- [*saf_readInfo_path*](#): 6.10: *Load information from the specified path*
- [*Path Info*](#): Introduction for current chapter

Check if any stat errors occurred

`saf_getInfo_staterror` is a function defined in `info.c`.

Synopsis:

int **saf_getInfo_staterror** (const SAF_PathInfo *info*)

Formal Arguments:

- `info`: [IN] database info object obtained from a [*saf_readInfo_path*](#) call.

Description: This function returns true if any errors occurred stating the path specified in [*saf_readInfo_path*](#).

Parallel Notes: Independent semantics only

See Also:

- [*saf_getInfo_errmsg*](#): 6.2: *Get stat error message*
- [*saf_readInfo_path*](#): 6.10: *Load information from the specified path*
- [*Path Info*](#): Introduction for current chapter

Load information from the specified path

`saf_readInfo_path` is a function defined in `info.c`.

Synopsis:

SAF_PathInfo **saf_readInfo_path** (const char **path*, int *independent*)

Formal Arguments:

- `path`: [IN] path of a file to get the info for
- `independent`: [IN] A flag for independent operation. If non-zero, perform the work and return the results only on the calling processor. Otherwise, this function must be called collectively by all processors in the communicator used to init the SAF library. In other words, call this function from one processor with a non-zero value for this argument or call it on all processors with a zero argument on all processors. Note also that if this call is made independently, then all succeeding calls involving the returned `SAF__PathInfo` object must be made independently and by the same processor.

Description: This function is used to query a file in the filesystem to obtain information about it. The information is returned in the `SAF__PathInfo` object. Once this object is obtained, one can query a number of things about the database using various query functions. Once you are done, remember to free the resources with *[saf_freeInfo_path](#)*.

Note that any path, except `NULL`, is acceptable to pass here. This function will obtain as much information about the specified path as possible. If the path either does not exist or the user does not have permission to read it, that fact can be obtained from *[saf_getInfo_permissions](#)*. Likewise, if the path does exist, is readable and is a SAF database, the particular version information, etc. can be obtained from functions like *[saf_getInfo_libversion](#)*, etc. A variety of failure modes are detectable by calling various functions in this part of the interface to learn about the kind of file to which *path* refers.

Preconditions:

- Path must be non-`NULL`. (low-cost)

Parallel Notes: Independent semantics only

See Also:

- *[saf_freeInfo_path](#)*: 6.1: Free `SAF_PathInfo`
- *[saf_getInfo_libversion](#)*: 6.6: Get the SAF library version
- *[saf_getInfo_permissions](#)*: 6.8: Obtain permissions of path
- *[Path Info](#)*: Introduction for current chapter

Databases

A database is an abstraction used to represent the container in which all data that is part of a common, aggregate collection is stored. For a typical simulation code, the database abstraction represents a single container in which **all fields from all time steps** for a given run of the simulation are stored. If, in fact, there are many simulation runs that are part of some larger ensemble of runs, then the database abstraction ought to represent a single container in which **all fields from all time steps from all simulations** are stored.

In our current software, there are two serious limitations with respect to how we implement the database abstraction.

First, no matter what container abstraction we introduce for our clients to read/write SAF field data, they ultimately interact with the resulting data via a number of other tools outside the current scope of the SAF effort. Many of these tools interact with the data as files in the filesystem. Examples are *rm*, *cp*, *ls*, *f_stat*, *ftp*, *diff*, etc. Granted, as files get larger and larger, these tools become unwieldy. These tools provide a view of the data in terms of files. Because of this, our customers have an expectation and a serious requirement to have control over how a database gets implemented in terms of files.

We have no *[saf_del_XXX_handle](#)* functions for databases because the client **always** must call *[saf_open_database](#)* to obtain a database handle and *[saf_close_database](#)* to free a database handle

Members

Database information not available

SAF_NOT_SET_DB is a symbol defined in SAFdb.h.

Synopsis:

SAF_NOT_SET_DB

Description: This constant can be used to indicate that the database is not available or is unknown.

See Also:

- [Databases](#): Introduction for current chapter

Close a database

saf_close_database is a function defined in db.c.

Synopsis:

int **saf_close_database** (SAF_Db **database*)

Formal Arguments:

- *database*: The open database to be closed.

Description: This function closes an open database, *database*, freeing all resources associated with that database.

Preconditions:

- *database* must be a database handle. (low-cost)
- *database* must currently be open. (low-cost)

Return Value: The constant SAF__SUCCESS is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This is a collective, SAF__ALL mode function which should be called across all processes in the database's communicator.

See Also:

- [Databases](#): Introduction for current chapter

Open a database

saf_open_database is a function defined in db.c.

Synopsis:

SAF_Db * **saf_open_database** (const char **path*, SAF_DbProps **properties*)

Formal Arguments:

- *path*: The name of the database.
- *properties*: This argument, if not null, provides database properties that will override the default properties provided by [saf_createProps_database](#).

Description: Opens or creates a database for read and/or write access (depending on `properties`) using the communicator specified in `properties`. The name of the database, `path`, is a file name. The `properties` argument, if not `SAF__DEFAULT_DBPROPS`, provides database properties that will override the default properties set by *saf_createProps_database*.

Preconditions:

- `path` must be non-null. (low-cost)
- `properties` must be a valid handle if supplied. (high-cost)

Return Value: Returns a new handle to the opened database on success; `NULL` on failure (or an exception is raised).

Parallel Notes: This is a collective, `SAF__ALL` mode, call in the communicator specified by the `properties` passed in the call.

Issues: It would be nice to identify the current processor decomposition, if possible. At the moment, we can't. But the idea would be that if we're opening an already existing database, we should search for a `PROCESSOR` collection on the top set(s) such that the size of that collection is equal to the value returned by `MPI_Comm_size` above. In this way, the database could "know" which sets are associated with which processors. At present we don't do this.

See Also:

- *saf_createProps_database*: 8.2: Create a new database property list with default values
- *Databases*: Introduction for current chapter

Update database contents

`saf_update_database` is a function defined in `db.c`.

Synopsis:

```
int saf_update_database (SAF_Db *database)
```

Formal Arguments:

- `database`: The database to update

Description: This function is used to force the library to update the contents of the database to the most recent operation issued by the client. In the case of file I/O, all pending writes will be flushed so that files are consistent with the most recent operation.

Preconditions:

- `database` must be a database handle. (low-cost)
- `database` must not be open for read-only access. (no-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This call is collective across all processes in the MPI communicator used to open the database.

See Also:

- *Databases*: Introduction for current chapter

Database Properties

There are a number of properties that affect the behavior of a database. Each member function of this portion of the API sets a property to be associated with a database to a given value. See the individual member functions for a more

detailed description of the database properties and their meaning. For a general description of how properties are used (See **Properties**).

Members

Default properties

SAF_DEFAULT_DBPROPS is a symbol defined in SAFdbprops.h.

Synopsis:

SAF_DEFAULT_DBPROPS

Description: Identifiers for default properties for databases.

See Also:

- *Database Properties*: Introduction for current chapter

Create a new database property list with default values

saf_createProps_database is a function defined in *dbprops.c*.

Synopsis:

SAF_DbProps * **saf_createProps_database** (void)

Description: This function creates a database property list which can be passed to the *saf_open_database* function. All properties in this list are set to their default values:

```
1  Clobber = false;
2  DbComm = LibComm;
3  ImportFile = $SAF_STD_TYPES_PATH //or
4              std_types.saf //or
5              FILE:~/.std_types.saf //or
6              FILE:$SAF_INSTALL/share/std_types.saf;
7  ReadOnly = false;
```

Return Value: A handle to a new database properties list initialized to default values. Otherwise either an error value is returned or an exception is thrown depending on the error handling library property currently in effect (See **Properties**).

Known Bugs: This function sometimes return an error instead of throwing an exception when the library error mode is SAF__ERRMODE_THROW.

See Also:

- *saf_open_database*: 7.3: *Open a database*
- *Database Properties*: Introduction for current chapter

Free database property list

saf_freeProps_database is a function defined in *dbprops.c*.

Synopsis:

SAF_DbProps * **saf_freeProps_database** (SAF_DbProps **properties*)

Description: Releases resources inside the database property list and frees the property list that was allocated in *saf_createProps_database*.

Return Value: Always returns null.

Parallel Notes: Independent

Issues: Releasing the resources used by the property list was never implemented and so is not implemented here yet either.

See Also:

- *saf_createProps_database*: 8.2: *Create a new database property list with default values*
- *Database Properties*: Introduction for current chapter

Clobber an existing database on open

`saf_setProps_Clobber` is a function defined in `dbprops.c`.

Synopsis:

int **saf_setProps_Clobber** (SAF_DbProps **properties*)

Formal Arguments:

- *properties*: The database property list which will be modified by this function (See **Properties**).

Description: When *saf_open_database* is called with the name of an existing database the default action is to open that database. New data will be appended to it. However, if this property is set, then the existing database will be unlinked before it is opened.

Preconditions:

- *properties* must not be null. (high-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently.

Known Bugs: This function sometimes return an error instead of throwing an exception when the library error mode is `SAF__ERRMODE_THROW`.

See Also:

- *saf_open_database*: 7.3: *Open a database*
- *Database Properties*: Introduction for current chapter

Specify MPI database communicator

`saf_setProps_DbComm` is a function defined in `dbprops.c`.

Synopsis:

int **saf_setProps_DbComm** (SAF_DbProps **properties*, MPI_Comm *communicator*)

Formal Arguments:

- *properties*: The database property list which will be modified by this function (See **Properties**).
- *communicator*: The MPI communicator.

Description: When a database is opened it uses the library communicator by default. However, this function can be called to set up a different MPI communicator to open a database. However, that communicator **must be** a subset of the communicator used to initialize the library.

Preconditions:

- `properties` must not be null. (high-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently. It is not defined in serial installations of the library. This function **does not** duplicate the communicator. It simply copies it to the properties. When these properties are used in a *`saf_open_database`* call, the communicator will at that time be duplicated. So, don't free the MPI communicator between the time this property is set in a given `SAF__DbProps` structure and the time that `SAF__DbProps` structure is used in a *`saf_open_database`* call.

Issues: If the client is going to override the MPI communicator that would ordinarily be associated with the database handle, we have a minor problem with calls to set other properties whose behavior might require special action for parallel: which communicator should they use?

See Also:

- *`saf_open_database`*: 7.3: *Open a database*
- *Database Properties*: Introduction for current chapter

Create an memory-resident database

`saf_setProps_MemoryResident` is a function defined in `dbprops.c`.

Synopsis:

int **`saf_setProps_MemoryResident`** (`SAF_DbProps *properties`)

Description: Setting this property permits the creation of a memory-resident database. A memory-resident database is convenient for creating objects that you would like to be transient. All objects created in a memory-resident database will be lost when the database is closed.

Preconditions:

- `properties` must not be null. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently, however all tasks must agree whether promise mode is to be used for a particular database when that database is opened.

See Also:

- *Database Properties*: Introduction for current chapter

Specify read-only database access

`saf_setProps_ReadOnly` is a function defined in `dbprops.c`.

Synopsis:

int **`saf_setProps_ReadOnly`** (`SAF_DbProps *properties`)

Formal Arguments:

- `properties`: The database property list which will be modified by this function (See **Properties**).

Description: By default a database is opened for read/write access. This function changes the access property so the database is opened for read-only access.

Opening for read-only when the client is, in fact, only reading the database can potentially have a dramatic impact on parallel performance. The reason is that the lower level data modeling kernel, VBT, can duplicate the metadata tables on all processors and wholly eliminate all MPI communication involved in interacting with the database. At present, VBT does **not** actually do this but will, in the future, be modified to do so.

Preconditions:

- `properties` must not be null. (high-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function can be called independently.

See Also:

- *Database Properties*: Introduction for current chapter

Sets

Sets in **SAF** represent infinite point sets. As discussed in the chapter on collections (see **Collections**), in theory all nodes, edges, faces, volumes, etc. are sets.

However, in **SAF**, set objects (e.g. something created with a call to *saf_declare_set*) are instantiated only to represent infinite point sets that are decomposed into other, more primitive entities. Examples are materials, processor pieces, domains, parts in an assembly, blocks, nodesets, etc.

Members

The null set handle

`SAF__NULL_SET` is a macro defined in `saf.h`.

Synopsis:

SAF__NULL_SET (Db)

Description: This macro evaluates to the set handle for the null set of the database. The null set handle is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See **Constants**).

See Also:

- *Sets*: Introduction for current chapter

The universe set handle

`SAF__UNIVERSE` is a macro defined in `saf.h`.

Synopsis:

SAF__UNIVERSE (Db)

Description: This macro evaluates to the set handle for the universe set of the database (See **Constants**).

See Also:

- *Sets*: Introduction for current chapter

Declare a set

`saf_declare_set` is a function defined in `set.c`.

Synopsis:

```
SAF_Set * saf_declare_set (SAF_ParMode pmode, SAF_Db *db, const char *name, int max_topo_dim,  
                           SAF_SilRole role, SAF_ExtendMode extmode, SAF_Set *set)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database handle in which to create the set.
- `name`: The name of the set being declared.
- `max_topo_dim`: The topological dimension of the set. If the set will contain sets of different topological dimensions then this must be the maximum topological dimension of any set in the subset inclusion lattice rooted below `set`.
- `role`: The role of the set. Possible values are `SAF__SPACE` for a spatial set, `SAF__TIME` for a time-base set, `SAF__PARAM` for a parameter space set, or `SAF__USERD` for a user-defined role.
- `extmode`: Indicates whether or not the base-space represented by the set is extendible. Possible values are `SAF__EXTENDIBLE_TRUE` or `SAF__EXTENDIBLE_FALSE`.
- `set`: [OUT] Optional memory for link to the newly declared set.

Description: Every set has a *maximum topological dimension* indicating how the infinity of points that are the set are organized. Are they organized along some curve (1D), surface (2D), volume (3D), etc.? More formally, the *maximum topological dimension* of a set indicates the maximum rank of local coordinate systems over all neighborhoods of the infinite point set.

Note that a *maximum topological dimension* of 0 does **not** mean that the set contains a single point or no points. It means that the set contains only a finite number of points. That is the set is **not** an infinite point set but a finite one.

Preconditions:

- `pmode` must be valid. (low-cost)
- `DATABASE` must be a valid handle. (low-cost)
- `name` cannot be `NULL`. (low-cost)
- `name` must not begin with a leading '@'. (low-cost)
- `max_topo_dim` must be positive. (low-cost)
- `role` must be `SAF__TIME`, `SAF__SPACE`, or `SAF__PARAM`. (low-cost)
- If `role` is `SAF__TIME` then `max_topo_dim` must be 1. (low-cost)
- `extmode` must be either `SAF__EXTENDIBLE_TRUE` or `SAF__EXTENDIBLE_FALSE`. (low-cost)

Return Value: Returns a pointer to a set link on success; null on failure. The `set` argument is the successful return value, or if `set` is null, a new set link is allocated for the return.

Issues: Eventually roles specific to the creation of algebraic types and cell types will be added.

I think we can eliminate the `role` argument here and instead deduce it from the `SAF__Quantity` associated with the default coordinates for the set. For example, if the default coordinates represent a length quantity, then the `role` must be `SAF__SPACE`. If they represent a time quantity, then the `role` must be `SAF__TIME`.

See Also:

- [Sets](#): Introduction for current chapter

Obtain a set description

`saf_describe_set` is a function defined in `set.c`.

Synopsis:

```
int saf_describe_set (SAF_ParMode pmode, SAF_Set *set, char **name, int *max_topo_dim,
                    SAF_SilRole *role, SAF_ExtendMode *extmode, SAF_TopMode *topmode,
                    int *num_colls, SAF_Cat **cats)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `set`: The set to be described.
- `name`: [OUT] The returned name of the set. Pass `NULL` if you do not want this information returned (see `*Returned Strings*`).
- `max_topo_dim`: [OUT] The topological dimension of the set. A `NULL` pointer can be passed if the caller is not interested in obtaining this information.
- `role`: [OUT] The subset inclusion lattice role of the set. A `NULL` pointer can be passed if the caller is not interested in obtaining this information.
- `extmode`: [OUT] Whether the set is extendible or not. A `NULL` pointer can be passed if the caller is not interested in obtaining this information.
- `topmode`: [OUT] Whether the set is a top-level set in the `SIL` or not
- `num_colls`: [OUT] The number of collections currently defined on the set. A `NULL` pointer can be passed if the caller is not interested in obtaining this information.
- `cats`: [OUT] The list of collection categories of the collections defined on the set. A `NULL` pointer can be passed if the caller is not interested in obtaining this information. `cats` should point to the `NULL` pointer if the client wants the library to allocate space, otherwise `cats` should point to something allocated by the caller. In the latter case, the input value of `num_colls` indicates the number of handles the `cats` argument can hold.

Description: This function returns information about a set.

Preconditions:

- `pmode` must be valid. (low-cost)
- `set` must be a valid set handle. (low-cost)
- `num_colls` must be returned if `cats` is requested. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Sets](#): Introduction for current chapter

Find set by matching criteria

`saf_find_matching_sets` is a function defined in `set.c`.

Synopsis:

```
int saf_find_matching_sets (SAF_ParMode  pmode,  SAF_Db  *db,  const char  *name_grep,
                           SAF_SilRole  srole,  int    tdim,  SAF_ExtendMode  extmode,
                           SAF_TopMode  topmode, int  *num, SAF_Set  **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database in which to search
- `name_grep`: The name of the desired set(s) or a limited regular expression that the set names must match. If this argument begins with a leading “at sign”, ‘@’, character, the remaining characters will be treated as a limited form of a regular expression akin to that supported by ‘ed.’ The constant `SAF__ANY_NAME` can be passed if the client does not want to limit the search by name.
- `srole`: The subset inclusion lattice role of the desired set(s). The `SAF__ANY_SILROLE` constant can be passed if the client is not interested in restricting the search on this criteria.
- `tdim`: The topological dimension of the desired set(s). The `SAF__ANY_TOPODIM` constant can be passed if the client is not interested in restricting the search on this criteria.
- `extmode`: User to specify if the set is extendible or not (whether it can grow or not). Pass `SAF__EXTENDIBLE_TRUE`, `SAF__EXTENDIBLE_FALSE`, or `SAF__EXTENDIBLE_TORF`
- `topmode`: whether the matching sets should be top sets. Pass `SAF__TOP_TRUE`, `SAF__TOP_FALSE`, or `SAF__TOP_TORF`
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function will find sets by searching the **entire** database and matching certain criteria. Because it finds sets by matching criteria, this function **does not** exploit the subset inclusion lattice to improve performance.

If the `name_grep` argument begins with a leading “at sign” character, ‘@’, the remaining characters will be treated as a limited form of a regular expression akin to that supported in ‘ed’. Otherwise, it will be treated as a specific name for a set. If the name does not matter, pass `SAF__ANY_NAME`.

If the library was not compiled with -lgen support library, then if regular expressions are used, the library will behave as though `SAF__ANY_NAME` was specified.

Preconditions:

- `pmode` must be valid. (low-cost)
- `DATABASE` must be a database handle. (low-cost)
- The `srole` must be one of `SAF__TIME`, `SAF__SPACE`, `SAF__PARAM`, or `SAF__ANY_SILROLE`. (low-cost)
- If `srole` is `TIME` then `tdim` must be 1. (low-cost)
- `tdim` must be `SAF__ANY_TOPODIM` or positive. (low-cost)
- `extmode` cannot be arbitrarily non-zero for truth. (low-cost)
- `topmode` cannot be arbitrarily non-zero for truth. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Sets](#): Introduction for current chapter

Find sets by traversing the subset inclusion lattice

`saf_find_sets` is a function defined in `set.c`.

Synopsis:

```
int saf_find_sets (SAF_ParMode pmode, SAF_FindSetMode fmode, SAF_Set *set, SAF_Cat *cat,
                  int *num, SAF_Set **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `fmode`: The find mode. Possible values are `SAF__FSETS_TOP` to find the top-level set in the subset inclusion lattice in which `set` is a member; `SAF__FSETS_BOUNDARY` to find the boundary of set `set`; `SAF__FSETS_SUBS` to find all sets which are immediate subsets of `set` by the specified collection category; `SAF__FSETS_SUPS` to find all sets which are immediate supersets of `set` by the specified collection category; and `SAF__FSETS_LEAVES` to find all leaf sets in the subset inclusion lattice rooted at `set` (a leaf set is a set that is a descendent of `set` by the specified collection category and which has no sets below it).
- `set`: The set in the subset inclusion lattice at which to begin searching.
- `cat`: The collection category upon which to search for subsets, supersets, or leaf sets.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: There are two ways to search for sets. One is to simply search the whole database looking for sets that match a particular search criteria such as a name, base dimension, etc. which is handled by `saf_find_matching_sets`. The other is to search for sets by traversing the subset inclusion lattice which is handled by this function. This latter approach is typically faster as it involves only a portion of all sets in the database.

The possible modes to the find call are described below.

```
1 FMODE==SAF_FSETS_TOP
```

This mode of the find will find the top-most ancestor of a given set.

```
1 FMODE==SAF_FSETS_BOUNDARY
```

This mode of the find will find the boundary set of a given set. Note, currently this mode will return a boundary only if one exists in the file. It will **not** attempt to compute a boundary.

```
1 FMODE==SAF_FSETS_SUBS
```

This mode will find all sets that are immediate subsets of the given set by the specified collection category, if any is specified. If the specified collection category is `SAF__ANY_CAT`, then all immediate subsets will be returned, regardless of category.

1

FMODE=SAF_FSETS_SUPS

This mode will find all sets that are immediate supersets of the given set by the specified collection category, if any is specified. If the specified collection category is `SAF__ANY_CAT`, then all immediate supersets will be returned, regardless of category. Note, in typical cases, there is often only one superset of a given set by a given collection category.

1

FMODE=SAF_FSETS_LEAVES

This mode finds all leaf sets in the subset inclusion lattice rooted at `set` (a leaf set is a set that is a descendent of `set` by the specified collection category and which has no sets below it. `SAF__ANY_CAT` is allowed to be the specified collection category.

Preconditions:

- `pmode` must be valid. (low-cost)
- `set` must be a valid set handle. (low-cost)
- `fmode` must be `SAF__FSETS..._TOP, _SUBS, _SUPS, _LEAVES` or `_BOUNDARY`. (low-cost)
- `cat` arg applicable only for `SAF__FSETS_SUPS`, `SAF__FSETS_SUBS` and `SAF__FSETS_LEAVES` modes. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)
- `num` is required in a top-level `SAF__FSETS_LEAVES` mode call. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: If `fmode` is `SAF__FSETS_TOP`, the memory allocation rules for [SAF](#) become irrelevant. If we confine ourselves to topological information (e.g. statements about base-space sets without respect to fields, particularly coordinate fields), then there is only ever one top (maximal ancestor) for any set. Of course, if we have two totally independent objects, say a hammer and a wall, such that the hammer has interpenetrated the wall, then the set that represents the intersection between the hammer and the wall is a subset of both. However, this intersection is a result of the fact that the hammer's coordinate field places it inside the wall. In the purely topological setting, the sets that represent the hammer and the wall are each top sets. Thus, in theory the set that represents their intersection has two maximal ancestors. We do **not** worry about this case here. Thus, a query for `SAF__FSETS_TOP` is always a single set and so either the client asks for the single set handle to be allocated, or it does not.

If `fmode` is `SAF__FSETS_BOUNDARY`, some similar arguments apply. There is only ever one boundary of another set. Thus, if the client queries for the boundary, it is assumed the client has either allocated a single set handle or the library will and simply fill it in.

For the `SAF__FSETS_TOP` and `SAF__FSETS_BOUNDARY` cases, `cat` must be `SAF__NOT_APPLICABLE_CAT`.

This function looks for Relations only in the same scope that stores `set` and thus cannot traverse a subset inclusion lattice that extends outside that scope. [rpm 2004-06-21]

For a `SAF__FSETS_LEAVES` search with a null collection category (`SAF__ANY_CAT`) this function will return a list of unique sets by pruning out the duplicates. However, the pruning occurs down at the leaves and not in the internal nodes of the graph, and therefore we may end up traversing portions of the graph repeatedly. [rpm 2004-06-21]

See Also:

- [saf_find_matching_sets](#): 9.5: *Find set by matching criteria*
- [Sets](#): Introduction for current chapter

Get an attribute from a set

`saf_get_set_att` is a function defined in `set.c`.

Synopsis:

```
int saf_get_set_att (SAF_ParMode pmode, SAF_Set *set, const char *key, hid_t *type, int *count,
                    void **value)
```

Description: This function is identical to the generic [saf_get_attribute](#) function except that it is specific to `SAF__Set` objects to provide the client with compile time type checking. For a description, see [saf_get_attribute](#).

See Also:

- [saf_get_attribute](#): 23.1: *Read a non-sharable attribute*
- [Sets](#): Introduction for current chapter

Put an attribute to a set

`saf_put_set_att` is a function defined in `set.c`.

Synopsis:

```
int saf_put_set_att (SAF_ParMode pmode, SAF_Set *set, const char *key, hid_t type, int count, const
                    void *value)
```

Description: This function is identical to the generic [saf_put_attribute](#) function except that it is specific to `SAF__Set` objects to provide the client with compile time type checking. For a description, see [saf_put_attribute](#).

See Also:

- [saf_put_attribute](#): 23.2: *Create or update a non-sharable attribute*
- [Sets](#): Introduction for current chapter

Collection Categories

Collection categories are used to categorize collections of sets or cells. Each collection on a set is one of a particular category. There is only ever one collection of a particular category on a set. Typically, collection categories are used to categorize, for example collections of nodes, elements, processors, blocks, domains, etc. However, collection categories may be used however the client wishes to categorize different collections of sets or cells.

Members

The self decomposition of a set

`SAF_SELF` is a macro defined in `saf.h`.

Synopsis:

```
SAF_SELF (Db)
```

Description: This macro evaluates to the collection category handle for the self decomposition of a set (See **Constants**).

See Also:

- [Collection Categories](#): Introduction for current chapter

Declare a collection category

`saf_declare_category` is a function defined in `cat.c`.

Synopsis:

```
SAF_Cat * saf_declare_category (SAF_ParMode pmode, SAF_Db *db, const char *name,  
                                SAF_Role *role, int tdim, SAF_Cat *cat)
```

Formal Arguments:

- `db`: The database handle.
- `name`: The collection category name.
- `role`: Role of collections of this category (see **Collection Roles**).
- `tdim`: The maximum topological dimension of the members of collections of this category.
- `cat`: [OUT] The returned collection category handle.

Description: This function declares a collection category.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- The database must not be open for read-only access. (no-cost)
- A `name` must be supplied for the category. (low-cost)
- `role` must be a valid collection role. (low-cost)
- Topological dimension, `tdim`, must be positive. (low-cost)

Return Value: The new category handle is returned on success; NULL on failure. If `cat` is non-null then it will be initialized and used as the return value.

Parallel Notes: This call must be collective across the database communicator.

See Also:

- *Collection Categories*: Introduction for current chapter

Get a description of a collection category

`saf_describe_category` is a function defined in `cat.c`.

Synopsis:

```
int saf_describe_category (SAF_ParMode pmode, SAF_Cat *cat, char **name, SAF_Role *role,  
                           int *tdim)
```

Formal Arguments:

- `cat`: A collection category handle.
- `name`: If non-NULL, the returned name of the collection category (see **Returned Strings**).
- `role`: If non-NULL, the returned role of the collection category (see **Collection Roles**).
- `tdim`: If non-NULL, the returned maximum topological dimension of members of collections of this category.

Description: This call describes a collection category.

Preconditions:

- `pmode` must be valid. (low-cost)
- `cat` must be a valid category handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This call must be collective across the database communicator in which the category is defined.

See Also:

- *Collection Categories*: Introduction for current chapter

Find collection categories

`saf_find_categories` is a function defined in `cat.c`.

Synopsis:

```
int saf_find_categories (SAF_ParMode pmode, SAF_Db *db, SAF_Set *containing_set, const
                        char *name, SAF_Role *role, int tdim, int *num, SAF_Cat **found)
```

Formal Arguments:

- `db`: Database on which to restrict the search.
- `containing_set`: The set upon which to restrict the search. The special macro `“SAF__UNIVERSE“(db)` (which takes a database handle as an argument) allows the search to span all categories of the specified database.
- `name`: The name of the categories upon which to restrict the search. The constant `SAF__ANY_NAME` allows the search to span categories with any name.
- `role`: The role of the categories upon which to restrict the search. A null pointer allows the search to span categories with any role (see **Collection Roles**).
- `tdim`: The topological dimension of the categories upon which to restrict the search. The constant `SAF__ANY_TOPODIM` allows the search to span categories with any topological dimension.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function will find collection categories matching the specified `name`, `role` and `tdim`. It searches collection categories defined on the `containing_set`, which can be set to `“SAF__UNIVERSE“(db)`, implying that all collection categories in the **entire** database should be searched. Since the number of collection categories is relatively small, **and global**, such a search should not take much time.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)
- `role` must be a valid role handle or `NULL`. (low-cost)
- `containing_set` must be a valid set handle or `NULL`. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: This function must be called collectively across the database communicator of the `containing_set`.

See Also:

- *Collection Categories*: Introduction for current chapter

Get an attribute with a cat

`saf_get_cat_att` is a function defined in `cat.c`.

Synopsis:

```
int saf_get_cat_att (SAF_ParMode pmode, SAF_Cat *cat, const char *name, hid_t *datatype,
                    int *count, void **value)
```

Formal Arguments:

- `cat`: Collection category owning the attribute for which we're searching.
- `name`: Name of the attribute.
- `datatype`: [OUT] Datatype of the attribute as it is stored.
- `count`: [OUT] Number of elements contained in the attribute.
- `value`: [OUT] On successful return this will point to an allocated array containing `count` elements each of type `datatype`.

Description: This function is identical to the generic *saf_get_attribute* function except that it is specific to `SAF__Cat` objects to provide the client with compile time type checking. For a description, see *saf_get_attribute*.

See Also:

- *saf_get_attribute*: 23.1: Read a non-sharable attribute
- *Collection Categories*: Introduction for current chapter

Put an attribute with a cat

`saf_put_cat_att` is a function defined in `cat.c`.

Synopsis:

```
int saf_put_cat_att (SAF_ParMode pmode, SAF_Cat *cat, const char *name, hid_t datatype, int count,
                    const void *value)
```

Formal Arguments:

- `pmode`: Parallel mode for adding the new attribute.
- `cat`: Collection category for which the new attribute is added.
- `name`: The name of the attribute.
- `datatype`: The datatype of each element of the `value` for the attribute.
- `count`: The number of elements pointed to by `value`, each of type `datatype`.
- `value`: The array of `count` elements each of type `datatype` to use for the attribute's value.

Description: This function is identical to the generic *saf_put_attribute* function except that it is specific to `SAF__Cat` objects to provide the client with compile time type checking. For a description, see *saf_put_attribute*.

See Also:

- *saf_put_attribute*: 23.2: Create or update a non-sharable attribute
- *Collection Categories*: Introduction for current chapter

Collections

In theory, all nodes, edges, faces, elements, blocks, materials, processor pieces etc. are just sets. See *saf_declare_set* for more of a description of sets. In practice, we have a need to distinguish between two kinds of sets: *primitive* ones such as the elements of a mesh, and *aggregate* ones, such as material or block sets. We call these two classes of sets, *Cells* and *Sets* respectively. Cells are primitive sets such as the nodes or elements of a mesh. What makes them primitive? They are not decomposed into any other sets, whose union can form them. In lattice theory terms, cells are *Join Irreducible Members* (or *JIMS*) of the subset inclusion lattice.

Sets are aggregate sets such as a processor piece or a block. In lattice theory terms, sets are *Join Reducible Members* (or *JRMS* pronounced “germs”). The key point here is that cells are **never** instantiated as first class sets in this API (e.g. using the *saf_declare_set* call). Instead cells only ever exist as members of collections.

On the other hand, collections themselves may be composed of either cells or sets. When a collection is declared, the client either specifies a cell-type for the members, implying the collection is composed of cells, or not, implying the collection is composed of sets.

Since collections are defined by their containing set and a collection category, this pair serves to define a collection and there is no specific `SAF___Xxxx` handle explicitly for collections.

Members

Declare a collection

`saf_declare_collection` is a function defined in `coll.c`.

Synopsis:

```
int saf_declare_collection (SAF_ParMode  pmode,  SAF_Set  *containing_set,  SAF_Cat  *cat,
                           SAF_CellType  ctype,  int  count,  SAF_IndexSpec  ispec,
                           SAF-DecompMode is_decomp)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `containing_set`: The containing set of the collection. In `SAF___ONE` parallel mode, all processes except the process identified by the rank argument of the `SAF___ONE` macro are free to pass `SAF___NULL_SET` with the set’s database handle.
- `cat`: The collection category.
- `ctype`: The cell type of the members of the collection. If this is a non-primitive collection, pass `SAF___CELLTYPE_SET`. If this is a primitive collection of mixed cell type, pass `SAF___CELLTYPE_MIXED`. If this is a primitive collection of arbitrarily connected cells, pass `SAF___CELLTYPE_ARB`. Otherwise, it must be a primitive collection of homogeneous type and the caller should pass one of the cell types specified by `SAF___CellType`.
- `count`: The number of members of the collection. If the containing set is an extendible set, the count can be changed by a call to *saf_extend_collection*.
- `ispec`: The indexing scheme of the collection (e.g., how are members of the collection identified within the collection). We have predefined some macros for common cases: `SAF___1DC`, `SAF___2DC`, and `SAF___3DC` for C-ordered and indexed arrays and likewise for Fortran-ordered and indexed arrays (replace the “C” with an “F” in the macro name).
- `is_decomp`: Indicates if the specified collection is a decomposition of its containing set. That is, if we take the union of all the members of the collection do we form a set that is equal to the containing set?

Description: Collections are contained in sets. Thus there is no explicit object handle for a collection. Instead, a collection is referenced by a pair: the containing set handle and the category handle.

If the set is extendible, then any collection declared on it is considered extendible.

Preconditions:

- `pmode` must be valid. (low-cost)
- `containing_set` must be a valid set handle for participating processes. (low-cost)
- `cat` must be a valid category handle for participating processes. (low-cost)
- `is_decomp` must be either `SAF__DECOMP_TRUE` or `SAF__DECOMP_FALSE` for participating processes. (low-cost)
- `ispec` rank and sizes must be valid for participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: As currently implemented, the `count` and `ispec` args are redundant. However, in general, the indexing schema used to identify members of the collection is nearly totally independent of the count. A common example **SAF does not yet** support is the case in which the indexing ids for the members of the collection is some other arbitrary list of ints (or character string names, etc). For example, all of the nodes on the top set is **not** necessarily indexed `0..“num_nodes“-1`. Under these conditions, the indexing scheme is another, problem sized array of ints. However, to handle this, we probably need a `saf_write_collection_indexing` function to actually write that data to a file. Writing it in this call would violate our current policy where problem-sized disk I/O occurs only on calls with “write” or “read” in their names.

See Also:

- [*saf_extend_collection*](#): 11.3: *Add members to a collection*
- [*Collections*](#): Introduction for current chapter

Describe a collection

`saf_describe_collection` is a function defined in `coll.c`.

Synopsis:

```
int saf_describe_collection (SAF_ParMode pmode, SAF_Set *containing_set, SAF_Cat *cat,
                           SAF_CellType *t, int *count, SAF_IndexSpec *ispec,
                           SAF-DecompMode *is_decomp, SAF_Set **member_sets)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `containing_set`: The containing set of the desired collection. In `SAF__ONE` parallel mode, all processes except the process identified by the rank argument of the `SAF__ONE` macro are free to pass `SAF__NULL` with the set's database handle.
- `cat`: The collection category of the desired collection.
- `t`: [OUT] The cell-type of the members of the collection. Pass `NULL` if this return value is not desired.
- `count`: [OUT] The returned count of the collection. Pass `NULL` if this return value is not desired.
- `ispec`: [OUT] The returned indexing specification for the collection. Pass `NULL` if this return value is not desired.

- `is_decomp`: [OUT] Whether the collection is a decomposition of the containing set. Pass `NULL` if this return value is not desired.
- `member_sets`: If the collection is non-primitive, this argument is used to return the specific set handles for the sets that are in the collection. Pass `NULL` if this return value is not desired. Otherwise, if `member_sets` points to `NULL`, the library will allocate space for the returned set handles. Otherwise the caller allocates the space and the input value of `count` indicates the size of the space in number of set handles.

Description: Returns information about a collection.

Preconditions:

- `pmode` must be valid. (low-cost)
- The `containing_set` must be valid for all participating processes. (low-cost)
- `cat` must be a valid category handle for participating processes. (low-cost)
- `NUM_SETS` and `member_sets` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: having both arguments `NULL` will cause `_saf_valid_memhints` to return false, but in this particular case, having both arguments `NULL` is ok, so we don't call `_saf_valid_memhints` if both are 0.

See Also:

- [Collections](#): Introduction for current chapter

Add members to a collection

`saf_extend_collection` is a function defined in `coll.c`.

Synopsis:

```
int saf_extend_collection (SAF_ParMode pmode, SAF_Set *containing_set, SAF_Cat *cat,
                           int add_count, SAF_IndexSpec add_ispec)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `containing_set`: The containing set of the collection.
- `cat`: The collection category of the collection.
- `add_count`: The number of members to add to the collection.
- `add_ispec`: The new indexing scheme.

Description: This function allows the client to add members to an existing collection. While you can extend a collection, you cannot change the number of dimensions in the indexing scheme. You can only change the size in each dimension and then you can only increase it. That is, if the collection was indexed using 2 dimensional indexing, it cannot be changed to 3 dimensional indexing.

Preconditions:

- `pmode` must be valid. (low-cost)
- `containing_set` must be a valid set handle for participating processes. (low-cost)
- `cat` must be a valid category handle for participating processes. (low-cost)
- `add_ispec` sizes must be valid, total `add_count` and be compatible with the existing indexing. (med-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Collections*: Introduction for current chapter

Find collections

`saf_find_collections` is a function defined in `coll.c`.

Synopsis:

```
int saf_find_collections (SAF_ParMode pmode, SAF_Set *containing_set, SAF_Role *role,  
                          SAF_CellType cell_type, int topo_dim, SAF-DecompMode decomp_mode,  
                          int *num, SAF_Cat **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `containing_set`: The containing set in which to search for collections. In `SAF__ONE` parallel mode, all processes except the process identified by the rank argument of the `SAF__ONE` macro are free to pass `SAF__NULL_SET` with the set's database handle.
- `role`: The role of the collection. Pass `NULL` if you do not wish to limit the search by this parameter.
- `cell_type`: The cell-type of the members of the collection. Pass `SAF__ANY_CELLTYPE` if you do not wish to limit the search by this parameter.
- `topo_dim`: The topological dimension of the collection. Pass `SAF__ANY_TOPODIM` if you do not wish to limit the search by this parameter.
- `decomp_mode`: Whether the found collections must be a decomposition of the containing set. Pass `SAF__DECOMP_TORF` if it does not matter.
- `num`: For this and the succeeding argument, (see **Returned Handles**).
- `found`: For this and the preceding argument, (see **Returned Handles**).

Description: This function is used to search for collections on a given set. In addition, the client can limit the search to collections of a given role, with a given cell-type or those which are or are not a decomposition of the containing set.

Preconditions:

- `pmode` must be valid. (low-cost)
- `containing_set` must be a valid set handle for participating processes. (low-cost)
- `NUM_COLLs` and `CATS` must be compatible for return value allocation. (low-cost)
- `role` must be a valid role handle if supplied. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: The documentation for this function originally said it would return all collections in the whole database if the `containing_set` arg passed was `“SAF__UNIVERSE“`(db). However, it is clear from the implementation that it cannot do that.

See Also:

- *Collections*: Introduction for current chapter

Compare two collections

`saf_same_collections` is a function defined in `coll.c`.

Synopsis:

`hbool_t saf_same_collections (SAF_Set *Sa, SAF_Cat *Ca, SAF_Set *Sb, SAF_Cat *Cb)`

Formal Arguments:

- `Sa`: The set component of the first or left operand of the equality comparison operator.
- `Ca`: The category component of the first or left operand of the equality comparison operator.
- `Sb`: The set component of the second or right operand of the equality comparison operator.
- `Cb`: The category component of the second or right operand of the equality comparison operator.

Description: Compare two given collections for equality. Note that each collection is specified as a set-category pair. These pairs are equal if they have “the same” sets and “the same” categories.

Parallel Notes: Independent

See Also:

- [Collections](#): Introduction for current chapter

Subset Relations

Subset relations are used to define a relationship between two sets in which one set, the intended *subset*, is the subset of the other set, the intended *superset*. In order to define a subset relation both sets require a common, decomposing collection. That is, there must exist a collection of the same category on both sets and that collection must be a decomposition of its containing set. For more information, see **Relation Notes** and [saf_declare_subset_relation](#).

Members

Conveniently specify a boundary subset

`SAF_BOUNDARY` is a macro defined in `saf.h`.

Synopsis:

`SAF_BOUNDARY (P, B)`

Description: This macro provides a convenient way to specify four of the args, *sup_cat*, *sub_cat*, *sbmode*, and *cbmode* of the [saf_declare_subset_relation](#) call. Use it when the subset is the boundary of the superset. The arguments `P` and `B` represent collection categories of collections on superset and subset, respectively. See [saf_declare_subset_relation](#) for a more detailed description.

See Also:

- [saf_declare_subset_relation](#): 12.5: *Declare a subset relation*
- [Subset Relations](#): Introduction for current chapter

Conveniently specify a typical subset

`SAF_COMMON` is a macro defined in `saf.h`.

Synopsis:

SAF_COMMON (C)

Description: This macro provides a convenient way to specify four of the args, *sup_cat* , *sub_cat* , *sbmode*, and *cbmode* of the *saf_declare_subset_relation* call. Use it when you have a typical subset. The argument C is meant to be a collection category in common to both sup and sub sets. See *saf_declare_subset_relation* for a more detailed description.

See Also:

- *saf_declare_subset_relation*: 12.5: *Declare a subset relation*
- *Subset Relations*: Introduction for current chapter

Conveniently specify an embedded boundary subset

SAF_EMBEDBND is a macro defined in saf.h.

Synopsis:**SAF_EMBEDBND (P, B)**

Description: This macro provides a convenient way to specify four of the args, *supcat* , *sub_cat* , *sbmode*, and *cbmode* of the *saf_declare_subset_relation* call. Use it when the subset is some internal boundary in the superset but is NOT the boundary of the superset. The arguments P and B represent collection categories of collections on superset and subset, respectively.

See Also:

- *saf_declare_subset_relation*: 12.5: *Declare a subset relation*
- *Subset Relations*: Introduction for current chapter

Conveniently specify an general subset

SAF_GENERAL is a macro defined in saf.h.

Synopsis:**SAF_GENERAL (BND)**

Description: This macro provides a convenient way to specify four of the args, *sup_cat* , *sub_cat* , *sbmode*, and *cbmode* of the *saf_declare_subset_relation* call. Use it when all that is known is that the subset is indeed a subset of the superset, but the details of their relationship are unknown (e.g. no data). The argument BND is a boolean meant to indicate if the subset is the boundary of the superset.

See Also:

- *saf_declare_subset_relation*: 12.5: *Declare a subset relation*
- *Subset Relations*: Introduction for current chapter

Declare a subset relation

saf_declare_subset_relation is a function defined in rel.c.

Synopsis:


```
SAF_Rel * saf_declare_subset_relation(SAF_ParMode pmode, SAF_Db *db, SAF_Set *sup,
                                     SAF_Set *sub, SAF_Cat *sup_cat, SAF_Cat *sub_cat,
                                     SAF_BoundMode sbmode, SAF_BoundMode cbmode,
                                     SAF_RelRep *srtype, hid_t A_type, void *A_buf,
                                     hid_t B_type, void *B_buf, SAF_Rel *rel)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database in which to place the new relation.
- `sup`: The superset. In `SAF__ONE` parallel mode, all processors except the one identified by the `SAF__ONE` argument should pass the null set of the database by using the `SAF__NULL` macro.
- `sub`: The subset. In `SAF__ONE` parallel mode, all processors except the one identified by the `SAF__ONE` argument should pass the null set of the database by using the `SAF__NULL` macro.
- `sup_cat`: The collection category on the `sup` set upon which the subset relation is being defined. Note that collections of this category must have already been defined on `sup`. Otherwise, an error is generated. Note, the four args, `sup_cat`, `sub_cat`, `sbmode`, `cbmode`, are typically passed using one of the macros described above, `SAF__COMMON` `(C)`, ```SAF__BOUNDARY` `(P,“B“)`, `SAF__EMBEDBND` `(P,“B“)` or `SAF__GENERAL` `(BND)`
- `sub_cat`: The collection category on the `sub` set upon which the subset relation is being defined. Note that collections of this category must have already been defined on `sub`. Otherwise an error is generated.
- `sbmode`: Indicates whether `sub` is the boundary of `sup`. Pass either `SAF__BOUNDARY_TRUE` or `SAF__BOUNDARY_FALSE`
- `cbmode`: Indicates whether **members** of collection on `sub` are **on** the boundary of members of the collection on `sup`. Pass either `SAF__BOUNDARY_TRUE` or `SAF__BOUNDARY_FALSE`
- `srtype`: Subset relation types. This argument describes how the data in `ABUF` represents the subset. Valid values are `SAF__HSLAB` meaning that `ABUF` points to a hyperslab specification and `SAF__TUPLES` meaning that `ABUF` points to a list of N-tuples.
- `A_type`: The type of the data in `A_buf`
- `A_buf`: This buffer contains references, one for each member of the domain collection (on `sub`), to members of the range collection (on `sup`). The client may pass `NULL` here meaning that the raw data will be bound to the object during write, rather than declaration.
- `B_type`: The type of the data in `B_buf`
- `B_buf`: This buffer is valid **only** when the members of the domain collection (on `sub`) are on the boundaries of the members of the range collection (on `sup`). In this case, the data contained in this buffer identifies “which piece” of the boundary each member of the domain collection is. Otherwise, the client should pass `NULL` here. As with `ABUF`, the client may pass also `NULL` here meaning the raw data will be bound to the object during write, rather than declaration.
- `rel`: [OUT] Optional returned relation handle.

Description: This call is used to declare a subset relation between two sets. The relation is specified in terms of collections on both sets. The subset, `sub`, can be either a boundary of `sup` or not. Which case is indicated by the `sbmode` argument which can be either `SAF__BOUNDARY_TRUE` or `SAF__BOUNDARY_FALSE`.

In addition, The **members of the collection** on the `sub` set are either **on the boundary of** the members of the collection on the `sup` set or not (the only other acceptable case is one in which the members of the collection on the `sub` are **equal to** the members of the collection on `sup`). Which case is indicated by the value of the `cbmode` argument, can be either `SAF__BOUNDARY_TRUE` or `SAF__BOUNDARY_FALSE`.

Thus, there are two statements made about boundary information. One about the sets, `sup` and `sub` and one about the members of the collections on `sup` and `sub`. Furthermore, the statement about the sets, indicated by `sbmode`, is that `sub` is **the** boundary of `sup` or it is not. The statement about the collections, indicated by `cbmode`, is that the members of the `sub` collection are **on the boundary of** the members of the `sup` collection or not.

The values in `ABUF` enumerate the members of the collection on `sub` that are either on the boundaries of or equal to the members of the collection on `sup`. In the **on the boundary of** case (e.g. `cbmode` `==` `SAF__BOUNDARY_TRUE`) the values in `BBUF`, if non-NULL, indicate “which” piece of the `sup` collection member’s boundary each member of `sub` collection is. For example, if the `sub` collection is faces and the `sup` collection is a bunch of hexes, `BBUF` can be used to identify which of the 6 faces each member of `sub` collection is. This information is optional.

The group of four formal arguments `sup_cat`, `sub_cat`, `sbmode`, `cbmode` select from the various cases described above. For convenience, we provide a number of macros for these four arguments for the common cases...

:ref: `SAF_COMMON <SAF_COMMON>('C)` : a subset relationship in which the subset is specified by enumerating those members of the superset that are **in** the subset. This is the most common case. Argument `C` is the collection category both sets have in common.

`SAF_BOUNDARY <SAF_BOUNDARY>('P', 'B')` : a subset relationship in which `sub` is **the** boundary of `sup` and the members of `B` on `sub` are on the boundaries of the members of `P` on `sup`.

`SAF_EMBEDBND <SAF_EMBEDBND>('P', 'B')` : a subset relationship in which `sub` is some embedded boundary in `sup` and members of the collection `B` on `sub` are on the boundaries of the members of collection `P` on `sup`.

`SAF_GENERAL <SAF_GENERAL>('BND')` : a subset relationship in which all that is known is that `sub` is indeed a subset of `sup`. The details of the relationships are not known. In this case, the `BND` is a boolean indicating if `sub` is **the** boundary of `sup`.

Finally, there is the subset relation representation type, `srtype`...

By and large, the details of the relation data can be derived from knowledge of the indexing schemes used in the domain and range collections of the relation. For example, if the range is indexed using some `N` dimensional indexing scheme, then the relation will either be an `N` dimensional hyperslab or a list of `N`-tuples.

In the case of `SAF__HSLAB`, it is assumed the memory pointed to by `ABUF` contains 3 `N`-tuples of the form (starts, counts, strides) where starts, counts, and strides indicate the starting point of the hyperslab in `N` dimensions, the number of items in each dimension and the stride (through the range collection) in each dimension respectively. The order of dimensional axes in each of these arrays is assumed to match the terms in which the range collection’s indexing is specified.

In the case of `SAF__TUPLES`, it is assumed the memory pointed to by `ABUF` contains a list of `N`-tuples where each `N` tuple identifies one member of the range collection. The offsets argument to `SAF__TUPLES`, if present, indicates a fixed `N`-tuple offset to be associated with each `N`-tuple in `ABUF`.

There are two ways the client may pass the data buffer holding the relation data; either here as the `ABUF` argument of the declare call or later as the `ABUF` argument of the write call. The client cannot do both. It must choose. This flexibility was provided to aim the API for in-memory communications as well as persistent file writes. By and large, the client should pass `NULL` for the `ABUF` arg here and pass the buffer in the write call whenever it is writing persistent data to the file. However, whenever the client is planning to do in-memory communication, it should specify `ABUF` here.

Preconditions:

- `pmode` must be valid. (low-cost)
- `sup` must be a valid set handle. (low-cost)
- `sub` must be a valid set handle. (low-cost)
- `sbmode` must be either `SAF__BOUNDARY_TRUE` or `SAF__BOUNDARY_FALSE`. (low-cost)
- `cbmode` must be either `SAF__BOUNDARY_TRUE` or `SAF__BOUNDARY_FALSE`. (low-cost)

- `cbmode` must be `SAF__BOUNDARY_TRUE` if `sbmode` is `SAF__BOUNDARY_TRUE` for all participating processes. (low-cost)
- Either `A_buf` is null and both `sup_cat` and `sub_cat` are not valid cat handles or. (low-cost)
- `B_buf` can be non-NULL only when `cbmode` is `SAF__BOUNDARY_TRUE`. (low-cost)
- On the reserved, “self” collection, `cbmode` and `sbmode` must be `SAF__BOUNDARY_FALSE`. (low-cost)
- `srtype` must be a valid relation representation handle. (low-cost)
- `srtype` must be either `SAF__HSLAB` or `SAF__TUPLES`. (low-cost)
- `rel` must be non-NULL. (low-cost)
- `A_type` must be an integer type if supplied. (low-cost)
- `B_type` must be an integer type if supplied. (low-cost)

Return Value: On success returns either the supplied `rel` argument or a pointer to a newly allocated relation link. Returns the null pointer on failure.

Issues: We may want to have separate datatypes for `ABUF` and `BBUF` as `BBUF`’s values are likely to always fit in a byte though I don’t know any clients that actually store them that way.

At present, we assert that for any boundary case, the range collection (on `sup`) must be of a primitive type (e.g. not `SAF__CELLTYPE_SET`). However, this **really** need not be the case. Any set which has a boundary set can then have “pieces” of that boundary that could be referred to use the local address space of that boundary. It just so happens that the most common case for this is when we are referring to cell-types.

If we could guarantee all processors’ `is_top` member were identical, we could wrap this call so that we don’t try to put the set record if its already NOT a top set.

See Also:

- [Subset Relations](#): Introduction for current chapter

Get a description of a subset relation

`saf_describe_subset_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_describe_subset_relation (SAF_ParMode pmode, SAF_Rel *rel, SAF_Set *sup,
                                SAF_Set *sub, SAF_Cat *sup_cat, SAF_Cat *sub_cat,
                                SAF_BoundMode *sbmode, SAF_BoundMode *cbmode,
                                SAF_RelRep *srtype, hid_t *data_type)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation handle.
- `sup`: [OUT] The superset. Pass NULL if you do not want this value returned.
- `sub`: [OUT] The subset. Pass NULL if you do not want this value returned.
- `sup_cat`: [OUT] The collection category on the `sup` set upon which the subset relation is defined. Note that collections of this category must have already been defined on `sup`. Otherwise, an error is generated. Note that the four args `sup_cat`, `sub_cat`, `sbmode`, and `cbmode` are typically passed using one of the macros described in the [saf_declare_subset_relation](#) call, `SAF__COMMON`, `SAF__BOUNDARY`, `SAF__EMBEDBND` or `SAF__GENERAL`. Pass NULL if you do not want this value returned.

- `sub_cat`: [OUT] The collection category on the sub set upon which the subset relation is defined. Again, pass NULL if you do not want this value returned.
- `sbmode`: [OUT] Indicates whether `sub` is the boundary of `sup`. A value of `SAF__BOUNDARY_TRUE`, indicates that the `sub` is a boundary of `sup`. A value of `SAF__BOUNDARY_FALSE` indicates `sub` is **not** a boundary of `sup`. Pass NULL if you do not want this value returned.
- `cbmode`: [OUT] Indicates whether **members** of collection on `sub` are **on** the boundaries of members of the collection on `sup`. A value of `SAF__BOUNDARY_TRUE` indicates they are. A value of `SAF__BOUNDARY_FALSE` indicates they are not. Pass NULL if you do not want this value returned.
- `srtype`: [OUT] The representation specification. Pass NULL if you do not want this handle returned. See [saf_declare_subset_relation](#) for the meaning of values of this argument.
- `data_type`: [OUT] The data-type of the data stored with the relation. Pass NULL if you do not want this value returned.

Description: Returns information about a subset relation.

Preconditions:

- `pmode` must be valid. (low-cost)
- The `rel` argument must be a valid handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [saf_declare_subset_relation](#): 12.5: *Declare a subset relation*
- [Subset Relations](#): Introduction for current chapter

Find subset relations

`saf_find_subset_relations` is a function defined in `rel.c`.

Synopsis:

```
int saf_find_subset_relations (SAF_ParMode pmode, SAF_Db *db, SAF_Set *sup, SAF_Set *sub,
                             SAF_Cat *sup_cat, SAF_Cat *sub_cat, SAF_BoundMode sbmode,
                             SAF_BoundMode cbmode, int *num, SAF_Rel **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: Database in which to limit the search.
- `sup`: The superset to limit search to.
- `sub`: The subset to limit search to.
- `sup_cat`: The collection category on the superset to limit search to. Pass `SAF__ANY_CAT` if you do not want to limit the search to any particular category.
- `sub_cat`: The collection category on the subset to limit search to. Pass `SAF__ANY_CAT` if you do not want to limit the search to any particular category.
- `sbmode`: If `SAF__BOUNDARY_TRUE`, limit search to relations in which the subset is the boundary of the superset.
- `cbmode`: If `SAF__BOUNDARY_TRUE`, limit search to relations in which the members of the subset are on the boundaries of the members of the superset.

- `num`: For this and the succeeding argument, (see **Returned Handles**).
- `found`: For this and the preceding argument, (see **Returned Handles**).

Description: This function finds any subset relations that might exist between two sets or a subset relation on a specific collection category.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `sup_cat` must either be a valid category handle or `SAF___ANY_CAT`. (low-cost)
- `sub_cat` must either be a valid category handle or `SAF___ANY_CAT`. (low-cost)
- `sub` must be a valid set handle. (low-cost)
- `sup` must be a valid set handle. (low-cost)
- `num` and `found` must be compatible for the return value allocation. (low-cost)

Return Value: The constant `SAF___SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Subset Relations*: Introduction for current chapter

Get datatype and size for a subset relation

`saf_get_count_and_type_for_subset_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_get_count_and_type_for_subset_relation (SAF_ParMode pmode, SAF_Rel *rel,
                                              SAF_RelTarget *target, size_t *abuf_sz,
                                              hid_t *abuf_type, size_t *bbuf_sz,
                                              hid_t *bbuf_type)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation handle.
- `target`: Optional relation targeting information.
- `abuf_sz`: [OUT] The number of items that would be placed in the A-buffer by a call to the *saf_read_subset_relation* function. The caller may pass value of `NULL` for this parameter if this value is not desired.
- `abuf_type`: [OUT] The type of the items that would be placed in the A-buffer by a call to the *saf_read_subset_relation* function. The caller may pass value of `NULL` for this parameter if this value is not desired.
- `bbuf_sz`: [OUT] The number of items that would be placed in the B-buffer by a call to the *saf_read_subset_relation* function. The caller may pass value of `NULL` for this parameter if this value is not desired.
- `bbuf_type`: [OUT] The type of the items that would be placed in the B-buffer by a call to the *saf_read_subset_relation* function. The caller may pass value of `NULL` for this parameter if this value is not desired.

Description: This function is used to retrieve the number and type of A-buffer and B-buffer data items that would be retrieved by a call to the [saf_read_subset_relation](#) function. This function may be used by the caller to determine the sizes of the buffers needed when pre-allocation is desired or to determine how to traverse the buffer(s) returned by the [saf_read_subset_relation](#) function.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle for all participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: Relation targeting is not yet implemented.

See Also:

- [saf_read_subset_relation](#): 12.9: *Read the data for a subset relation*
- [Subset Relations](#): Introduction for current chapter

Read the data for a subset relation

`saf_read_subset_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_read_subset_relation (SAF_ParMode pmode, SAF_Rel *rel, SAF_RelTarget *target,
                             void **abuf, void **bbuf)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation whose data is to be read.
- `target`: Relation targeting information.
- `abuf`: The data representing those members in the range collection (on the superset) that are related to the members in the domain collection (on the subset).
- `bbuf`: Optional data for boundary subsets indicating which local piece of boundary each member in the domain collection represents in each member of the range collection (see [saf_declare_subset_relation](#))

Description: Read the data associated with a subset relation. Note that there is no information about the buffers passed as formal arguments to this call. Why? Because any information about the “native” buffers is known via the [saf_describe_subset_relation](#) call. The client may “target” the data read in this call for a particular data-type, etc. by using the [saf_target_subset_relation](#) call.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle. (low-cost)
- `abuf` cannot be null for all participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: If the client requests `bbuf` but none was written, is that an error? Unfortunately, the only answer that works in all cases is to declare this an error. This is so because it is not possible to notify the client that none was written except by returning `“bbuf”==“NULL”` and that is *not* possible in the case that the client has pre-allocated `bbuf`

(except if we opt to free the pre-allocated `bbuf`, and then set it to `NULL` which I don't think would be a good idea). We limit returning error to **only** this case. The other case returns `bbuf` `==` `NULL`

See Also:

- [*saf_declare_subset_relation*](#): 12.5: *Declare a subset relation*
- [*saf_describe_subset_relation*](#): 12.6: *Get a description of a subset relation*
- [*saf_target_subset_relation*](#): 12.10: *Set the destination form of a subset relation*
- [*Subset Relations*](#): Introduction for current chapter

Set the destination form of a subset relation

`saf_target_subset_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_target_subset_relation (SAF_RelTarget *target, SAF_RelRep *srtype, hid_t type)
```

Formal Arguments:

- `target`: [OUT] Relation targeting information to be initialize herein.
- `srtype`: Target subset relation types.
- `type`: Target data types.

Description: This function establishes the target (destination) form of subset relation data during either read or write. When used prior to a write call, it establishes the form of data in the file. When used prior to a read call, it establishes the form of data as desired in memory.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: Not implemented yet.

See Also:

- [*Subset Relations*](#): Introduction for current chapter

Reuse data in a subset relation

`saf_use_written_subset_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_use_written_subset_relation (SAF_ParMode pmode, SAF_Rel *rel, SAF_Rel *oldrel,
                                     hid_t A_buf_type, hid_t B_buf_type, SAF_Db *file)
```

Formal Arguments:

- `pmode`: the parallel mode.
- `rel`: The handle for the relation to be updated.
- `oldrel`: The handle for the relation pointing to the data to be re-used.
- `A_buf_type`: The type of data that would be written for the A buffer (see [*saf_write_subset_relation*](#)) if this call was actually doing any writing.
- `B_buf_type`: The type of data that would be written for the B buffer (see [*saf_write_subset_relation*](#)) if this call was actually doing any writing.

- `file`: The file the data would be written to if this call was actually doing any writing.

Description: This call binds data for an existing relation to a new relation. This call can be used in place of a *saf_write_subset_relation* call if the data that would have been written in the subset relation is identical to some other relation data already written to the database.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle. (low-cost)
- `oldrel` must be a valid relation handle. (low-cost)
- `oldrel` must be same as `rel` to re-use data written to it. (no-cost)
- `oldrel` must be same as `rel` to re-use data written to it. (no-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *saf_write_subset_relation*: 12.12: *Write a subset relation*
- *Subset Relations*: Introduction for current chapter

Write a subset relation

`saf_write_subset_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_write_subset_relation (SAF_ParMode pmode, SAF_Rel *rel, hid_t A_type, void *A_buf,  
                               hid_t B_type, void *B_buf, SAF_Db *file)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation whose data is to be written.
- `A_type`: The type of `A_buf` (if not already supplied through the *saf_declare_subset_relation* call).
- `A_buf`: The data (if not already supplied through the *saf_declare_subset_relation* call).
- `B_type`: The type of `B_buf` (if not already supplied through the *saf_declare_subset_relation* call).
- `B_buf`: The data (if not already supplied through the *saf_declare_subset_relation* call).
- `file`: The optional destination file to write the data to. A null pointer for this argument indicates that the data is to be written to the same file as `rel`.

Description: This call writes relation data to the specified file.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle. (low-cost)
- `A_buf` should be specified either here or in the *saf_declare_subset_relation*. (low-cost)
- `B_buf`, if present, should be specified either here or in the. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: `SAF_EACH` mode is a collective call where each of the `N` tasks provides a unique relation. `SAF` will create a single HDF5 dataset to hold all the data and will create `N` blobs to point into nonoverlapping regions in that dataset.

Issues: Overwrite is not currently allowed.

See Also:

- *[saf_declare_subset_relation](#)*: 12.5: *Declare a subset relation*
- *[Subset Relations](#)*: Introduction for current chapter

Topology Relations

Topology relations are used to define the inter-relationships between the members of a collection and how those members are knitted together to form a *mesh*. For more information, see **Relation Notes** and *[saf_declare_topo_relation](#)*. Also, see this detailed discussion of the difference between mapping of *degrees of freedom* of a field to the members of a collection and the mapping of the members of a collection to themselves to indicate topological relationships, (github.com/markcmiller86/SAF/blob/master/src/safapi/docs/dof_maps_and_topo_rels.pdf)

Members

Declare a topological relation

`saf_declare_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
SAF_Rel * saf_declare_topo_relation (SAF_ParMode pmode, SAF_Db *db, SAF_Set *set,
                                     SAF_Cat *pieces, SAF_Set *range_set,
                                     SAF_Cat *range_cat, SAF_Cat *storage_decomp,
                                     SAF_Set *my_piece, SAF_RelRep *trtype, hid_t A_type,
                                     void *A_buf, hid_t B_type, void *B_buf, SAF_Rel *rel)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The dataset where the new relation will be created.
- `set`: The containing set of the collection whose members are being sewn together by the relation.
- `pieces`: The collection of members that are being sewn together.
- `range_cat`: Together, `range_set` and `range_cat` identify the range of the relation (e.g., collection used to glue the pieces together). There are really only two valid values for `RANGES_S`: the set `set` or the set `my_piece`.
- `storage_decomp`: The decomposition of `set` upon which the relation is stored.
- `my_piece`: The piece of the decomposition being declared here.
- `trtype`: The relation types. One of `SAF__STRUCTURED`, `SAF__UNSTRUCTURED`, or `SAF__ARBITRARY`.
- `A_type`: The type of the data in `A_buf`.
- `A_buf`: The buffer. Pass `NULL` if you would rather provide this in the write call.
- `B_type`: The type of the data in `B_buf`.

- `B_buf`: The buffer. Pass `NULL` if you would rather provide this in the write call.
- `rel`: [OUT] Optional memory that will be initialized (and returned) to point to the new relation.

Description: A topology relation describes how the individual members of a collection are sewn together to form a mesh. A topology relation is composed of one or more steps. Each step in the relation represents a portion of the dimensional cascade in representing an N dimensional set in terms of a bunch of $N-1$ dimensional sets that form its boundary, which are, in turn, represented by a bunch of $N-2$ dimensional sets, etc. The last step is always on zero dimensional sets (e.g., `SAF__CELLTYPE_POINT` cells). Typically, there is only ever one step from a primitive decomposition to `SAF__CELLTYPE_POINT` cells (nodes). In this case, the topology relation is a list describing the nodal connectivity for each element in the decomposition.

In the case of `SAF__STRUCTURED`, all other arguments are currently ignored and rectangular structure is assumed. Later, different types of structure will be supported. In the case of `SAF__UNSTRUCTURED`, `ABUF` is a pointer to one value of type `DATA_TYPE` representing the number of range references for each member of the domain collection and `BBUF` is an array of type `DATA_TYPE` containing that number of range references for each member of the domain. In the case of `SAF__ARBITRARY`, `ABUF` is a pointer to an array of values of type `DATA_TYPE` equal to the size of the domain collection. Each value in the `ABUF` array represents the number of range references for the corresponding number of the domain collection. `BBUF` is a pointer to the range references.

By convention, a topology relation should be declared on the maximal set in the subset inclusion lattice (e.g., the top-most set in the subset inclusion lattice) for which it makes sense to define the topology. If the topology relation is, in fact, stored in non-contiguous chunks, then the client should use the `storage_decomp` argument of the topology relation to declare that the relation data is stored in pieces on the given decomposition.

Preconditions:

- `pmode` must be valid. (low-cost)
- `set` must be a valid set handle. (low-cost)
- `pieces` must be a valid category. (low-cost)
- `range_set` must be a valid set. (low-cost)
- `range_cat` must be a valid category. (low-cost)
- `MY_PIECES` must be a valid handle. (low-cost)
- `storage_decomp` must be either the self decomposition or a valid cat handle. (low-cost)
- `trtype` must be a consistent relation representation handle. (low-cost)
- `trtype` must be a valid topology representation. (low-cost)
- If supplied, `A_type` must be an integer type (or `SAF__HANDLE` if decomposed). (low-cost)
- `A_type` must be supplied if `A_buf` is supplied. (low-cost)
- `B_type` must be an integer type or handle type. (low-cost)
- `B_type` must be supplied if `B_buf` is supplied. (low-cost)
- `A_type` must be handle if storage decomposition is not self. (low-cost)

Return Value: On success, returns a pointer to the new relation: either the `rel` argument or an allocated relation. Returns the null pointer on failure.

See Also:

- [Topology Relations](#): Introduction for current chapter

Get description of topological relation

`saf_describe_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_describe_topo_relation (SAF_ParMode  pmode,  SAF_Rel  *rel,  SAF_Set  *set,
                               SAF_Cat  *pieces, SAF_Set  *range_set, SAF_Cat  *range_cat,
                               SAF_Cat  *storage_decomp,  SAF_RelRep  *trtype,
                               hid_t  *data_type)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation to be described.
- `set`: [OUT] The containing set of the collection that is sewn together by the relation.
- `pieces`: [OUT] The collection of members that are sewn together.
- `range_cat`: [OUT] Together the `RANGE_S` and `RANGE_C` pair identifies the collection used to glue the pieces together.
- `storage_decomp`: [OUT] The decomposition of `set` upon which the relation is actually stored.
- `trtype`: [OUT] The topology relation type.
- `data_type`: [OUT] The type of the data.

Description: This function returns information about a topological relation.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle for all participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Topology Relations*: Introduction for current chapter

Find topological relations

`saf_find_topo_relations` is a function defined in `rel.c`.

Synopsis:

```
int saf_find_topo_relations (SAF_ParMode  pmode,  SAF_Db  *db,  SAF_Set  *set,
                             SAF_Set  *topo_ancestor, int *num, SAF_Rel **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database in which to search for topology relations.
- `set`: The set whose topology is sought.
- `topo_ancestor`: [OUT] In many cases, the topology for a given set is known only on some ancestor of the set. This return value indicates that ancestor. If `SAF__EQUIV` for `set` and `topo_ancestor` is true, then the

topology relations found by this call are indeed those defined on the specified set. Otherwise, they are defined on the `topo_ancestor`.

- `num`: For this and the succeeding argument, (see **Returned Handles**).
- `found`: For this and the preceding argument, (see **Returned Handles**).

Description: This function will find the topological relations governing a given set. Note that if the given set is one that is the subset of where the topological relations are actually declared, this call will return that set and the topological relation(s) defined on that set.

Preconditions:

- `pmode` must be valid. (low-cost)
- `set` must be a valid handle. (low-cost)
- `topo_ancestor` must be non-null. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: What if there are multiple topological relations governing a given set which are declared on different sets? The `topo_ancestor` argument needs to be of length **Pnum_rels*.

The `topo_ancestor` argument is not actually referenced or returned by this function.

See Also:

- *Topology Relations*: Introduction for current chapter

Get datatype and size for a topological relation

`saf_get_count_and_type_for_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_get_count_and_type_for_topo_relation (SAF_ParMode  pmode,  SAF_Rel  *rel,
                                             SAF_RelTarget *target, SAF_RelRep *Prep-
                                             Type, size_t  *abuf_sz, hid_t  *abuf_type,
                                             size_t  *bbuf_sz, hid_t  *bbuf_type)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation handle.
- `target`: Targeting information.
- `PrepType`: [OUT] The mapping representation type (arbitrary, structured, or unstructured). The caller may pass value of `NULL` for this parameter if this value is not desired.
- `abuf_sz`: [OUT] The number of items that would be placed in the A-buffer by a call to the *saf_read_topo_relation* function. The caller may pass value of `NULL` for this parameter if this value is not desired.
- `abuf_type`: [OUT] The type of the items that would be placed in the A-buffer by a call to the *saf_read_topo_relation* function. The caller may pass value of `NULL` for this parameter if this value is not desired.

- `bbuf_sz`: [OUT] The number of items that would be placed in the B-buffer by a call to the *saf_read_topo_relation* function. The caller may pass value of NULL for this parameter if this value is not desired.
- `bbuf_type`: [OUT] The type of the items that would be placed in the B-buffer by a call to the *saf_read_topo_relation* function. The caller may pass value of NULL for this parameter if this value is not desired.

Description: This function is used to retrieve the number and type of A-buffer and B-buffer data items that would be retrieved by a call to the *saf_read_topo_relation* function. This function may be used by the caller to determine the sizes of the buffers needed when pre-allocation is desired or to determine how to traverse the buffer(s) returned by the *saf_read_topo_relation* function.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle. (low-cost)
- If targeting of storage decomposition is used, the read must be a SAF__ALL mode read. (low-cost)

Return Value: The constant SAF__SUCCESS is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *saf_read_topo_relation*: 13.6: *Read topological relation data*
- *Topology Relations*: Introduction for current chapter

Is topological relation stored on self

`saf_is_self_stored_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_is_self_stored_topo_relation (SAF_ParMode pmode, SAF_Rel *rel, hbool_t *Presult)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The handle of the topological relation which is to be examined.
- `Presult`: [OUT] A pointer to caller supplied memory which is to receive the result of the test: true if the relation is self stored or false if it is stored on a decomposition. Note that it is permitted for the caller to pass a value of NULL for this parameter.

Description: This function is used by a client to test if a topology relation is stored on self. The boolean result is returned by reference.

Preconditions:

- `pmode` must be valid. (low-cost)

Return Value: The constant SAF__SUCCESS is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Topology Relations*: Introduction for current chapter

Read topological relation data

`saf_read_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_read_topo_relation (SAF_ParMode  pmode,  SAF_Rel  *rel,  SAF_RelTarget  *target,
                           void **abuf, void **bbuf)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The topology relation to be read.
- `target`: Relation targeting information.
- `abuf`: The returned data. See [saf_declare_topo_relation](#).
- `bbuf`: The returned data. See [saf_declare_topo_relation](#).

Description: This function reads topological relation data from the database.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid relation handle. (low-cost)
- `abuf` must be non-null. (low-cost)
- Either both `abuf` and `bbuf` point to NULL or both `abuf` and `bbuf` do not point to NULL. (low-cost)
- If targeting of storage decomposition is used, the read must be a `SAF__ALL` mode read. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [saf_declare_topo_relation](#): 13.1: *Declare a topological relation*
- [Topology Relations](#): Introduction for current chapter

Set the destination form of a topological relation

`saf_target_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_target_topo_relation (SAF_RelTarget *target, SAF_Set *range_set, SAF_Cat *range_cat,
                              SAF_Cat *decomp, SAF_RelRep *trtype, hid_t data_type)
```

Formal Arguments:

- `target`: [OUT] Relation targeting information to be initialized by this function.
- `range_set`: Optional set.
- `range_cat`: Together the `range_set` this identifies the target collection to be used to glue the pieces together. Currently both of these parameters are ignored.
- `decomp`: The optional target decomposition.
- `trtype`: The optional target relation types. Currently this parameter is ignored.
- `data_type`: The optional target data type.

Description: This function establishes the target (destination) form of topo relation data during either read or write. When used prior to a write call, it establishes the form of data in the file. When used prior to a read call, it establishes the form of data as desired in memory.

Preconditions:

- Must pass non-null target information. (low-cost)
- `decomp` must be either `NOT_SET`, `SELF_DECOMP` or a valid cat handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: Not all features have been implemented yet.

See Also:

- *Topology Relations*: Introduction for current chapter

Write topological relation data

`saf_write_topo_relation` is a function defined in `rel.c`.

Synopsis:

```
int saf_write_topo_relation (SAF_ParMode pmode, SAF_Rel *rel, hid_t A_type, void *A_buf,
                             hid_t B_type, void *B_buf, SAF_Db *file)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `rel`: The relation handle.
- `A_type`: See *saf_declare_topo_relation*.
- `A_buf`: See *saf_declare_topo_relation*.
- `B_type`: See *saf_declare_topo_relation*.
- `B_buf`: See *saf_declare_topo_relation*.
- `file`: The optional destination file. By default (if null) the data is written to the same file to which `rel` belongs.

Description: This function writes topological relation data to the given file.

Preconditions:

- `pmode` must be valid. (low-cost)
- `rel` must be a valid rel handle. (low-cost)
- `A_type` must be supplied if `A_buf` is supplied. (low-cost)
- A- and B-buffers and types must be set appropriately. (high-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: `SAF_EACH` mode is a collective call where each of the `N` tasks provides a unique relation. `SAF` will create a single HDF5 dataset to hold all the data and will create `N` blobs to point into nonoverlapping regions in that dataset.

See Also:

- *saf_declare_topo_relation*: 13.1: *Declare a topological relation*

- *Topology Relations*: Introduction for current chapter

Relations

No description available.

Members

The null relation handle

`SAF_NULL_REL` is a macro defined in `saf.h`.

Synopsis:

`SAF_NULL_REL` (Db)

Description: This macro evaluates to the relation handle for the null relation of the database. The null relation handle is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See **Constants**).

See Also:

- *Relations*: Introduction for current chapter

Field Templates

A field template represents all the abstract features of a field. That is, those features that are immutable as the data is exchanged between one scientific computing client and another. By contrast, a field (which is defined in terms of a field template) represents all the features of a field that might possibly change as the field is exchanged between scientific computing clients.

By and large, a field template can be viewed as defining a class of fields.

Members

The null field template handle

`SAF_NULL_FTmpl` is a macro defined in `saf.h`.

Synopsis:

`SAF_NULL_FTmpl` (Db)

Description: This macro evaluates to the field template handle for the null field template of the database. The null field template handle is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See **Constants**).

See Also:

- *Field Templates*: Introduction for current chapter

Declare a field template

`saf_declare_field_tmpl` is a function defined in `ftmpl.c`.

Synopsis:

```
SAF_FieldTmpl * saf_declare_field_tmpl (SAF_ParMode  pmode,  SAF_Db    *db,  const
                                         char  *name, SAF_Algebraic *atype, SAF_Basis *basis,
                                         SAF_Quantity *quantity, int  num_comp,
                                         SAF_FieldTmpl *ctmpl, SAF_FieldTmpl *ftmpl)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database handle in which to create the template.
- `name`: The name of the field template.
- `atype`: The algebraic type: `SAF__ALGTYPE_SCALAR`, `SAF__ALGTYPE_VECTOR`, `SAF__ALGTYPE_TENSOR`, `SAF__ALGTYPE_SYMTENSOR`, `SAF__ALGTYPE_FIELD`. If the algebraic type is `SAF__ALGTYPE_FIELD`, then all we know about the field is that it references other fields (i.e., an indirect field). Therefore, the next four arguments are not applicable. More generalized user defined type definitions will be available in later implementations.
- `basis`: The basis. Not implemented yet. Pass null
- `quantity`: The quantity. See [saf_declare_quantity](#) for quantity definitions and how to define new quantities.
- `num_comp`: Number of components. Although this may often be inferred from `atype`, SAF currently does no work to infer it. Pass `SAF__NOT_APPLICABLE_INT` if this template will be used in the declaration of an inhomogeneous field. Otherwise, pass the number of components. For a simple scalar field, the number of components is 1. See Fields for further discussion of inhomogeneous fields.
- `ctmpl`: This is an array of `NUM_COMPS` field template handles that comprise the composite field template or NULL if there are no component field templates. Pass NULL if this field template will be used in the declaration of an INhomogeneous field.
- `ftmpl`: Returned field template handle for composite fields. If the algebraic type (`atype`) is `SAF__ALGTYPE_FIELD`, then the returned field template may be used as a state template (see **State Templates**).

Description: This function declares a field template. A field template defines the implementation independent features of a field such as its algebraic type, the quantity it represents, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `ftmpl` must be non-null. (low-cost)
- `name` must be non-null. (low-cost)
- `num_comp` >= 1. (low-cost)
- `ctmpl` must be non-NULL if `num_comp` > 1. (low-cost)
- `atype` must be a valid algebraic type handle. (low-cost)
- `ctmpl` may be NULL only if `num_comp` == 1 and `atype` must be direct. (low-cost)
- `ctmpl` must be NULL if components are not appropriate. (low-cost)
- `basis` must be a valid basis handle or NULL. (low-cost)
- `quantity` must be a valid quantity handle if supplied. (low-cost)

Return Value: A pointer to the new field template handle is returned on success, either the `ftmpl` argument if non-null or a freshly allocated handle. A null pointer is returned on failure.

Issues: It would be better if we could create new field types (templates) from old ones, or alternatively, construct new algebraic types from old ones. This is most apparent when the `alg_type` is `SAF__FIELD`. This is the C-language equivalent of a void. *It tells us only that it is a reference to a field (in fact, the lib doesn't care if you pass "SAF__Rel" objects here) but does not say what kind of fields (e.g. field templates) it should reference. One might be inclined to think that the component fields templates can serve to define the type of references of a "SAF__FIELD" entity. However, this is not so. The component field templates define the *component fields.* We can illustrate by an example.

Suppose we have a time series of the coordinate field of an airplane. Each instant in time of the coordinate field is a field on `SPACE`. To create the coordinates as a function of time, we create a field on `TIME` whose `alg_type` is `SAF__FIELD`. What are its component fields? If we want somehow to use the component fields to define the kinds of field this `SAF__FIELD` entity refers to, we'd specify the field template for coordinate fields on `SPACE` as the component field template here. However, if we do that, how do we then specify the components of coordinates as a function of time, namely `x(t)`, `y(t)`, and `z(t)` whose field template is on `TIME`. We can't! In essence, we need to be able to say what kind of `SAF__FIELD` this entity is by passing a list of field templates as the algebraic type. Or, more specifically, we need to generalize the notion of algebraic type and allow the client to build new types from old ones. That will be deferred to a later release. For now, the best we can do with this is equivalent to a void* field reference.

We could extend the API and allow a list of field templates for the `atype` argument. Alternatively, would could allow the member field types to come in via the component fields of the component fields of a `SAF__FIELD` template. However, that is rather convoluted.

See Also:

- [*saf_declare_quantity*](#): 20.9: *Declare a new quantity*
- [*Field Templates*](#): Introduction for current chapter

Get a description of a field template

`saf_describe_field_tmpl` is a function defined in `ftmpl.c`.

Synopsis:

```
int saf_describe_field_tmpl (SAF_ParMode pmode, SAF_FieldTmpl *ftmpl, char **name,
                             SAF_Algebraic *alg_type, SAF_Basis *basis, SAF_Quantity *quantity,
                             int *num_comp, SAF_FieldTmpl **ctmpl)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `ftmpl`: The field template to be described.
- `name`: [OUT] The returned name. Pass `NULL` if you do not want the name returned. (see **Returned Strings**).
- `alg_type`: [OUT] The returned algebraic type. Pass `NULL` if you do not want the type returned.
- `basis`: [OUT] The returned basis. Pass null if you do not want the basis returned.
- `quantity`: [OUT] The returned quantity. Pass null if you do not want the name returned.
- `num_comp`: [OUT] The returned number of components. Pass `NULL` if you do not want the name returned. Note that if the field template is associated with an `INhomogeneous` field, the returned value will always be `SAF__NOT_APPLICABLE_INT`.
- `ctmpl`: [OUT] The returned array of component field template handles. Pass `NULL` if you do not want the array returned. If the field template is associated with an `INhomogeneous` field, the returned value, if requested, will always be `NULL`. (If the field template does not point to other field templates then this argument will be untouched by this function.)

Description: This function returns information about a field template.

Preconditions:

- `pmode` must be valid. (low-cost)
- `ftempl` must be a valid field template handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Field Templates*: Introduction for current chapter

Find field templates

`saf_find_field_tmpls` is a function defined in `ftempl.c`.

Synopsis:

```
int saf_find_field_tmpls (SAF_ParMode  pmode,  SAF_Db    *db,  const  char  *name,
                        SAF_Algebraic *atype,  SAF_Basis *basis, SAF_Quantity *quantity,
                        int *num, SAF_FieldTpl **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: the database context for this search (previously retrieved from `base_space`)
- `name`: The name of the field template.
- `atype`: The algebraic type to limit the search to. Pass `NULL` if you do not want to limit the search by this parameter.
- `basis`: The basis to limit the search to. Pass `NULL` if you do not want to limit the search by this parameter.
- `quantity`: The quantity to search for. Pass `NULL` if you do not want to limit the search by this parameter.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function finds field templates according to specific search criteria.

Preconditions:

- `pmode` must be valid. (low-cost)
- `atype` must be a valid algebraic handle if supplied. (low-cost)
- `basis` must be a valid basis handle if supplied. (low-cost)
- `quantity` must be a valid quantity handle if supplied. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Field Templates*: Introduction for current chapter

Get an attribute with a field template

`saf_get_field_tmpl_att` is a function defined in `ftempl.c`.

Synopsis:

```
int saf_get_field_tmpl_att (SAF_ParMode pmode, SAF_FieldTmpl *ftmpl, const char *name,  
                           hid_t *type, int *count, void **value)
```

Description: This function is identical to the generic [saf_get_attribute](#) function except that it is specific to `SAF__FieldTmpl` objects to provide the client with compile time type checking. For a description, see [saf_get_attribute](#).

See Also:

- [saf_get_attribute](#): 23.1: Read a non-sharable attribute
- [Field Templates](#): Introduction for current chapter

Put an attribute with a field template

`saf_put_field_tmpl_att` is a function defined in `ftempl.c`.

Synopsis:

```
int saf_put_field_tmpl_att (SAF_ParMode pmode, SAF_FieldTmpl *ftmpl, const char *name,  
                           hid_t type, int count, const void *value)
```

Description: This function is identical to the generic [saf_put_attribute](#) function except that it is specific to `SAF__FieldTmpl` objects to provide the client with compile time type checking. For a description, see [saf_put_attribute](#).

See Also:

- [saf_put_attribute](#): 23.2: Create or update a non-sharable attribute
- [Field Templates](#): Introduction for current chapter

Fields

A field is some physical phenomenon known or expected to have *value* at every point in the set over which the field is defined. In other words, a field represents some continuous (as opposed to discrete) function which is defined over the infinite point set specified as the *base space* for the field.

In [SAF](#), we divide the notion of a field into two pieces; a wholly abstract piece free of the details of how a field is *implemented*, and an implementation specific piece. The abstract piece is called a *field template*. See [saf_declare_field_tmpl](#) for more information. Essentially, a field template defines a *class* of fields. The implementation specific piece of a field is called, simply, a *field*.

Presently, [SAF](#) requires the client to create a new scientific modeling primitive (e.g. a field object) for each instance of a field's data. For example, if you have a pressure field that is evolving with time, each time the field's data is written to the database, the client needs to declare a new field object. The client is **not** simply writing more data for the same field object. As more experience is gained with the data model and implementation, this behavior will be modified to be more natural. It is ok for a client to simply create a field with the same name, etc. This will not cause any problems in [SAF](#). For example, if a client creates several pressure fields, all of which are instances of the same field at different points in time, that is ok. However, the client will probably want to organize those fields into a more aggregate *field of fields* (e.g. a field whose "values" are other fields on other base spaces). We call such a field an *indirect* field. In fact, the *states and suites* interface is provided as a convenient way to construct an indirect field representing various states of the problem output by the client. See **States** or [saf_declare_state_group](#). However, there are a variety of

situations in which a client may want to define an indirect field. The remaining portions of this chapter introduction discuss these situations in some detail. We'll begin with some definitions.

Degrees Of Freedom(**dofs):** The *degrees of freedom* or *dofs* of a field is the name we give to the data associated with the field. Typically, the dofs are the problem sized arrays of floats or doubles representing some independent (or dependent) variables of a simulation. We call these datums *degrees of freedom* because, within the context of SAF, they are the degrees of freedom **in the representation** of the field. It is important to recognize this context of the *representation* of the field. That is what SAF is solely concerned with: representing fields so that other clients can read and interpret them. In this context, every datum represents a degree of freedom. This sense of degree of freedom should **not** be confused with, for example, similar terminology in the linear system of equations a client might be solving. That is an entirely different context in which similar terminology is used to describe those datums that effect the solution of the system of equations being solved. SAF is concerned with data that effects the representation of the field. Why don't we call these *values*? Because the word "values" implies that the field is, in fact, equal to these numbers for some (maybe many) points in the base-space. And, this is only true when the field's evaluation function is *interpolating*. That is, the interpolation functions **pass through** the dofs controlling the interpolation. This is most certainly not true for a field represented by, for example, a Fourier series.

Indirect Field: An *indirect field* is any field whose algebraic type is `SAF__ALGTYPE_FIELD`. Equivalently, this means that if you were to call `saf_read_field` for such a field, you would obtain a bunch of `SAF__Field` field handles. Likewise, when the algebraic type of a field is **not** `SAF__ALGTYPE_FIELD`, the field is not an indirect field and we, instead, call it a *direct field*. Note that indirection is, in general, recursive. An indirect field can refer to fields that are themselves indirect fields. An example of an indirect field is the pressure over a mesh as a function of time for 9 time instances. There would be 9 instances of pressure fields on the mesh, one for each time instant. Each of these fields is just one of the instances of the pressure on the mesh. To characterize the pressure field's variation over time, we would define another field on the time base space having 9 dofs. Each dof would be a different one of the pressure fields over the mesh as illustrated in figure "indirect field-1.gif".

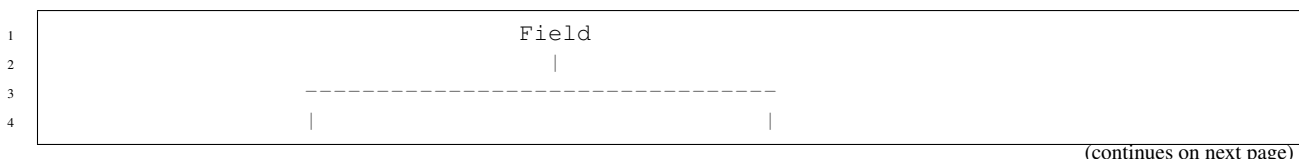
Homogeneous Field: A *homogeneous field* is any field whose *defining characteristics* **do not vary** over the base space upon which the field is declared. We include in "*defining characteristics*" all those parameters used to declare a field and its field template such as algebraic type, number of components, quantity, units, component interleave, component order, evaluation function and even its storage.

Any field that is not homogeneous is *inhomogeneous*. An example of an inhomogeneous field is a stress tensor defined over a 3D rocket body and its 2D fins. Over the 3D body, the field is a 3D symmetric tensor and over the 2D fins it is a 2D symmetric tensor. This is illustrated in "indirect field-2.gif".

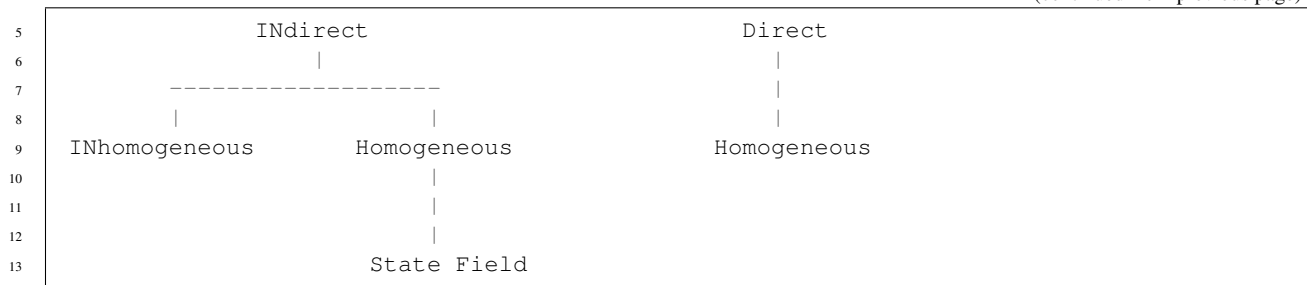
Another example is a coordinate field of a mesh whose storage is decomposed into separate chunks, one for each processor in a parallel decomposition. This is illustrated in "indirect field-3.gif".

SAF deals with inhomogeneous fields by breaking them up, recursively in general, into homogeneous pieces. Thus, the data for an inhomogeneous field is the handles to these field pieces. An inhomogeneous field is, therefore, also an indirect field. Furthermore, if a field is inhomogeneous, all bets are off about **any** of the field's *defining characteristics*. All that can be said, for sure, about an inhomogeneous field is that there is some decomposing collection of the field's base-space upon which it is *presumably* piecewise homogeneous. We say *presumably* here because any piece of an inhomogeneous field can itself be inhomogeneous so that, in general, its decomposition into homogeneous pieces is recursive.

With all of this information, we can construct a *pseudo class-hierarchy* for these various kinds of fields.



(continued from previous page)



Because an inhomogeneous field is also an indirect field, it is often convenient when talking about both inhomogeneous indirect fields and homogeneous indirect fields to simply refer to the two as inhomogeneous and indirect fields, respectively. There are some important conceptual differences between inhomogeneous and indirect fields worth mentioning here.

First, one requirement of the various fields comprising an inhomogeneous field is that the union of the base-spaces of all the homogeneous pieces **must** form a decomposition of the base-space of the inhomogeneous aggregate. **SAF** enforces this condition and will not permit a client to construct an inhomogeneous field for which this is not true.

Second, it does not make sense to conceive of interpolating between the pieces of an inhomogeneous field in the same way we might want to interpolate between the pieces of an indirect field. For example, it doesn't make sense to try to interpolate between the stress tensor on the fins and the stress tensor on the rocket body of the inhomogeneous field in "indirect field-2.gif" while it does make sense to try to interpolate between the 4th and 5th instances of the pressure field in "indirect field-1.gif".

Third, for all homogeneous fields, the number of dofs read from and written to a field is the product of the number of components in the field, the size of the collection the field's dofs are associated with and the association ratio (see [saf_declare_field](#)). This is true for homogeneous, direct fields where the dofs might be floats or doubles as well as homogeneous, indirect fields where the dofs are handles to other fields. However, for inhomogeneous fields, the number of field handles to be read and written is determined by the size of the decomposing collection upon which the field is presumably piecewise homogeneous. That collection is what determines the number of pieces the field is decomposed into.

FIELD TARGETING: **SAF** now offers some limited ability to *transform* a field during read. Currently, this capability is available **only during read**. Transformations during write will be made available later. Currently, on read, a client can invoke the following transformations:

- a. changes in precision (single\rightarrowdouble)
- b. changes in interleave (vector\rightarrowcomponent)
- c. changes in storage decomposition (self\rightarrowother immediate). By *immediate* we mean a decomposition which is immediately below the self set in the subset relation graph.

The targeting function, [saf_target_field](#), is used to tell **SAF** to invoke such transforms during read. The intent is that a reader will call [saf_target_field](#) before making a call to read the field. The target call will indicate the intended form of the field in the destination. Once targeting has been setup with a call to [saf_target_field](#), a call to [saf_read_field](#) will *do the right thing* resulting in the altered field in the destination's buffers. This is an experimental capability and interface. Field targeting will only work on serial **SAF** clients or single processor parallel **SAF** clients (i.e. **SAF** clients that have opened the database on just one processor).

Soon, **SAF** will offer some limited ability to *transform* a field during read or write. The intent is that a reader or writer will call [saf_target_field](#) before making a call to read or write the field. The target call will indicate the intended form of the field in the destination (the database during write or the client's memory during read). Once targeting has been setup with a call to [saf_target_field](#), a call to [saf_read_field](#) or [saf_write_field](#) will *do the right thing* resulting in the altered field in the destination's buffers.

Members

Conveniently specify a constant field

SAF_CONSTANT is a macro defined in saf.h.

Synopsis:

SAF_CONSTANT (db)

Description: This macro provides a convenient way to specify four of the args, *coeff_assoc*, *assoc_ratio*, *eval_coll*, and *eval_func* of the *saf_declare_field* call. Use it when you have a constant field. The db argument is meant to represent the database handle. See *saf_declare_field* for a more detailed description.

See Also:

- *saf_declare_field*: 16.12: *Declare a field*
- *Fields*: Introduction for current chapter

Conveniently specify a decomposition-centered field

SAF_DECOMP is a macro defined in saf.h.

Synopsis:

SAF_DECOMP (D)

Description: This macro provides a convenient way to specify four of the args, *coeff_assoc*, *assoc_ratio*, *eval_coll*, and *eval_func* of the *saf_declare_field* call. Use it when you have a field in which you have 1 degree of freedom for each set in a collection of sets forming a decomposition of their parent set. For example, if you have a collection of sets where each set represents one processor's piece and you wish to characterize a field that represents the min (or max) of some field over each piece. The argument D is meant to be a collection category for a non-primitive collection of set known to form a decomposition of the set upon which the field is being defined.

See Also:

- *saf_declare_field*: 16.12: *Declare a field*
- *Fields*: Introduction for current chapter

Conveniently specify a node-centered field

SAF_NODAL is a macro defined in saf.h.

Synopsis:

SAF_NODAL (N, Z)

Description: This macro provides a convenient way to specify four of the args, *coeff_assoc*, *assoc_ratio*, *eval_coll*, and *eval_func* of the *saf_declare_field* call. Use it when you have what is often referred to as a *node centered field*. The argument N is meant to be a collection category representing collections of SAF__CELLTYPE_POINT (nodes) cells. The argument Z is meant to be a collection category representing collections of element cells.

See Also:

- *saf_declare_field*: 16.12: *Declare a field*
- *Fields*: Introduction for current chapter

The null field handle

`SAF_NULL_FIELD` is a macro defined in `saf.h`.

Synopsis:

`SAF_NULL_FIELD` (Db)

Description: This macro evaluates to the field handle for the null field of the database. The null field handle is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See **Constants**).

See Also:

- *Fields*: Introduction for current chapter

More meaningful alias for `SAF_TOTALITY`

`SAF_WHOLE_FIELD` is a symbol defined in `saf.h`.

Synopsis:

`SAF_WHOLE_FIELD`

Description: The `SAF__TOTALITY` subset relation representation is most often only ever used during a *saf_write_field* call to indicate the entire field is being written rather than just a portion. However, in that context, the meaning of a *totality* is obscured. So, we provide `SAF__WHOLE_FIELD` as a more meaningful alias for that value. In addition, this macro replaces the three args, `MEMBER_COUNT`, `REQUEST_TYPE`, `MEMBER_IDS`, used in a partial *saf_write_field* call.

See Also:

- *saf_write_field*: 16.23: Write the data for a field
- *Fields*: Introduction for current chapter

Conveniently specify a zone-centered field

`SAF_ZONAL` is a macro defined in `saf.h`.

Synopsis:

`SAF_ZONAL` (Z)

Description: This macro provides a convenient way to specify four of the args, *coeff_assoc*, *assoc_ratio*, *eval_coll*, and *eval_func* of the *saf_declare_field* call. Use it when you have what is often referred to as a *zone centered field*. The argument Z is meant to be a collection category representing collections of element cells.

See Also:

- *saf_declare_field*: 16.12: Declare a field
- *Fields*: Introduction for current chapter

Find the parent of a component field

`_saf_find_parent_field` is a function defined in `field.c`.

Synopsis:


```
SAF_Field * _saf_find_parent_field(SAF_ParMode  pmode,    SAF_Field  *component_field,
                                   SAF_Field *retval)
```

Formal Arguments:

- `component_field`: Field for which we are searching for a parent.
- `retval`: [OUT] Optional buffer in which to store the result. If this is `NULL` then a buffer will be allocated for the return value.

Description: Find the parent of a component field. Find the field who has the input field as a component. This function caches the parent field in the component field in order to keep the performance good.

Preconditions:

- `pmode` must be valid. (low-cost)

See Also:

- *Fields*: Introduction for current chapter

Queries whether data has been written

`saf_data_has_been_written_to_comp_field` is a function defined in `field.c`.

Synopsis:

```
int saf_data_has_been_written_to_comp_field(SAF_ParMode  pmode,    SAF_Field  *field,
                                             hbool_t *Presult)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field handle.
- `Presult`: [OUT] A pointer to caller supplied memory which is to receive the answer to the question. A value of true is saved at this location if the field has had data written to it, false if not.

Description: Does a composite or component field have written data corresponding to this field.

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle for all participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Fields*: Introduction for current chapter

Does field have data

`saf_data_has_been_written_to_field` is a function defined in `field.c`.

Synopsis:

```
int saf_data_has_been_written_to_field(SAF_ParMode pmode, SAF_Field *field, hbool_t *Pre-
                                       sult)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field handle.
- `Presult`: [OUT] A pointer to caller supplied memory which is to receive the answer to the question. A value of true is saved at this location if the field has had data written to it, false if not.

Description: This function is used to check if a given field has a valid blob id (which it would if it has had data written to it and doesn't if it has not).

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle for all participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Fields](#): Introduction for current chapter

Declare a field as a candidate coordinate field

`saf_declare_coords` is a function defined in `field.c`.

Synopsis:

```
int saf_declare_coords (SAF_ParMode pmode, SAF_Field *field)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field to be characterized as a coordinate field.

Description: Use the function to indicate that a particular field is a coordinate field. This merely identifies a field as a candidate coordinate field. More than one field may serve as the coordinate field for a set. For example, in engineering codes, there are the deformed and un-deformed coordinates.

Preconditions:

- `pmode` must be valid. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Fields](#): Introduction for current chapter

Declare default coordinates of a given set

`saf_declare_default_coords` is a function defined in `field.c`.

Synopsis:

```
int saf_declare_default_coords (SAF_ParMode pmode, SAF_Set *base, SAF_Field *field)
```

Formal Arguments:

- `pmode`: the parallel mode

- `base`: the base space set whose default coordinates are being declared
- `field`: the field to serve as the default coordinates

Description: Many fields might be suitable to serve as a coordinate field. Absolute coordinates and displacements are just two examples. This reference manual is not the appropriate place to go into the specific mathematical requirements for a field to serve as a coordinate field. However, recognizing that more than one field can serve as a coordinate field raises the issue, which field should be used as the coordinate field if nothing else is specified. This function declares which field ought to be treated as the default coordinates.

Note that in order for a field to be declared as the default coordinate field for a set, the field must first be declared as a coordinate field.

Preconditions:

- `pmode` must be valid. (low-cost)
- `base` must be a valid set handle. (low-cost)
- `field` must be a valid field handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Fields](#): Introduction for current chapter

Declare a field

`saf_declare_field` is a function defined in `field.c`.

Synopsis:

```
SAF_Field * saf_declare_field(SAF_ParMode pmode, SAF_Db *db, SAF_FieldTpl *ftmpl,
                             const char *name, SAF_Set *base_space, SAF_Unit *unit,
                             SAF_Cat *homog_decomp, SAF_Cat *coeff_assoc, int assoc_ratio,
                             SAF_Cat *eval_cat, SAF_Eval *eval_func, hid_t data_type,
                             SAF_Field *comp_fds, SAF_Interleave comp_intlv, int *comp_order,
                             void **bufs, SAF_Field *fld)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database where the new field will be created.
- `ftmpl`: The field template handle for this field. Recall that the field template describes the abstract features of the field, including the quantity the field represents, and the algebraic type. The field being created in this `saf_declare_field` call is simply an instance of the abstract field characterized by the field template passed as this argument.
- `name`: The name of this field. If a writer client declares different fields with the same name, a reader client that searches for fields by name will find multiple matches. However, it is ok to declare different fields with the same name.
- `base_space`: The `base_space` of this field
- `unit`: The specific units of measure. If in the field template, the quantity was not specified, then the only valid value that can be passed for units is `SAF__NOT_APPLICABLE_UNIT`. Otherwise, pass `SAF__NOT_SET_UNIT` if you do not want to specify units for the field or pass one of the valid units of measure.

- `homog_decomp`: If the field is homogeneous, enter `SAF__SELF` here. Otherwise, the field is inhomogeneous and this argument must indicate a decomposing collection of the field's base-space upon which it is *presumably* homogeneous. We say *presumably* because it is not a **requirement** that the field be homogeneous on each of the members of the collection identified here. The field pieces defined on any one or all of those members can, in turn, also be inhomogeneous. The only requirement is that the collection identified here be a decomposition of the associated set and that, ultimately, the recursion of defining inhomogeneous fields in terms of other inhomogeneous fields terminates on a bunch of homogeneous pieces. A common use of this argument is to indicate that the field is broken into independent chunks of storage (either within a single processor or distributed across other processors). In fact, prior to [SAF-1.2.1](#), that was all this argument was used for and documented as supporting. Any collections contained in the base space set for which the `IS_DECOMP` argument in the [saf_declare_collection](#) call was `SAF__DECOMP_TRUE`, can be passed here. See the chapter introduction for fields for further information (see **Fields**).
- `coeff_assoc`: This argument identifies the category of a collection in the base space set which the field's coefficients are `n:1` associated with. For example, for a field whose coefficients are `1:1` with a collection of a category representing the nodes, you would identify that collection category with this argument. Likewise, for a field whose coefficients are `4:1` with a collection of a category representing the elements in the problem, you would identify that collection with this argument. Note, if the coefficients are associated with the base space itself, and not the members of a collection in the base-space set, you would pass `SAF__SELF` for this argument.
- `assoc_ratio`: This argument specifies the *n* in the `n:1` association described above. For example, if for every member of the collection representing the elements, you have 1 coefficient, then this value would be 1. This value is always non-negative.
- `eval_cat`: This argument specifies the collection whose members represent the pieces in the piecewise evaluation of the field. If there is only a single piece (e.g. the whole base space), then pass `SAF__SELF`. For example, a collection category identifying the nodes for the `coeff_assoc` argument and an `assoc_ratio` of 1 indicates only that we have 1 coefficient for each member of the collection of nodes. It does not indicate which collection in the base space (for example the elements), the field is actually piecewise evaluated on.
- `eval_func`: This argument identifies one of several evaluation functions currently known to [SAF](#). Again, [SAF](#) does not yet actually evaluate a field. It only stores the descriptive information to support its evaluation. See definition of `SAF__EvalFunc` enum for the possible values. Also, we have provided some convenience macros for this and `coeff_assoc`, `assoc_ratio`, and `eval_cat` arguments for common cases; *node centered* and *zone centered* fields. Pass `SAF__NODAL` for a node centered field, `SAF__ZONAL` for a zone centered field, `SAF__DECOMP` for a field that is piecewise constant over some *decomposing* collection (e.g. domains) or `SAF__CONSTANT` for a constant field.
- `data_type`: The type of data in `bufs` if `bufs` are provided.
- `comp_flds`: Array of component field handles. Pass null only if there are no components to this field (the field is a scalar field).
- `comp_intlv`: The particular fashion in which components are interleaved. Currently there are really only two: `SAF__INTERLEAVE_VECTOR` and `SAF__INTERLEAVE_COMPONENT`. These represent the `XYZXYZ...` "XYZ" and the `XXX... "XYYY"...` "YZZZ"..."Z" cases. Note that `interleave` really only deals within a single blob of storage. In the case of a composite field whose coefficients are stored independently on the component fields, `interleave` really has no meaning (use `SAF__INTERLEAVE_INDEPENDENT`). `Interleave` only has meaning on fields with storage. In the case of a scalar field `interleave` is also meaningless, both cases degenerate to the same layout: `XXX... "X"` (use `SAF__INTERLEAVE_NONE`).
- `comp_order`: Only relevant for fields with component fields. This value indicates the order of the fields in the `comp_flds` relative to the registered order. Pass `NULL` if the permutation is the identity.
- `bufs`: The field data buffers. Pass `NULL` if you would rather provide this on the write call. Note that the number and size of buffers (if any) is specified by the `interleave` and number of components. If the field has vector `interleave` then there may only be 1 buffer, if the field has component `interleave` then there must be `num_components` buffers. The number of components is defined in the field template specified by `ftmpl`.

- `fld`: [OUT] The optional returned field handle. If `NULL` is passed here then this function allocates the field handle before returning it.

Description: This function is used to declare a field. A field is some physical quantity known to have *value* at every point in the infinite set of points that serves as the field's base space. That is, a field is some continuous (as opposed to discrete) quantity that exists and has value *everywhere* over the base space the field is defined on - that is, at every point in the infinite set of points that is the field's base space.

We apologize for the large number of arguments in this function call. We have developed prototype interfaces that reduce this complexity significantly but introduce other issues. As more experience is gained with this software and data model, we'll have a better idea how to proceed.

In [SAF](#), the description of a field is done in two parts; a field template (see [saf_declare_field_tmpl](#)) and an instance of a field. The field template object describes all the abstract information about a field. The field object itself describes the implementation details of an instance of a field. For example, the field template object describes the abstract quantity of measure the field represents, such as length (see **Quantities**) while the field object describes specific units of measure for that quantity such as meters (see **Units**).

You will notice that the base space upon which the field is defined is **not** part of the field object. Instead it is part of the field template object. This allows the field template object to classify fields according to which pieces of the base space they are defined on.

In the hierarchy of sets that serve as candidate base spaces for fields, the idea is to declare a field on the top-most set in the hierarchy which contains all points the field is defined on but contains no points the field is **not** defined on. Such a set is also called the *maximal* set of the field. It could also be thought of as the region of support of the field.

There is a **big** difference between *declaring* a field that is identically zero over portions of a set and *declaring* the field only over the subset(s) for which it is non-zero. The former indicates that the field is known everywhere on the set and is zero in some places. The latter indicates that the field is known on the subset(s) and *undefined* (e.g. does not exist) anywhere else.

At present, [SAF](#) really does not do much to interpret the data or descriptive information for a field. Currently, [SAF](#) simply allows a writer client to describe the salient features of a field and a reader client to discover them. As [SAF](#) evolves, [SAF](#) will be able to interpret more and more about the field itself.

Preconditions:

- `pmode` must be valid. (low-cost)
- `base_space` must be a valid set handle. (low-cost)
- `name` must be non-null. (low-cost)
- `ftmpl` must be a valid field template. (low-cost)
- `STORAGE_DECOMP` must be either `SELF_DECOMP` or a valid cat handle. (low-cost)
- `coeff_assoc` must be a valid cat handle. (low-cost)
- `eval_cat` must be a valid cat handle. (low-cost)
- `unit` must be a valid unit handle if supplied. (low-cost)
- `unit` must agree with quantity defined on field template. (low-cost)
- `assoc_ratio` must be non-negative. (low-cost)
- `eval_func` must be a valid evaluation type handle if supplied. (low-cost)
- Units of component fields must match units of composite field. (low-cost)

Return Value: Returns a handle to the set on success (either the one passed in by the `fld` argument or one allocated herein); returns `NULL` on failure.

See Also:

- *saf_declare_collection*: 11.1: *Declare a collection*
- *saf_declare_field_tmpl*: 15.2: *Declare a field template*
- *Fields*: Introduction for current chapter

Get a description of a field

`saf_describe_field` is a function defined in `field.c`.

Synopsis:

```
int saf_describe_field(SAF_ParMode pmode, SAF_Field *field, SAF_FieldTmpl *ftmpl,
                      char **name, SAF_Set *base_space, SAF_Unit *unit, hbool_t *is_coord,
                      SAF_Cat *homog_decomp, SAF_Cat *coeff_assoc, int *assoc_ratio,
                      SAF_Cat *eval_coll, SAF_Eval *eval_func, hid_t *data_type, int *num_comps,
                      SAF_Field **comp_fds, SAF_Interleave *comp_intlv, int **comp_order)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field handle.
- `ftmpl`: [OUT] The returned field template handle. Pass NULL if you do not want this value returned.
- `name`: [OUT] The returned name of the field. Pass NULL if you do not want this value returned. (see **Returned Strings**).
- `base_space`: [OUT] The returned base space of the field. Pass NULL if you do not want this value returned.
- `unit`: [OUT] The returned unit of measure.
- `is_coord`: [OUT] A returned boolean indicating if the field is a coordinate field. Pass NULL if you do not want this value returned.
- `homog_decomp`: NULL: If the field is homogeneous, the value returned here, if requested, is always `SAF__SELF`. That is, `SAF__EQUIV``(``SAF__SELF``(db), ``homog_decomp)` will return true. Otherwise, it will return false, the field is inhomogeneous and this argument is the decomposition on which the field is presumably piecewise homogeneous. Pass NULL if you do not want this value returned.
- `coeff_assoc`: [OUT] The collection with which the field coefficients are associated in an $n:1$ relationship. Pass NULL if you do not want this value returned.
- `assoc_ratio`: [OUT] The n in the ' $n:1$ ' relationship described for the `coeff_assoc` argument. Pass NULL if you do not want this value returned.
- `eval_coll`: [OUT] The collection whose sets decompose the base space set and over which the field is actually evaluated. Pass NULL if you do not want this value returned.
- `eval_func`: [OUT] The evaluation function. Pass NULL if you do not want this value returned.
- `data_type`: [OUT] The file datatype of the field. Pass NULL if you do not want this value returned. The caller is responsible for invoking `H5Tclose` when the datatype is no longer needed. A negative returned value indicates no known file datatype.
- `num_comps`: [OUT] The number of components in the field. Pass NULL if you do not want this value returned.
- `comp_fds`: [OUT] The component fields. Pass NULL if you do not want this value returned.
- `comp_intlv`: [OUT] The particular fashion in which components are interleaved. Currently there are really only two: `SAF__INTERLEAVE_VECTOR` and `SAF__INTERLEAVE_COMPONENT`. These represent the `XYZXYZ...` '`XYZ`' and the `XXX...` '`YYYY`'... '`YYYY`'... '`Z`' cases. Note that `interleave` really only deals within a single blob of storage. In the case of a composite field whose coefficients are stored independently

on the component fields then interleave really has no meaning (use `SAF__INTERLEAVE_INDEPENDENT`). Interleave only has meaning on fields with storage. In the case of a scalar field interleave is also meaningless, both cases degenerate to the same layout: `XXX...“X“` (use `SAF__INTERLEAVE_NONE`).

- `comp_order`: [OUT] The component ordering in the field. Pass `NULL` if you do not want this value returned.

Description: NOT WRITTEN YET.

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle for all participating processes. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Fields](#): Introduction for current chapter

Find coordinate fields

`saf_find_coords` is a function defined in `field.c`.

Synopsis:

```
int saf_find_coords (SAF_ParMode  pmode,   SAF_Db   *db,   SAF_Set   *base,   int   *num,
                    SAF_Field **found)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: Database in which to limit the search.
- `base`: The base space for which coordinate fields are desired.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: Use this function to find the coordinate fields of a set. In general, we allow for more than one coordinate field to be defined. For example, in engineering codes, there are the deformed and undeformed coordinates. Thus, this function can return multiple fields. Even so, there is only ever one field known as the **default** coordinate field for a set. This field is found with a call to `saf_find_default_coords`.

Preconditions:

- `pmode` must be valid. (low-cost)
- `base` must be either a valid set handle or the universe set for all participating processes. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- `saf_find_default_coords`: 16.15: *Find default coordinate fields*
- [Fields](#): Introduction for current chapter

Find default coordinate fields

`saf_find_default_coords` is a function defined in `field.c`.

Synopsis:

`SAF_Field * saf_find_default_coords` (`SAF_ParMode pmode`, `SAF_Set *base`, `SAF_Field *field`)

Formal Arguments:

- `pmode`: The parallel mode
- `base`: The set for which the default coordinate field is returned
- `field`: [OUT] The returned field handle, if found, otherwise `SAF__NOT_SET_FIELD`

Description: Use this function to find the default coordinate fields of a set. There is only ever one default coordinate field for a set.

Preconditions:

- `pmode` must be valid. (low-cost)
- `base` must be a valid set handle for participating processes. (low-cost)

Return Value: On success, returns a pointer to the default coordinate field. If the `field` argument was supplied then it is filled in and becomes the return value, otherwise a new field link is allocated and returned. Returns `NULL` on failure. If no default coordinate field has been assigned to the `base` set then a valid object link is returned but that link is nil (i.e., a call to `:file:~SS_PERS_ISNULL ../sslib_refman.rest/SS_PERS_ISNULL.rst` on the return value is true but the return value is not a `NULL` C pointer).

See Also:

- [Fields](#): Introduction for current chapter

Find fields

`saf_find_fields` is a function defined in `field.c`.

Synopsis:

`int saf_find_fields` (`SAF_ParMode pmode`, `SAF_Db *db`, `SAF_Set *base`, `const char *name`,
`SAF_Quantity *quantity`, `SAF_Algebraic *atype`, `SAF_Basis *basis`,
`SAF_Unit *unit`, `SAF_Cat *coeff_assoc`, `int assoc_ratio`, `SAF_Cat *eval_decomp`,
`SAF_Eval *eval_func`, `int *nfound`, `SAF_Field **found`)

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: Database in which to limit the search.
- `base`: The base space to limit the search to. Pass `SAF__UNIVERSE` or `NULL` if you do not want to limit the search to any particular base space.
- `name`: Limit search to fields with this name. Pass `SAF__ANY_NAME` if you do not want to limit the search.
- `quantity`: Limit search to fields of specified quantity. Pass `NULL` to not limit search.
- `atype`: Limit the search to this algebraic type. Pass `SAF__ALGTYPE_ANY` if you do not want to limit the search.
- `basis`: Limit the search to this basis. Pass `SAF__ANY_BASIS` if you do not want to limit the search.
- `unit`: Limit search to fields with these units. Pass `SAF__ANY_UNIT` to not limit search.

- `coeff_assoc`: Limit search. Pass `SAF__ANY_CAT` to not limit the search.
- `assoc_ratio`: Limit search. Pass `SAF__ANY_RATIO` to not limit the search.
- `eval_decomp`: Limit search. Pass `SAF__ANY_CAT` to not limit the search.
- `eval_func`: Limit search. Pass `SAF__ANY_EFUNC` to not limit the search.
- `nfound`: For this and the succeeding argument, (see **Returned Handles**).
- `found`: For this and the preceding argument, (see **Returned Handles**).

Description: This function allows a client to search for fields in the database. The search may be limited by one or more criteria such as the name of the field, the quantity the field represents, the base space the field is defined on, etc., etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `base` must either be a valid set handle or the universe set if supplied. (low-cost)
- `quantity` must either be a valid quantity handle or `SAF__ANY_QUANTITY`. (low-cost)
- `atype` must be a valid algebraic type handle or `SAF__ANY_ALGEBRAIC`. (low-cost)
- `basis` must be a valid basis handle or `SAF__ANY_BASIS`. (low-cost)
- `unit` must either be a valid unit handle or `SAF__ANY_UNIT`. (low-cost)
- `eval_func` must be a valid evaluation function handle or `SAF__ANY_EVALUATION`. (low-cost)
- `coeff_assoc` must either be a valid cat handle or `SAF__ANY_CAT`. (low-cost)
- `eval_decomp` must either be a valid cat handle or `SAF__ANY_CAT`. (low-cost)
- `nfound` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: Should [SAF](#) traverse up the `SIL` to find all fields that are actually defined for the given set?

See Also:

- [Fields](#): Introduction for current chapter

Get datatype and size for a field

`saf_get_count_and_type_for_field` is a function defined in `field.c`.

Synopsis:

```
int saf_get_count_and_type_for_field (SAF_ParMode    pmode,      SAF_Field    *field,
                                     SAF_FieldTarget *target, size_t *Pcount, hid_t *Ptype)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field handle.
- `target`: Optional field targeting information.
- `Pcount`: [OUT] The number of items that would be placed in the buffer by a call to the `saf_read_field` function. The caller may pass a value of `NULL` for this parameter if this value is not desired.

- `p_type`: [OUT] The type of the items that would be placed in the buffer by a call to the [saf_read_field](#) function. The caller may pass a value of `NULL` for this parameter if this value is not desired. The returned HDF5 datatype can be closed by the caller when no longer needed.

Description: This function is used to retrieve the number and type of items that would be retrieved by a call to the [saf_read_field](#) function. This function may be used by the caller to determine the size of the buffer needed when pre-allocation is desired or to determine how to traverse the buffer returned by the [saf_read_field](#) function.

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle. (low-cost)
- If targeting of storage decomposition is used, the read must be a `SAF__ALL` mode read or the. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: Fields stored on a decomposition must have same datatype. It may be possible to relax this a bit. Also what if the field has been decomposed into blocks? say triangles and quads, field remapping may be possible but makes no sense as the DOFs would be all mixed-up some for triangles, some for quads.

See Also:

- [saf_read_field](#): 16.21: *Read the data for a field*
- [Fields](#): Introduction for current chapter

Get an attribute from a field

`saf_get_field_att` is a function defined in `field.c`.

Synopsis:

```
int saf_get_field_att (SAF_ParMode pmode, SAF_Field *fld, const char *name, hid_t *type, int *count, void **value)
```

Description: This function is identical to the generic [saf_get_attribute](#) function except that it is specific to `SAF__Field` objects to provide the client with compile time type checking. For a description, see [saf_get_attribute](#).

See Also:

- [saf_get_attribute](#): 23.1: *Read a non-sharable attribute*
- [Fields](#): Introduction for current chapter

Is field stored on self

`saf_is_self_stored_field` is a function defined in `field.c`.

Synopsis:

```
int saf_is_self_stored_field (SAF_ParMode pmode, SAF_Field *field, hbool_t *result)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The handle of the field which is to be examined.
- `result`: [OUT] Optional pointer to memory which is to receive the result of the test: true if the field is self stored or false if it is stored on a decomposition.

Description: This function is used by a client to test if a field is stored on self or on a decomposition. The boolean result is returned by reference.

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Fields](#): Introduction for current chapter

Put an attribute with a field

`saf_put_field_att` is a function defined in `field.c`.

Synopsis:

```
int saf_put_field_att (SAF_ParMode pmode, SAF_Field *field, const char *name, hid_t type, int count,
                     const void *value)
```

Description: This function is identical to the generic [saf_put_attribute](#) function except that it is specific to `SAF__Field` objects to provide the client with compile time type checking. For a description, see [saf_put_attribute](#).

See Also:

- [saf_put_attribute](#): 23.2: Create or update a non-sharable attribute
- [Fields](#): Introduction for current chapter

Read the data for a field

`saf_read_field` is a function defined in `field.c`.

Synopsis:

```
int saf_read_field (SAF_ParMode pmode, SAF_Field *field, SAF_FieldTarget *target, int member_count,
                  SAF_RelRep *req_type, int *member_ids, void **Pbuf)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field which is to be read.
- `target`: Field targeting information.
- `member_count`: A count of the number of members of the collection in which the field's dofs are `n:1` associated with that are actually being written in this call. This value is ignored if you are reading the entire field's dofs in this call (i.e., `req_type = SAF__TOTALITY`). Also note that as a convenience, we provide the macro `SAF__WHOLE_FIELD` which expands to a comma separated list of appropriate values for this argument and the next two, for the case in which the whole field is being read in this call.
- `req_type`: The type of I/O request. We use a relation representation type here to specify the type of the partial request because it captures the necessary information. Pass `SAF__HSLAB` if you are reading the dofs of a partial hyperslab of the members of the associated collection. In this case, `member_ids` points to 3 N-tuples of starts, counts and strides of the hyperslab (hypersample) request. Pass `SAF__TUPLES`, if you are reading the dofs for an arbitrary list of members of the associated collection. In this case, the `member_ids` points to

a list of N-tuples. In both cases, 'N' is the number of indexing dimensions in the associated collection. Finally, pass `SAF__TOTALITY` if you are reading the entire field's set of dofs.

- `member_ids`: Depending on the value of `req_type`, this argument points to 3 N-tuples storing, respectively, the starts, counts and strides **in each dimension** of the associated collection or to a list of `member_count` N-tuples, each one identifying a single member of the associated collection or to `NULL` in the case of a `SAF__TOTALITY` request.
- `Pbuf`: `[IN ``| ``OUT]` A pointer to a buffer pointer which is to receive the values read. The caller may supply a pointer to a value of `NULL` if this function is to allocate a buffer. If the caller supplies a pointer to a non-`NULL` pointer (to a buffer) then it is up to the caller to ensure that the buffer is of sufficient size to hold all of the data retrieved. The caller should use *`saf_describe_field`* or *`saf_get_count_and_type_for_field`* to determine the datatype of the values read.

Description: This function is used to read a field's data. If the field is **not** an indirect reference to other fields, this call involves **real** disk I/O. All functions in *SAF* with either "read" or "write" in the name potentially involve real disk I/O.

This function allows a client to read either the entire field's data or a portion of the field's data. Recall that the *degrees of freedom* (dofs) of a field are `n : 1` associated with the members of some collection in the set upon which the field is defined. We call this collection the *associated collection*.

In order to specify a partial request, the client is required to specify which members of the associated collection it is reading the dofs for. Ultimately, those members may be specified using a N dimensional hyperslab (or hypersample) or an arbitrary list of N-tuples. In either case, the number of dimensions, 'N', is the number of indexing dimensions in the associated collection.

At present, there are several limitations. First, the collection must be 1 dimensionally indexed. Next, only the hyperslab mode or a single member in tuple-mode are supported; not hypersamples and not an arbitrary list. Also, if the field is a multi-component field, then the only supported interleave mode is `SAF__INTERLEAVE_VECTOR`.

Finally, we provide as a convenience the macro `SAF__WHOLE_FIELD` which expands to a comma separated list of values, 0, `SAF__TOTALITY`, `NULL`, for the three arguments `member_count`, `req_type`, `member_ids` for the case in which the client is reading the whole field in this call.

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle. (low-cost)
- `Pbuf` must be non-null. (low-cost)
- If partial I/O request, associated collection must be 1D indexed, `req_type` must be `SAF__HSLAB`. (high-cost)
- If field targeting of storage decomposition is used, the read must be a `SAF__ALL` mode read or the. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: A partial I/O request looks a lot like a subset relation. In fact, we even use the same data type, `SAF__SRTYPE`, to identify the type of partial I/O request. It may be difficult for a client to distinguish between making a partial I/O request and making real subsets. In theory, there **really** should be no difference. The act of reading/writing a portion of a field **is** the act of defining a subset of the base space the field is defined on and then restricting the field to that subset. In the current implementation, this requires, at a minimum, the ability to create transient objects such as the subset representing the piece of the field being read/written in this call. In addition, it **really** requires decoupling the storage containers into which field's data goes from declaring and reading/writing fields.

For an indirect field the the local fields are all "similar". That is, they have the same algebraic type, association category, units and such. This function should check for this but doesn't. In the future some differences can be smoothed-over (such as units) but some probably can not (such as algebraic type).

The proper use of `pmode` is not fully worked out.

Multiple indirection may actually fall out of this solution but that is not at all clear.

When remapping an indirect field we only look in the top-scope of the file containing the field's base space when searching for the subset relations. [rpm 2004-05-24]

See Also:

- [*saf_describe_field*](#): 16.13: *Get a description of a field*
- [*saf_get_count_and_type_for_field*](#): 16.17: *Get datatype and size for a field*
- [*Fields*](#): Introduction for current chapter

Set the destination form of a field

`saf_target_field` is a function defined in `field.c`.

Synopsis:

```
int saf_target_field (SAF_FieldTarget      *target,          SAF_Unit      *targ_units,
                    SAF_Cat      *targ_storage_decomp,    SAF_Cat      *targ_coeff_assoc,
                    int  targ_assoc_ratio,  SAF_Cat  *targ_eval_coll,  SAF_Eval  *targ_func,
                    hid_t targ_data_type, SAF_Interleave comp_intlv, int *comp_order)
```

Formal Arguments:

- `target`: [OUT] The target information that will be initialized by this call.
- `targ_units`: The new units. This parameter is ignored at this time.
- `targ_storage_decomp`: The new storage decomposition.
- `targ_coeff_assoc`: This parameter is ignored at this time.
- `targ_assoc_ratio`: This parameter is ignored at this time.
- `targ_eval_coll`: This parameter is ignored at this time.
- `targ_func`: This parameter is ignored at this time.
- `targ_data_type`: The new destination data type. When the [*saf_write_field*](#) function is called the datatype of the dataset produced is determined by this parameter. When the [*saf_read_field*](#) function is called, the datatype of the values placed in the caller's memory is determined by this parameter. If a value of `H5I_INVALID_HID` is passed for this parameter then datatype targeting is turned off and the default mechanism for determining the destination datatype is used.
- `comp_intlv`: The particular fashion in which components are interleaved. Currently there are really only two: `SAF__INTERLEAVE_VECTOR` and `SAF__INTERLEAVE_COMPONENT`. These represent the `XYZXYZ...` "XYZ" and the `XXX... "XYYY"... "YZZZ"... "Z"` cases. Note that `interleave` really only deals with a single blob of storage. In the case of a composite field whose coefficients are stored independently on the component fields then `interleave` really has no meaning (use `SAF__INTERLEAVE_INDEPENDENT`). `Interleave` only has meaning on fields with storage. In the case of a scalar field `interleave` is also meaningless, both cases degenerate to the same layout: `XXX... "X"` (use `SAF__INTERLEAVE_NONE`). This parameter is ignored at this time.
- `comp_order`: Only relevant for fields with component fields. This value indicates the order of the field IDs in the `COMP_FLDS` relative to the registered order. Pass `NULL` if the permutation is the identity. This parameter is ignored at this time.

Description: Setup targeting information for a field during read or write. Please see the introductory note in the Field's chapter for some information on field targeting.

Preconditions:

- `STORAGE_DECOMP` must be either `NOT_SET`, `SELF_DECOMP` or a valid cat handle. (low-cost)
- `target` must be non-null. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [*saf_read_field*](#): 16.21: *Read the data for a field*
- [*saf_write_field*](#): 16.23: *Write the data for a field*
- [*Fields*](#): Introduction for current chapter

Write the data for a field

`saf_write_field` is a function defined in `field.c`.

Synopsis:

```
int saf_write_field (SAF_ParMode  pmode,    SAF_Field  *field,    int  member_count,
                    SAF_RelRep *req_type, int *member_ids, int nbufs, hid_t buf_type, void **bufs,
                    SAF_Db *file)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `field`: The field to write.
- `member_count`: A count of the number of members of the collection in which the field's dofs are `n:1` associated that are actually being written in this call. This value is ignored if you are writing the entire field's dofs in this call (i.e., `req_type` is `SAF__TOTALITY`). Also note that as a convenience, we provide the macro `SAF__WHOLE_FIELD` which expands to a comma separated list of appropriate values for this argument and the next two, for the case in which the whole field is being written in this call.
- `req_type`: The type of I/O request. We use a relation representation type here to specify the type of the partial request because it captures the necessary information. Pass `SAF__HSLAB` if you are writing the dofs of a partial hyperslab of the members of the associated collection. In this case, `member_ids` points to 3 N-tuples of starts, counts and strides of the hyperslab (hypersample) request. Pass `SAF__TUPLES`, if you are writing the dofs for an arbitrary list of members of the associated collection. In this case, the `member_ids` points to a list of N-tuples. In both cases, 'N' is the number of indexing dimensions in the associated collection. Finally, pass `SAF__TOTALITY` if you are writing the entire field's set of dofs.
- `member_ids`: Depending on the value of `req_type`, this argument points to 3 N-tuples storing, respectively, the starts, counts and strides **in each dimension** of the associated collection or to a list of `member_count` N-tuples, each one identifying a single member of the associated collection or to `NULL` in the case of a `SAF__TOTALITY` request.
- `nbufs`: The number of buffers. Valid values are either 1 or a value equal to the number of components of the field. A value greater than 1 indicates that the field is stored component by component, one buffer for each component. Note, however, that current limitations of partial requests support only fields that are interleaved by `SAF__INTERLEAVE_VECTOR`. This, in turn, means that in a partial I/O request, `nbufs` can only ever be one.
- `buf_type`: The type of the objects in the buffer(s). If the buffer datatype was provided in the [*saf_declare_field*](#) call that produced the field handle then this parameter should have a negative value. If however the datatype was

not provided in the *saf_declare_field* or if the handle was the result of a find operation then the datatype must be provided in this call.

- `bufs`: The buffers.
- `file`: Optional file into which the data is written. If none is supplied then the data is written to the same file as the `field`.

Description: This function is used to write a field's data. If the field is **not** an indirect reference to other fields, this call involves **real** disk I/O. All functions in **SAF** with either "read" or "write" in the name potentially involve real disk I/O.

This function allows a client to write either the entire field's data or a portion of the field's data. Recall that the *degrees of freedom* (dofs) of a field are $n:1$ associated with the members of some collection in the set upon which the field is defined. We call this collection the *associated collection*.

In order to specify a partial request, the client is required to specify which members of the associated collection it is writing the dofs for. Ultimately, those members may be specified using a N dimensional hyperslab (or hypersample) or an arbitrary list of N -tuples. In either case, the number of dimensions, ' N ', is the number of indexing dimensions in the associated collection.

At present, there are several limitations. First, the collection must be 1 dimensionally indexed. Next, only the hyperslab mode or a single member in tuple-mode are supported, not hypersamples and not an arbitrary list. Finally, if the field is a multi-component field, then the only supported interleave mode is `SAF__INTERLEAVE_VECTOR`.

For indirect fields, the notion of writing on the composite or component fields is lost. On an indirect, composite field, the values written must be handles to other composite fields. Likewise for its component fields. The values written must be handles for other component fields.

Finally, we provide as a convenience the macro `SAF__WHOLE_FIELD` which expands to a comma separated list of values, 0, `SAF__TOTALITY`, `NULL`, for the three arguments `member_count`, `req_type`, `member_ids` for the case in which the client is writing the whole field in this call.

Preconditions:

- `pmode` must be valid. (low-cost)
- `field` must be a valid field handle. (low-cost)
- `bufs` must be specified here or in the *saf_declare_field* call (not both). (low-cost)
- Pass either valid `bufs` and `nbufs`>`>0 or ``NULL and "nbufs"==0. (low-cost)`
- If partial I/O request, collection must be 1D indexed, `req_type` must be `SAF__HSLAB`. (high-cost)
- Buffer datatype must be specified in field declaration or write. (low-cost)
- Buffer datatype must be consistent between field declaration and write. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: `SAF__EACH` mode is a collective call where each of the N tasks provides a unique relation. **SAF** will create a single HDF5 dataset to hold all the data and will create N blobs to point into nonoverlapping regions in that dataset. In `SAF__EACH` mode the call must still be collective across the `file` communicator (or the communicator of the dataset to which `field` belongs if `file` is null). This requirement is due to the fact that an HDF5 dataset may need to be created and such an operation is collective.

Issues: A partial I/O request looks a lot like a subset relation. In fact, we even use the same data type, `SAF__SRTYPE`, to identify the type of partial I/O request. It may be difficult for a client to distinguish between making a partial I/O request and making real subsets. In theory, there **really** should be no difference. The act of writing a portion of a field **is** the act of defining a subset of the base space the field is defined on and then restricting the field to that subset. In the current implementation, this requires, at a minimum, the ability to create transient objects such as the subset

representing the piece of the field being written in this call. In addition, it **really** requires decoupling the storage containers into which field's data goes from declaring and writing fields.

For a compound data-type on a composite field, we probably ought to confirm a) the rank of the compound type is equal to the number of components, b) the type of each member of the compound type is equal to the type of each of the component fields (assuming both are ordered the same), and c) the names of the member types are the same as the component fields. Currently we are only checking a).

Computing the actual size of the I/O request here is NO SMALL TASK. It depends on a combination of factors including the number of buffers, the number of members whose dofs are being written, the association ratio, the data-type and whether the field is direct or indirect.

Is it possible that a `SAF__EACH` call will have a different offset and data for each task? If so we'll have to do some communicating first otherwise :file:`ss_file_synchronize ../sslib_refman.rest/ss_file_synchronize.rst` will see that each task made incompatible modifications to this object. This code just checks that for now. [rpm 2004-06-07]

See Also:

- *saf_declare_field*: 16.12: *Declare a field*
- *Fields*: Introduction for current chapter

State Templates

A state template is a pattern for what types of fields can be grouped into a state. This pattern is specified by a list of field templates.

Members

The null state template handle

`SAF_NULL_STMPL` is a macro defined in `saf.h`.

Synopsis:

`SAF_NULL_STMPL` (Db)

Description: This macro evaluates to the state template handle for the null state template of the database. The null set handle is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See **Constants**).

See Also:

- *State Templates*: Introduction for current chapter

Declare a state template

`saf_declare_state_tmpl` is a function defined in `stempl.c`.

Synopsis:

```
SAF_StateTmpl * saf_declare_state_tmpl (SAF_ParMode pmode, SAF_Db *database, const
                                         char *name, int num_ftmpls, SAF_FieldTmpl *ftmpls,
                                         SAF_StateTmpl *stmpl)
```

Formal Arguments:

- `pmode`: The parallel mode.

- `name`: The name of the state template.
- `num_ftmpls`: Number of field templates that will comprise this state template.
- `ftmpls`: Array of field template handles.
- `stmpl`: The returned state template handle.

Description: This creates a state template associated with a specified suite.

Return Value: Returns a pointer to the new state template on success; null on failure. If the caller supplies a non-null `stmpl` argument then this pointer will be the return value, otherwise a state template link will be allocated.

See Also:

- [State Templates](#): Introduction for current chapter

Get a description of a state template

`saf_describe_state_tmpl` is a function defined in `stempl.c`.

Synopsis:

```
int saf_describe_state_tmpl (SAF_ParMode  pmode,  SAF_StateTmpl  *stmpl,  char  **name,
                           int *num_ftmpls, SAF_FieldTmpl **ftmpls)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `stmpl`: The state template handle.
- `name`: [OUT] The returned name. Pass NULL if you do not want the name returned.
- `num_ftmpls`: [OUT] The returned number of field templates which comprise this state template.
- `ftmpls`: [OUT] The returned field templates.

Description: Returns a description of a state template.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [State Templates](#): Introduction for current chapter

Find a state template

`saf_find_state_tmpl` is a function defined in `stempl.c`.

Synopsis:

```
int saf_find_state_tmpl (SAF_ParMode  pmode,  SAF_Db  *database,  const  char  *name,
                       int *num_stmpls, SAF_StateTmpl **stmpls)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `database`: the database context for this search
- `name`: The name of the state template you are searching for. Pass `SAF__ANY_NAME` if you do not wish to limit your search to just this name.

- `num_stmpls`: For this and the succeeding argument [see Returned Handles].
- `stmpls`: For this and the preceding argument [see Returned Handles].

Description: Find state templates in a suite.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *State Templates*: Introduction for current chapter

Get an attribute attached to a state template

`saf_get_state_tmpl_att` is a function defined in `stempl.c`.

Synopsis:

```
int saf_get_state_tmpl_att (SAF_ParMode pmode, SAF_StateTmpl *stmpl, const char *att_key,
                           hid_t *att_type, int *count, void **value)
```

Description: This function is identical to the generic *[saf_get_attribute](#)* function except that it is specific to `SAF__StateTmpl` objects to provide the client with compile time type checking. For a description, see *[saf_get_attribute](#)*.

See Also:

- *[saf_get_attribute](#)*: 23.1: Read a non-sharable attribute
- *State Templates*: Introduction for current chapter

Attach an attribute to a state template

`saf_put_state_tmpl_att` is a function defined in `stempl.c`.

Synopsis:

```
int saf_put_state_tmpl_att (SAF_ParMode pmode, SAF_StateTmpl *stmpl, const char *att_key,
                           hid_t att_type, int count, const void *value)
```

Description: This function is identical to the generic *[saf_put_attribute](#)* function except that it is specific to `SAF__StateTmpl` objects to provide the client with compile time type checking. For a description, see *[saf_put_attribute](#)*.

See Also:

- *[saf_put_attribute](#)*: 23.2: Create or update a non-sharable attribute
- *State Templates*: Introduction for current chapter

States

A state is a “slice” through a suite at a fixed parameter value (typically time). For example, a state contains all the following that is associated with a specific time step of a simulation:

- pointer to the computational mesh (i.e., a set);
- pointer to the default coordinate field (an independent variable) of the mesh;
- the time value (also an independent variable) of the state;

- pointers to all the fields (the dependent variables) attached to the mesh at the specific time step.

What if the desired output changes from state to state?. For example, suppose a client writes various combinations of Coordinates (C), Pressure (P), Temperature (T), Velocity (V) and Stress (S) fields according to the following sequence...

1								C	C		
2			C	C	C	C		V	V		
3	C	C	V	V	V	V	C	C	T	T	
4	V	V	T	T	T	T	V	V	P	P	
5	S	S	P	P	P	P	S	S	S	S	
6	+---+---+---+---+---+---+---+---+---+										
7	0	1	2	3	4	5	6	7	8	9	<-- indices

1	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	<-- times
---	---	-----	---	-----	---	-----	---	-----	---	-----	-----------

The client should declare a state template that contains field templates for all the fields that will be referenced at any state. In the example above, the state template should consist of field templates for C, V, T, P, and S. For the states that don't contain all the fields, the client should pass a `SAF__NOT_APPLICABLE_FIELD` for those fields that aren't applicable for the state being written.

Members

The null state group handle

`SAF_NULL_STATE_GRP` is a macro defined in `saf.h`.

Synopsis:

`SAF_NULL_STATE_GRP` (Db)

Description: This macro evaluates to the state handle for the null state of the database. The null state handle is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See **Constants**).

See Also:

- States*: Introduction for current chapter

Declare a state group

`saf_declare_state_group` is a function defined in `state.c`.

Synopsis:

```
SAF_StateGrp * saf_declare_state_group (SAF_ParMode  pmode,   SAF_Db    *db,   const
                                       char *name, SAF_Suite *suite, SAF_Set  *mesh_space,
                                       SAF_StateTpl  *stmpl,   SAF_Quantity *quantity,
                                       SAF_Unit    *unit,   hid_t   coord_data_type,
                                       SAF_StateGrp *state_grp)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `db`: The database in which to declare the new state group.
- `name`: The name of this state group.

- `suite`: The suite that these states are associated with.
- `mesh_space`: The set representing the computational mesh
- `stmpl`: A state template that defines the pattern (via a list of field templates) of fields that can be stored in each state.
- `quantity`: The quantity associated with the axis of the parametric space. For example, `SAF__TIME_QUANTITY`.
- `unit`: The units associated with the axis of the parametric space.
- `coord_data_type`: The data type of the coordinates of the parametric space.
- `state_grp`: The returned handle for a state group.

Description: A state group contains all of the states associated with a suite. It contains:

```
1  - a name
2  - pointer to the suite that these states are attached to
3  - array of sets that represent the computational meshes associated with each state
4  - a coordinate field containing two components
5      -- a scalar field whose values are the parametric values (e.g., time values)
↪ associated with each state
6      -- a field whose values are IDs of the default coordinate fields (the
↪ independent variable) of the
7          computational mesh associated with each state
8  - a field containing IDs of all the fields (dependent variables) attached to the
↪ computational mesh at each state
```

Return Value: Returns a pointer to the new state group on success; null on failure. If the caller supplies a `state_grp` argument then that becomes the return value, otherwise a new state group link is allocated herein.

Issues: The new implementation of state group supercedes the current concept of a “state field”. A “state field”, as currently implemented, is just one component of a state group. Thus, we can delete all references to `SAF__StateFld` and add the new type `SAF__StateGrp` that contains:

```
1  - the name of the state group
2  - pointer to the SAF_Suite that the state group is attached to
3  - array of pointers to the SAF_Sets that represent the computational meshes
↪ associated with each state
4  - pointer to a SAF_Field coordinate field containing two components (this field may
↪ have to be an indirect field
5      if we don't support composite fields with heterogeneous components):
6          -- a scalar field whose values are the parametric values (e.g., time values)
↪ associated with each state
7          -- an indirect field whose values are IDs of the default coordinate fields of
↪ the computational mesh
8          associated with each state
9  - pointer to an indirect SAF_Field containing IDs of all the fields (dependent
↪ variables) attached to the
10 computational mesh at each state (this indirect field is what is currently a
↪ "state field")
```

See Also:

- [States](#): Introduction for current chapter

Get a description of a state group

`saf_describe_state_group` is a function defined in `state.c`.

Synopsis:

```
int saf_describe_state_group (SAF_ParMode pmode, SAF_StateGrp *state_grp, char **name,
                             SAF_Suite *suite, SAF_StateTpl *stmpl, SAF_Quantity *quantity,
                             SAF_Unit *unit, hid_t *coord_data_type, int *num_states)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `state_grp`: The state group to be described.
- `name`: [OUT] Returned name of the state group. Pass `NULL` if you do not want this value returned.
- `suite`: [OUT] Returned suite the state group is associated with.
- `stmpl`: [OUT] Returned state template. Pass `NULL` if you do not want this value returned.
- `quantity`: [OUT] The returned quantity associated with the axis of the parametric space. For example, `SAF__TIME_QUANTITY`.
- `unit`: [OUT] The returned units associated with the axis of the parametric space.
- `coord_data_type`: [OUT] The returned data type of the coordinates of the parametric space.
- `num_states`: [OUT] Returned number of states that have been written to this state group. Pass `NULL` if you do not want this value returned.

Description: Returns the description of a state group

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [States](#): Introduction for current chapter

Find state groups

`saf_find_state_groups` is a function defined in `state.c`.

Synopsis:

```
int saf_find_state_groups (SAF_ParMode pmode, SAF_Suite *suite, const char *name,
                           int *num_state_grps, SAF_StateGrp **state_grps)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `suite`: The suite within which to search.
- `name`: The name of the state group for which to search. Pass `SAF__ANY_NAME` if you do not want to limit your search.
- `num_state_grps`: [OUT] Returned number of state groups found.
- `state_grps`: [OUT] Returned state groups found.

Description: This finds the state groups that match specified criteria.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [States](#): Introduction for current chapter

Get an attribute attached to a state group

`saf_get_state_grp_att` is a function defined in `state.c`.

Synopsis:

```
int saf_get_state_grp_att (SAF_ParMode pmode, SAF_StateGrp *state_grp, const char *key,  
                           hid_t *type, int *count, void **value)
```

Description: This function is identical to the generic [saf_get_attribute](#) function except that it is specific to `SAF__StateGrp` objects to provide the client with compile time type checking. For a description, see [saf_get_attribute](#).

See Also:

- [saf_get_attribute](#): 23.1: Read a non-sharable attribute
- [States](#): Introduction for current chapter

Attach an attribute to a state group

`saf_put_state_grp_att` is a function defined in `state.c`.

Synopsis:

```
int saf_put_state_grp_att (SAF_ParMode pmode, SAF_StateGrp *state_grp, const char *key,  
                           hid_t type, int count, const void *value)
```

Description: This function is identical to the generic [saf_put_attribute](#) function except that it is specific to `SAF__StateGrp` objects to provide the client with compile time type checking. For a description, see [saf_put_attribute](#).

See Also:

- [saf_put_attribute](#): 23.2: Create or update a non-sharable attribute
- [States](#): Introduction for current chapter

Retrieve a state

`saf_read_state` is a function defined in `state.c`.

Synopsis:

```
int saf_read_state (SAF_ParMode pmode, SAF_StateGrp *state_grp, int state_index, SAF_Set *mesh,  
                   SAF_Field *deflt_coords, void *coord_data, SAF_Field **fields)
```

Formal Arguments:

- `pmode`: The parallel mode
- `state_grp`: The state group from which this state will be read.

- `state_index`: An index that specifies which state within the state group will be read. This index is 0-based.
- `mesh`: [OUT] Returned ID of the mesh associated with this state.
- `deflt_coords`: [OUT] Returned ID of the default coordinate field of `mesh`; we may want to delete this argument since the client can call *[saf_find_default_coords](#)* for `mesh`.
- `coord_data`: [OUT] Returned coordinate of `state_index` within the state group. For instance, this is typically the time value of the state.
- `fields`: The IDs of the fields (the dependent variables) to be read from this state. The caller may supply a pointer to a value of `NULL` if this function is to allocate a buffer. If the caller supplies a pointer to a non-nil pointer, then it is the responsibility of the caller to ensure that the buffer is of sufficient size to contain the coordinates. This size (`NUM_FIELDS`) is the number of field templates (`NUM_FTMPLS`) obtained by a call to *[saf_describe_state_tmpl](#)*.

Description: Read all the elements of a state. This includes the following:

- ID of the computational mesh (i.e., a set ID) associated with this state;
- ID of the default coordinate field of the mesh;
- the parametric value (e.g., the time value) associated with this state;
- IDs of all the fields (the dependent variables) attached to the mesh at this state.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *[saf_describe_state_tmpl](#)*: 17.3: *Get a description of a state template*
- *[saf_find_default_coords](#)*: 16.15: *Find default coordinate fields*
- *[States](#)*: Introduction for current chapter

Write out a state

`saf_write_state` is a function defined in `state.c`.

Synopsis:

```
int saf_write_state (SAF_ParMode  pmode,      SAF_StateGrp  *state_grp,  int  state_index,
                    SAF_Set      *mesh_space,  hid_t  coord_data_type,  void  *coord_data,
                    SAF_Field *fields)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `state_grp`: The state group into which this state will be inserted.
- `state_index`: The index within the state group at which this state will be written. This index is 0-based.
- `mesh_space`: The ID of the mesh associated with this state.
- `coord_data_type`: The data type of `COORD`
- `coord_data`: The coordinate of `state_index` within the state group. For instance, this is typically the time value of the state.
- `fields`: The fields (the dependent variables) to be written to this state.

Description: Write out all the elements of a state. This includes the following:

- ID of the computational mesh (i.e., a set ID) associated with this state;
- ID of the default coordinate field of the mesh;
- the parametric value (e.g., the time value) associated with this state;
- IDs of all the fields (the dependent variables) attached to the mesh at this state.

The state is referenced by an index which is an index into each of the arrays that compose the state group. See the description of a state group.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: This function does the following actions under the covers to implement the cross-product base space.

- increments the “`SAF__SPACE_SLICE`” collection
- adds a subset relation between the set identified by the `MESH` argument and the suite set

See Also:

- [States](#): Introduction for current chapter

Suites

A suite is a cartesian product of two base spaces (i.e., sets): one is the mesh from a simulation (typically a set with a `SIL` role of `SAF__SPACE`) and the other is a base space representing time or some other parametric space. Fields can be defined on a suite that are either slices through space (at constant times) or slices through time (at constant locations in space). The former are referred to as states; the latter are histories.

Members

The null suite handle

`SAF_NULL_SUITE` is a macro defined in `saf.h`.

Synopsis:

`SAF_NULL_SUITE` (Db)

Description: This macro evaluates to the suite handle for the null suite of the database. The null suite is most often only used in a `SAF__ONE` parallel call where many processors are participating solely for the sake of collectivity (See [*Constants*](#)).

See Also:

- [Suites](#): Introduction for current chapter

Declare a suite

`saf_declare_suite` is a function defined in `suite.c`.

Synopsis:

```
SAF_Suite * saf_declare_suite (SAF_ParMode pmode, SAF_Db *database, const char *name,  
                              SAF_Set *mesh_space, SAF_Set *param_space, SAF_Suite *suite)
```

Formal Arguments:

- `pmode`: The parallel mode.

- `database`: The SAF database handle.
- `name`: The name of the suite.
- `mesh_space`: The set representing the computational mesh. this is currently only a single set, so assume that the user cannot supply a list of `mesh_space` sets when declaring a suite
- `param_space`: The set representing the parametric space, such as time. If this is NULL, a set will be created with a SIL role of type TYPE.
- `suite`: [OUT] Optional memory for the returned handle. If null then a new handle is allocated by this function.

Description: This creates a suite with the given name.

Return Value: Returns a pointer to a new suite on success; null on failure. If the caller supplies a `suite` argument then this will be the pointer that is returned instead of allocating a new suite handle.

Issues: Currently, a `SAF__Suite` is `#typedef'd` to a `SAF__Set`. This may still work if we can identify suites with a new SIL role (“SAF__SUITE”).

The ability to declare “subsuites” may be needed but won’t be available in this implementation. This implementation will handle just 1-dimensional parametric spaces, such as time.

This function will create an extendible `SAF__Set` with two collections: one for associating states (fields across space at a fixed value of the specified parameter, usually time) and one for associating histories (fields across the specified parameter, usually time, at a fixed point in space). We will declare two new categories, “SAF__SPACE_SLICE” and “SAF__PARAM_SLICE”, respectively, that will be used in the declaration of these collections. These new categories will be a minor enhancement to the current *self* category.

If `param_space` is passed as NULL, this function will create a `SAF__Set` to represent the parametric space.

See Also:

- [Suites](#): Introduction for current chapter

Get a description of a suite

`saf_describe_suite` is a function defined in `suite.c`.

Synopsis:

```
int saf_describe_suite (SAF_ParMode pmode, SAF_Suite *suite, char **name, int *num_space_sets,
                      SAF_Set **mesh_space, SAF_Set **param_space)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `suite`: A suite handle.
- `name`: [OUT] The returned name of the suite. Pass NULL if you do not want this value returned.
- `num_space_sets`: [OUT] The number of sets returned in `mesh_space`.
- `mesh_space`: [OUT] The returned array of sets representing the computational meshes associated with each state of the suite. This is the list of sets in the “SAF__SPACE_SLICE” collection.
- `param_space`: [OUT] The returned array of sets representing the parametric space, such as time. These are associated with the histories of the suite and are thus contained in the “SAF__PARAM_SLICE” collection. This will not be implemented at this time.

Description: Returns the description of a suite. This includes the name of the suite and the sets associated with the “SAF__SPACE_SLICE” and “SAF__PARAM_SLICE” collections.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Suites](#): Introduction for current chapter

Find suites

`saf_find_suites` is a function defined in `suite.c`.

Synopsis:

```
int saf_find_suites (SAF_ParMode pmode, SAF_Db *database, const char *name, int *num_suites,  
                    SAF_Suite **suites)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `database`: The database in which to search.
- `name`: The name to limit the search to. The constant `SAF__ANY_NAME` can be passed if the client does not want to limit the search by name.
- `num_suites`: [OUT] The returned number of suites.
- `suites`: [OUT] The returned suites.

Description: Find all the suites in a [SAF](#) database. Under the cover, this finds all sets with a `SIL_ROLE` of `SAF__SUITE`.

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [Suites](#): Introduction for current chapter

Get an attribute attached to a suite

`saf_get_suite_att` is a function defined in `suite.c`.

Synopsis:

```
int saf_get_suite_att (SAF_ParMode pmode, SAF_Suite *suite, const char *att_key, hid_t *att_type,  
                      int *count, void **value)
```

Description: This function is identical to the generic [saf_get_attribute](#) function except that it is specific to `SAF__Suite` objects to provide the client with compile time type checking. For a description, see [saf_get_attribute](#).

See Also:

- [saf_get_attribute](#): 23.1: Read a non-sharable attribute
- [Suites](#): Introduction for current chapter

Attach an attribute to a suite

`saf_put_suite_att` is a function defined in `suite.c`.

Synopsis:

```
int saf_put_suite_att (SAF_ParMode pmode, SAF_Suite *suite, const char *att_key, hid_t att_type,
                     int count, const void *value)
```

Description: This function is identical to the generic [saf_put_attribute](#) function except that it is specific to `SAF__Suite` objects to provide the client with compile time type checking. For a description, see [saf_put_attribute](#).

See Also:

- [saf_put_attribute](#): 23.2: Create or update a non-sharable attribute
- [Suites](#): Introduction for current chapter

Quantities

A quantity in the general sense is a property ascribed to phenomena, bodies, or substances that can be quantified for, or assigned to, a particular phenomenon, body, or substance. The library defines seven basic quantities (length, mass, time, electric current, thermodynamic temperature, amount of a substance, and luminous intensity) and additional quantities can be derived as products of powers of the seven basic quantities (e.g., “volume” and “acceleration”). All quantities are unitless – they describe what can be measured but not how to measure it.

Unlike many other quantity implementations, this one is able to distinguish between dimensionless things like mass fractions (mass/mass) and length fractions (length/length). It does so by canceling numerators with denominators except when the numerator and denominator are equal. That is, mass/mass is considered a different quantity than length/length.

The library defines the seven basic quantities whose names follow the format “SAF__QX” where “X” is replaced by one of the words `LENGTH`, `MASS`, `TIME`, `CURRENT`, `TEMP`, `AMOUNT`, or `LIGHT`. Additional quantities can be derived from these by first creating an empty quantity and then multiplying powers of other quantities. For instance, volume per unit time would be defined as

```
1  SAF_Quantity *q_vpt = saf_declare_quantity(SAF_ALL, db, "volume per time", "vol/time
   ↪", NULL);
2  saf_multiply_quantity(SAF_ALL, q_vpt, SAF_QLENGTH, 3);
3  saf_multiply_quantity(SAF_ALL, q_vpt, SAF_QTIME, -1);
```

The reader is encouraged to visit physics.nist.gov/cuu/Units/units.html to get more information about quantities and units.

Members

The quantity Amount

`SAF_QAMOUNT` is a symbol defined in `SAFquant.h`.

Synopsis:

SAF_QAMOUNT

Description: A macro which refers to *Amount*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- [Quantities](#): Introduction for current chapter

The quantity Current

SAF_QCURRENT is a symbol defined in SAFquant.h.

Synopsis:

SAF_QCURRENT

Description: A macro which refers to *Current*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- *Quantities*: Introduction for current chapter

The quantity Length

SAF_QLENGTH is a symbol defined in SAFquant.h.

Synopsis:

SAF_QLENGTH

Description: A macro which refers to *Length*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- *Quantities*: Introduction for current chapter

The quantity Light

SAF_QLIGHT is a symbol defined in SAFquant.h.

Synopsis:

SAF_QLIGHT

Description: A macro which refers to *Light*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- *Quantities*: Introduction for current chapter

The quantity Mass

SAF_QMASS is a symbol defined in SAFquant.h.

Synopsis:

SAF_QMASS

Description: A macro which refers to *Mass*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- *Quantities*: Introduction for current chapter

An arbitrary named quantity

`SAF_QNAME` is a macro defined in `SAFquant.h`.

Synopsis:

`SAF_QNAME` (DB, NAME)

Description: A macro which refers to an arbitrary named quantity

See Also:

- [Quantities](#): Introduction for current chapter

The quantity Temperature

`SAF_QTEMP` is a symbol defined in `SAFquant.h`.

Synopsis:

`SAF_QTEMP`

Description: A macro which refers to *Temperature*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- [Quantities](#): Introduction for current chapter

The quantity Time

`SAF_QTIME` is a symbol defined in `SAFquant.h`.

Synopsis:

`SAF_QTIME`

Description: A macro which refers to *Time*, one of the 7 basic quantities defined at physics.nist.gov/cuu/Units/units.html

See Also:

- [Quantities](#): Introduction for current chapter

Declare a new quantity

`saf_declare_quantity` is a function defined in `quant.c`.

Synopsis:

`SAF_Quantity * saf_declare_quantity` (`SAF_ParMode pmode`, `SAF_Db *db`, `const char *description`, `const char *abbreviation`, `const char *url`, `SAF_Quantity *quant`)

Formal Arguments:

- `description`: A short description of the new quantity (e.g., “volume per time”).
- `abbreviation`: An optional abbreviation or symbol name for the quantity.
- `url`: An optional `url` to the quantity documentation.

- `quant`: [OUT] Optional quantity handle to initialize (and return).

Description: This function declares a new quantity whose product of powers is unity. The client is expected to multiply powers of other quantities into this new quantity (via [saf_multiply_quantity](#)) in order to complete its definition.

Preconditions:

- `pmode` must be valid. (low-cost)

Return Value: A new quantity handle is returned on success. Otherwise a `SAF__ERROR_HANDLE` value is returned or an exception is raised, depending on the error handling property of the library.

Parallel Notes: This function must be called collectively in the database communicator.

See Also:

- [saf_multiply_quantity](#): 20.14: *Multiply a quantity into a quantity definition*
- [Quantities](#): Introduction for current chapter

Query quantity characteristics

`saf_describe_quantity` is a function defined in `quant.c`.

Synopsis:

```
int saf_describe_quantity (SAF_ParMode pmode, SAF_Quantity *quantity, char **description,
                           char **abbreviation, char **url, unsigned *flags, unsigned *power)
```

Formal Arguments:

- `quantity`: Quantity about which to retrieve information.
- `description`: If non-null then upon return this will point to an allocated copy of the quantity description.
- `abbreviation`: If non-null then upon return this will point to an allocated copy of the quantity abbreviation if one is defined.
- `url`: If non-null then upon return this will point to an allocated copy of the quantity documentation url if one is defined.
- `flags`: If non-null then the special quantity flags are written into the location indicated by this pointer.
- `power`: If non-null then upon return this seven-element array will be filled in with the powers of the seven basic quantities.

Description: Given a `quantity` this function returns any information which is known about that quantity.

Preconditions:

- `pmode` must be valid. (low-cost)
- `quantity` must be a valid quantity handle. (low-cost)

Return Value: A non-negative value is returned on success. Failure is indicated by a negative return value or the raising of an exception, depending on the error handling property of the library.

Parallel Notes: This function must be called collectively in the database communicator.

See Also:

- [Quantities](#): Introduction for current chapter

Divide a quantity into a quantity definition

`saf_divide_quantity` is a macro defined in `SAFquant.h`.

Synopsis:

`saf_divide_quantity` (PMODE, Q, DIVISOR, POWER)

Description: This macro simply calls *`saf_multiply_quantity`* with a negated `POWER` argument.

See Also:

- *`saf_multiply_quantity`*: 20.14: *Multiply a quantity into a quantity definition*
- *Quantities*: Introduction for current chapter

Convenience function for finding a quantity

`saf_find_one_quantity` is a function defined in `quant.c`.

Synopsis:

`SAF_Quantity *` **`saf_find_one_quantity`** (`SAF_Db *database`, `const char *desc`, `SAF_Quantity *buf`)

Formal Arguments:

- `database`: The database in which to find the specified quantity.
- `desc`: Quantity description to find.
- `buf`: [OUT] Optional quantity handle to initialize and return.

Description: This is a simple version of `saf_find_quantity` that takes fewer arguments.

Return Value: On success, a handle for the first quantity found which has description `desc` in database `database` is returned. Otherwise a `SAF__ERROR_HANDLE` is returned.

Parallel Notes: This function must be called collectively in the database communicator.

See Also:

- *Quantities*: Introduction for current chapter

Find quantities

`saf_find_quantities` is a function defined in `quant.c`.

Synopsis:

`int` **`saf_find_quantities`** (`SAF_ParMode pmode`, `SAF_Db *db`, `const char *desc`, `const char *abbr`, `const char *url`, `unsigned flags`, `int *power`, `int *num`, `SAF_Quantity **found`)

Formal Arguments:

- `db`: Database in which to limit the search.
- `desc`: Optional quantity description for which to search.
- `abbr`: Optional abbreviation for which to search.
- `url`: Optional `url` for which to search.
- `flags`: Optional flags for which to search, or `SAF__ANY_INT`.

- `power`: Optional base quantity powers for which to search. If the pointer is non-null then the elements can be `SAF__ANY_INT` for the ones in which the caller is not interested.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function allows a client to search for quantities in the database. The search may be limited by one or more criteria such as the name of the quantity, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`

See Also:

- [Quantities](#): Introduction for current chapter

Multiply a quantity into a quantity definition

`saf_multiply_quantity` is a function defined in `quant.c`.

Synopsis:

```
int saf_multiply_quantity (SAF_ParMode pmode, SAF_Quantity *quantity, SAF_Quantity *multiplier, int power)
```

Formal Arguments:

- `quantity`: IN``[``OUT] The quantity which is affected by this operation
- `multiplier`: What to multiply into `quantity`
- `power`: Number of times to multiply `multiplier` into `quantity`

Description: After creating a new quantity with [saf_declare_quantity](#), the `quantity` is defined by multiplying powers of other quantities into it, one per call to this function. A division can be accomplished by supplying a negative `power` (a `power` of zero has no effect).

Preconditions:

- `pmode` must be valid. (low-cost)
- `quantity` must be a valid quantity handle. (low-cost)
- `multiplier` must be a valid quantity handle. (low-cost)

Return Value: This function returns some non-negative value on success; otherwise, it either returns a negative value or raises an exception, depending on the error handling property of the library.

Parallel Notes: Depends on the `pmode` argument.

See Also:

- [saf_declare_quantity](#): 20.9: *Declare a new quantity*
- [saf_divide_quantity](#): 20.11: *Divide a quantity into a quantity definition*

- *Quantities*: Introduction for current chapter

Units

A unit is a particular physical quantity, defined and adopted by convention, with which other particular quantities of the same kind are compared to express their value. The library has two classes of units: *basic units* and *derived units*. Basic units measure some arbitrary amount of a specific quantity while derived units are created by multiplying, scaling, and translating powers of other units (basic and/or derived). All units are associated with a specific quantity of the database either explicitly or implicitly. Implicit association is allowed if the appropriate quantity is not ambiguous. The library is able to convert an array of measurements from one unit to another if the source and destination unit measure the same specific quantity.

The definition of a basic unit is a two step process. First an empty definition is created with *saf_declare_unit*, then the unit is associated with a quantity with *saf_quantify_unit*. Example: define meters as a basic unit of length. That is, meters measures some arbitrary amount of length and will be the basis for deriving all compatible units.

```
1 SAF_Unit *m = saf_declare_unit(SAF_ALL,db,"meter","m",NULL);
2 saf_quantify_unit(SAF_ALL,m,SAF_QLENGTH,1);
3 saf_commit(m,SAF_ALL,database);
```

The definition of derived units is similar when the new unit measures the same quantity. Example: define kilometers as 1000 meters (km and m both measure the same quantity).

```
1 SAF_Unit *km = saf_declare_unit(SAF_ALL,db,"kilometer","km",NULL);
2 saf_multiply_unit(SAF_ALL,km,1000,m,1);
```

Another way to define a unit is to multiply other units together. When this happens the new unit measures a different quantity than its unit divisors. In most cases the library can figure out what specific quantity to use for the unit, but this is not possible when the library contains multiple quantity definitions for similar quantities (e.g., ‘molecular amount’ and ‘monetary amount’ are both amount-of-a-substance quantities, but the library has two separate quantity definitions because it should not be possible to convert between moles and dollars). Example: define coulomb as an ampere second instead of some arbitrary amount of charge:

```
1 SAF_Unit *C = saf_declare_unit(SAF_ALL,db,"coulomb","C",NULL);
2 saf_multiply_unit(SAF_ALL,C,1,A,1); // ampere
3 saf_multiply_unit(SAF_ALL,C,1,s,1); // second
4 SAF_Quantity *charge = saf_find_one_quantity(db,"electric charge",NULL);
5 saf_quantify_unit(SAF_ALL,C,charge,1);
```

In the previous example the *saf_quantify_unit* could have been omitted since the library only defines one electric charge quantity and there is no ambiguity.

Two notable units are thermodynamic temperature measured in absolute Celsius and Fahrenheit. Both of these are the same amount as a degree Kelvin or a degree Rankine, but are offset by some amount. These units can be declared with *saf_offset_unit*:

```
1 SAF_Unit *absC = saf_declare_unit(SAF_ALL,db,"absolute Celceus","absC",NULL);
2 saf_multiply_unit(SAF_ALL,absC,1,k,1); // degree Kelvin
3 saf_offset_unit(SAF_ALL,absC,273.5); // 0 deg C is 273.15 k
```

Another special type of unit is one which uses a logarithmic scale instead of a linear scale. For example, a decibel is a dimensionless measure of the ratio of two powers, equal to ten times the logarithm to the base ten of the ratio of two powers. In acoustics the decibel is 20 times the common log of the ratio of sound pressures, with the denominator usually being 2e-5 pascal. The *saf_log_unit* can be used to define such a unit:

```
1  SAF_Unit *dB = saf_declare_unit(SAF_ALL,db,"decibel","dB",NULL);
2  SAF_Quantity *spr = saf_find_one_quantity(db,"sound pressure ratio",NULL);
3  saf_quantify_unit(SAF_ALL,dB,spr,1);
4  saf_log_unit(SAF_ALL,dB,10,20);
```

The *saf_offset_unit* and *saf_log_unit* can only be applied to a unit after all multiplications have been performed, and such a unit cannot be used to derive other units.

Members

Declare a new unit

`saf_declare_unit` is a function defined in `unit.c`.

Synopsis:

```
SAF_Unit * saf_declare_unit (SAF_ParMode pmode, SAF_Db *db, const char *name, const
                             char *abbr, const char *url, SAF_Unit *unit)
```

Formal Arguments:

- `db`: The database in which to create the new unit.
- `name`: Optional singular unit name.
- `abbr`: Optional singular abbreviation
- `url`: Optional documentation url.
- `unit`: [OUT] Optional unit handle to initialize and return.

Description: This function declares a new unit whose product of quantity powers is unity. The client is expected to multiply powers of other quantities or units into this new unit via *saf_multiply_unit*.

Preconditions:

- `pmode` must be valid. (low-cost)

Return Value: A new unit handle is returned on success. Otherwise a `SAF__ERROR_HANDLE` is returned or an exception is raised, depending on the error handling property of the library.

See Also:

- *saf_multiply_unit*: 21.8: *Multiply a unit into a unit definition*
- *Units*: Introduction for current chapter

Query unit characteristics

`saf_describe_unit` is a function defined in `unit.c`.

Synopsis:

```
int saf_describe_unit (SAF_ParMode pmode, SAF_Unit *unit, char **name, char **abbr,
                      char **url, double *scale, double *offset, double *logbase, double *logcoef,
                      SAF_Quantity *quantity)
```

Formal Arguments:

- `unit`: Unit about which to retrieve information.
- `name`: If non-null then upon return this will point to an allocated copy of the unit singular name.

- `abbr`: If non-null then upon return this will point to an allocated copy of the unit singular abbreviation.
- `url`: If non-null then upon return this will point to an allocated copy of the `url` for the unit's documentation.
- `scale`: If non-null then upon return `*scale` will be the scale factor for the unit.
- `offset`: If non-null then upon return `*offset` will be the offset for the unit.
- `logbase`: If non-null then upon return `*logbase` will be the logarithm base for the unit. The returned value zero indicates no logarithm is applied.
- `logcoef`: If non-null then upon return `*logcoef` will be the multiplier of the logarithmic scale.
- `quantity`: If non-null then upon return this will point to the handle of the quantity on which this unit is based. If the `unit` has not been defined yet (such as calling this function immediately after [saf_declare_unit](#)) then the quantity handle will be initialized to a null link.

Description: Given a `unit`, this function returns any information which is known about that unit.

Preconditions:

- `pmode` must be valid. (low-cost)
- `unit` must be a valid unit handle. (low-cost)

Return Value: A non-negative value is returned on success. Failure is indicated by a negative return value or the raising of an exception, depending on the error handling property of the library.

See Also:

- [saf_declare_unit](#): 21.1: *Declare a new unit*
- [Units](#): Introduction for current chapter

Divide a unit into a unit definition

`saf_divide_unit` is a macro defined in `SAFunit.h`.

Synopsis:

`saf_divide_unit` (`U`, `SCALE`, `DIVISOR`, `POWER`)

Description: This macro simply calls [saf_multiply_unit](#) with a negated `POWER` argument and the reciprocal of the `SCALE` argument.

See Also:

- [saf_multiply_unit](#): 21.8: *Multiply a unit into a unit definition*
- [Units](#): Introduction for current chapter

Convenience function for finding a unit

`saf_find_one_unit` is a function defined in `unit.c`.

Synopsis:

`SAF_Unit *`**`saf_find_one_unit`** (`SAF_Db *`*database*, `const char *`*name*, `SAF_Unit *`*buf*)

Formal Arguments:

- `database`: The database in which to find the specified unit.
- `name`: The singular name of the unit to find, e.g., “meter”.

- `buf`: [OUT] Optional unit handle to initialize and return.

Description: This is a simple version of `saf_find_unit` that takes fewer arguments.

Return Value: On success, a handle for the first unit found which has name `name` or abbreviation `name` in database. Otherwise a `SAF__ERROR_HANDLE` is returned.

See Also:

- [Units](#): Introduction for current chapter

Find the not applicable unit

`saf_find_unit_not_applicable` is a function defined in `unit.c`.

Synopsis:

`SAF_Unit * saf_find_unit_not_applicable (void)`

Description: Find and return the not applicable unit.

Return Value: On success, a handle for the first unit found. Otherwise a `SAF__ERROR_HANDLE` is returned.

See Also:

- [Units](#): Introduction for current chapter

Find units

`saf_find_units` is a function defined in `unit.c`.

Synopsis:

`int saf_find_units (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *abbr,
const char *url, double scale, double offset, double logbase, double logcoef,
SAF_Quantity *quant, int *num, SAF_Unit **found)`

Formal Arguments:

- `db`: Database in which to limit the search.
- `name`: Optional unit description for which to search.
- `abbr`: Optional abbreviation for which to search.
- `url`: Optional `url` for which to search.
- `scale`: Optional scale for which to search (or pass `SAF__ANY_DOUBLE`).
- `offset`: Optional offset for which to search (or pass `SAF__ANY_DOUBLE`).
- `logbase`: Optional logarithm base for which to search (or pass `SAF__ANY_DOUBLE`).
- `logcoef`: Optional logarithm coefficient for which to search (or pass `SAF__ANY_DOUBLE`).
- `quant`: Optional quantity for which to search.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function allows a client to search for units in the database. The search may be limited by one or more criteria such as the name of the unit, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`

See Also:

- [Units](#): Introduction for current chapter

Apply a logarithmic scale to a unit

`saf_log_unit` is a function defined in [unit.c](#).

Synopsis:

```
int saf_log_unit (SAF_ParMode pmode, SAF_Unit *unit, double logbase, double logcoef)
```

Formal Arguments:

- `unit`: The unit which is being translated by `OFFSET`.
- `logbase`: The base of the logarithm
- `logcoef`: The amount by which to multiply the unit after taking the log.

Description: Some units of measure use a logarithmic scale. An example is decibels. This function sets the base for the logarithm. A `logbase` of zero implies a linear scale and is the default for all units. This function should only be called after any calls to [saf_multiply_unit](#) and [saf_offset_unit](#) for `unit`.

```
U' = LOGCOEF *log* UNIT
```

where **log** is to the base `logbase`.

Preconditions:

- `pmode` must be valid. (low-cost)
- `unit` must be a valid unit handle. (low-cost)
- `logbase` must be non-negative. (low-cost)
- `logcoef` must be non-zero if a logarithmic scale is used. (low-cost)

Return Value: This function returns some non-negative value on success; otherwise, it either returns a negative value or raises an exception, depending on the error handling property of the library.

See Also:

- [saf_multiply_unit](#): 21.8: *Multiply a unit into a unit definition*
- [saf_offset_unit](#): 21.9: *Translate unit by an offset*
- [Units](#): Introduction for current chapter

Multiply a unit into a unit definition

`saf_multiply_unit` is a function defined in *unit.c*.

Synopsis:

```
int saf_multiply_unit (SAF_ParMode pmode, SAF_Unit *unit, double coef, SAF_Unit *multiplier,  
                      int power)
```

Formal Arguments:

- `unit`: The unit which is being modified by multiplying `multiplier` into it.
- `coef`: A real coefficient multiplied into `unit`
- `multiplier`: The optional multiplicand unit
- `power`: The power to which `multiplier` is raised before multiplying it into `unit`

Description: After creating a new unit with *saf_declare_unit*, the `unit` is defined by multiplying scaled powers of other units into it, one per call to this function. A division by `multiplier` can be accomplished by supplying a negative `power`, although `coef` is always multiplied into U. Essentially, the result is:

$$\text{UNIT}' = \text{UNIT} * \text{COEF} * (\text{MULTIPLIER} ^ \text{POWER})$$

If `multiplier` is `NULL` then it is assumed to be unity. In other words, the scale factor can be adjusted for the unit by calling this function with only a `coef` value.

Preconditions:

- `pmode` must be valid. (low-cost)
- `unit` must be a valid unit handle. (low-cost)
- `unit` must have a zero offset (the default). (low-cost)
- `unit` must not have a logarithm base assigned (the default). (low-cost)
- `coef` must be positive. (low-cost)
- `multiplier` must be a valid unit handle if supplied. (low-cost)
- `multiplier` must have a zero offset if supplied. (low-cost)
- `multiplier` must not use a logarithmic scale if supplied. (low-cost)

Return Value: This function returns some non-negative value on success; otherwise, it either returns a negative value or raises an exception, depending on the error handling property of the library.

See Also:

- *saf_declare_unit*: 21.1: *Declare a new unit*
- *saf_divide_unit*: 21.3: *Divide a unit into a unit definition*
- *Units*: Introduction for current chapter

Translate unit by an offset

`saf_offset_unit` is a function defined in *unit.c*.

Synopsis:

```
int saf_offset_unit (SAF_ParMode pmode, SAF_Unit *unit, double offset)
```

Formal Arguments:

- `unit`: The unit which is being translated by `offset`.
- `offset`: The amount by which to translate the unit.

Description: Some units of measure have a scale which is translated from the origin by some amount. The most notable examples are absolute degrees Celsius and Fahrenheit.

```
1 SAF_Unit absC = saf_declare_unit("Celsius", "absC");
2 SAF_multiply_unit(absC, 1, kelvin, 1);
3 SAF_offset_unit(absC, 273.15);
```

Preconditions:

- `pmode` must be valid. (low-cost)
- `unit` must be a valid unit handle. (low-cost)
- `unit` must not have a logarithm base assigned (the default). (low-cost)

Return Value: This function returns some non-negative value on success; otherwise, it either returns a negative value or raises an exception, depending on the error handling property of the library.

See Also:

- [Units](#): Introduction for current chapter

Associates a unit of measure with a specific quantity

`saf_quantify_unit` is a function defined in `unit.c`.

Synopsis:

```
int saf_quantify_unit (SAF_ParMode pmode, SAF_Unit *unit, SAF_Quantity *quantity, double scale)
```

Formal Arguments:

- `unit`: The unit whose quantity information is being set.
- `quantity`: The quantity which this unit measures.
- `scale`: This argument can be used to defined a new unit as some scale of the base unit for the quantity without requiring the unit definition to include a multiplication by the base unit. The `scale` is multiplied into any scale which is already present.

Description: A basic unit is a unit which measures an arbitrary amount of some quantity, and is defined simply by associating the unit with its quantity by calling this function. (no multiplications by other units are necessary).

Derived units are built by multiplying together powers of one or more other units. If just one unit is multiplied into the new definition then the new definition will refer to the same specific quantity as the unit on which it is based (if the power is one). Otherwise, when units are multiplied together the quantity measured by the product is different than the quantity measured by any of the multiplicands. When this happens it may be necessary for the client to call this function to associate a specific quantity with this new unit (it is not necessary if the library can deduce the specific quantity unambiguously from the unit's database).

Preconditions:

- `pmode` must be valid. (low-cost)
- `unit` must be a valid unit handle. (low-cost)
- `quantity` must be a valid quantity handle. (low-cost)
- `scale` must be positive. (low-cost)

Return Value: A non-negative value is returned on success; otherwise either a negative value is returned or an exception is raised, depending on the error handling property of the library.

See Also:

- *Units*: Introduction for current chapter

Attributes

As mentioned in the object handles chapter (see Object Handles) there currently (saf-1.2.0) exist two styles of handles: “old” handles and “new” handles. For each “old” object class there are functions to put (`saf_put_XXX_att`) and get (`saf_get_XXX_att`) attributes, as well as generic forms of these functions (`saf_put_attribute` and `saf_get_attribute`) which operate on any object type but do not provide rigorous compile-time type checking. The “new” object classes use only `saf_putAttribute` and `saf_getAttribute`, which employ compile-time and run-time type checking.

There is an important limitation to the attributes interface in **SAF**. First and foremost, it should be clearly understood that there is **no expectation** that any data stored in attributes be shareable. If there is any expectation that **any** software other than the writer of the attributes should be sensitive to and/or aware of the data stored in them, the data should **not** be stored in attributes. If for some reason, your client is unable to model important features of the data without encoding something into attributes, then the current implementation of this data model is failing.

By convention, attributes whose names begin with a dot (“.”) are read-only. Thus, a client may create and initialize a new attribute whose name begins with a dot, but thereafter any client operating on the database can only read the value of that attribute.

Issues: Each attribute has its own HDF5 dataset in the **SAF** file. For a `SAF__EACH` mode call, we need to loop creating *num_procs* datasets.

Also, performance of attribute access is likely to be poor, particularly in parallel.

Members

Read a non-sharable attribute

`saf_get_attribute` is a function defined in `utils.c`.

Synopsis:

```
int saf_get_attribute (SAF_ParMode pmode, ss_pers_t *obj, const char *name, hid_t *type, int *count,  
                      void **value)
```

Formal Arguments:

- `pmode`: One of the parallel modes.
- `obj`: The handle to the object from which the attribute will be read.
- `name`: The name of the attribute. See `SAF__ATT_NAMES` and other reserved attribute names for special kinds of attribute queries.
- `type`: IN` `[``OUT] If `type` is NULL, this argument will be ignored. If `type` points to a valid datatype, then the attribute will be converted to the specified type as it is read. If it does not, there will be **no data conversion** and the output value will be the datatype of the data returned (the caller should invoke `H5Tclose`).
- `count`: [OUT] The number of items in the attribute. If `count` is NULL, then the value of `count` will not be returned.
- `value`: IN` `[``OUT] Points to an array of `count` values each having datatype `type`. If `value` is NULL, then no attribute values will be returned. If `value` points to NULL, then the library will allocate the array of values which is returned. Otherwise the library assumes that `value` points to an array whose size is sufficient

for storing `count` values of datatype `type`. That is, if `value` is pointing to non-NULL, then so must `count` point to non-NULL **and** the value pointed to by `count` will be used by SAF as the size, in items of type `type`, of the block of memory pointed to by `value`. For a `SAF__ATT_NAMES` query if the caller supplies a buffer for this argument then it should be a buffer of `char` pointers, the values of which will be allocated by this function.

Description: This function provides a method by which existing, generic, non-sharable attributes may be read from an object. Attributes are pieces of meta data which fall outside the scope of the sharable data model (i.e., things which are not fields) but which are often useful for conveying additional information. The meaning of a particular attribute is determined by convention, requiring additional communication between the writer and the reader (often in the form of documentation or word of mouth).

Preconditions:

- `pmode` must be valid. (low-cost)
- `name` must not be null. (low-cost)
- `count` and `value` must be compatible for return value allocation. (low-cost)
- `obj` must not be null. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`

Issues: On error, the argument return values are undefined.

If the attribute `name` is `SAF__ATT_NAMES` then the client must not preallocate space for the `value` return value, but must allow the library to handle the allocation. That is, if the arg passed for `name` is `SAF__ATT_NAMES`, the client must not pass `value` such that it points to a non-null pointer.

The reserved attribute name queries, `SAF__ATT_FIRST` and `SAF__ATT_NEXT`, are not yet implemented.

This is a weird interface. There should be a separate function to obtain the datatype and count of an attribute so that this function doesn't need to return those values. The `type` and `count` arguments should instead specify what value is returned by this function. And the `value` should be just an optional `'void*'` buffer which if not supplied is allocated and which is the successful return value of this function. [rpm 2004-08-25]

If the pool allocation is being used then we'll have a problem if there are more attributes than what the string pool can store.

See Also:

- [Attributes](#): Introduction for current chapter

Create or update a non-sharable attribute

`saf_put_attribute` is a function defined in `utils.c`.

Synopsis:

```
int saf_put_attribute (SAF_ParMode pmode, ss_pers_t *obj, const char *name, hid_t type, int count,
                     const void *value)
```

Formal Arguments:

- `pmode`: One of the parallel modes.
- `obj`: The handle to the object the attribute is to be associated with.
- `name`: The name of the attribute.
- `type`: The datatype of the attribute.

- `count`: The number of items of type `type` pointed to by `*value`.
- `value`: The attribute value(s) (an array of `count` value(s) of type `type`).

Description: This function provides a method by which generic, non-sharable attributes may be added to an object. Attributes are pieces of meta data which fall outside the scope of the sharable data model (i.e., things which are not fields) but which are often useful for conveying additional information. The meaning of a particular attribute is determined by convention, requiring additional, apriori agreement between the writer and the reader (often in the form of documentation or word of mouth) as to the meaning and intent of a given attribute/value pair.

If `type` is `H5T_C_S1` (which isn't very useful by itself since it's just a one-byte string that's always the `NUL` character) then a temporary datatype is created which is exactly as long as the `value` string including its `NUL` terminator. `value` in this case should be a pointer to `char`. Be aware that querying the attribute for its datatype will not return `H5T_C_S1` unless the string was empty.

Preconditions:

- `pmode` must be valid. (low-cost)
- `obj` must not be null. (low-cost)
- `name` must not be null. (low-cost)
- `count` must be non-negative. (low-cost)
- If `count` is non-zero, `value` must not be null. (low-cost)
- Database in which object exists must not be open for read-only access. (no-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`

See Also:

- [Attributes](#): Introduction for current chapter

Miscellaneous Utilities

No description available.

Members

Synchronization barrier

`SAF_BARRIER` is a macro defined in `saf.h`.

Synopsis:

`SAF_BARRIER` (`Db`)

Description: A macro which causes all processors in the communicator used to open the database or, if `Db` is `NULL`, to initialize the library, to wait until all reach this point (See [*Constants*](#)).

See Also:

- [Miscellaneous Utilities](#): Introduction for current chapter

Determine if two handles refer to the same object

SAF_EQUIV is a macro defined in saf.h.

Synopsis:

SAF_EQUIV (A, B)

Description: This macro returns true if the two object handles passed to it refer to the same object. Otherwise, it returns false.

See Also:

- *Miscellaneous Utilities*: Introduction for current chapter

Array size

SAF_NELMTS is a macro defined in safP.h.

Synopsis:

SAF_NELMTS (X)

Description: Return number of elements in array.

See Also:

- *Miscellaneous Utilities*: Introduction for current chapter

The rank of the calling process

SAF_RANK is a macro defined in saf.h.

Synopsis:

SAF_RANK (Db)

Description: A macro which evaluates to the MPI_Rank of the calling processor in the communicator used to open the database. If NULL is passed for the Db argument, the MPI_Rank of the calling process in the communicator used to initialize the library is returned. In serial, a value of 0 is returned. If not called within an enclosing pair of *saf_init* / *saf_final* calls, the value -1 is returned (See **Constants**).

See Also:

- *saf_final*: 4.2: *Finalize access to the library*
- *saf_init*: 4.3: *Initialize the library*
- *Miscellaneous Utilities*: Introduction for current chapter

The size of the communicator

SAF_SIZE is a macro defined in saf.h.

Synopsis:

SAF_SIZE (Db)

Description: A macro which evaluates to the `MPI_Size` of the communicator used to open the database. If `NULL` is passed for the `Db` argument, the `MPI_Size` of the communicator used to initialize the library is returned. In serial a value of 1 is returned. If not called within an enclosing pair of *saf_init* / *saf_final* calls, the value -1 is returned (See **Constants**).

See Also:

- *saf_final*: 4.2: *Finalize access to the library*
- *saf_init*: 4.3: *Initialize the library*
- *Miscellaneous Utilities*: Introduction for current chapter

Determine if a handle is a valid handle

`SAF_VALID` is a macro defined in `saf.h`.

Synopsis:

SAF_VALID (A)

Description: This macro returns true if the handle passed to it is valid, that is, that its members define a legitimate handle. Otherwise, it returns false.

See Also:

- *Miscellaneous Utilities*: Introduction for current chapter

Exclusive OR operator

`SAF_XOR` is a macro defined in `safP.h`.

Synopsis:

SAF_XOR (A, B)

Description: Returns A XOR B

See Also:

- *Miscellaneous Utilities*: Introduction for current chapter

Copy a string

`_saf_strdup` is a function defined in `utils.c`.

Synopsis:

char * **_saf_strdup** (const char *s)

Description: Same functionality as `strdup` but returns an empty string when `s` is the null pointer.

Return Value: Returns an allocated, null terminated string on success; null on failure.

Parallel Notes: Independent

See Also:

- *Miscellaneous Utilities*: Introduction for current chapter

Exchange handles

`saf_allgather_handles` is a function defined in `utils.c`.

Synopsis:

```
ss_pers_t * saf_allgather_handles (ss_pers_t *_pers, int *commsize, ss_pers_t *result)
```

Formal Arguments:

- `_pers`: A Pointer to the handle to be exchanged. Every participant must supply a valid handle of the same type and in the same scope as every other participant.
- `commsize`: [OUT] A pointer to optional caller supplied memory which is to receive the integer number of handles returned by this function. This is the number of participants or the size of the communicator associated with the given database.
- `result`: [OUT] An optional pointer to an array that will be initialized with a handle from each MPI task in task rank order. If this buffer is supplied then it should be at least as large as the communicator associated with the DB argument. If not supplied (i.e., null) then a buffer will be allocated for the return value.

Description: This function is used to exchange handles created locally by processes for global writing. This is generally done when collecting the local handles to be written stored with an indirect relation or field.

Preconditions:

- `_pers` must be a valid object link. (low-cost)

Return Value: Returns a buffer of handles on success or null on failure. The buffer is either the non-null value of the `result` argument or a buffer which is allocated by this function.

Parallel Notes: This call must be collective across the communicator for the given database.

See Also:

- *Miscellaneous Utilities*: Introduction for current chapter

Version Numbers

The SAF source code has various version numbers assigned to parts of the system: source files, header files, library, API definition, and database files.

Source file versions are identical to CVS file revision numbers. These numbers are **not** stored in the source file but rather maintained by CVS. (We don't store them in the source file because it makes it more difficult to synchronize local and remote source trees since a 'cvs commit' would modify all the source files.) We use CVS in such a way that the main branch always contains the latest development version of SAF. When a public release is about to occur a new branch is created, version numbers are adjusted on both branches, and development stops on the new branch.

The header files and library each have a four-part version number: *major*, *minor*, *patch*, and *comment*. The version number of the header files must exactly match the version number of the library, or the library will refuse to operate. The major number is incremented only if the API changes in a way which is not backward compatible. The increment happens when the development branch is split to produce a new release branch, and the minor number is set to zero or one (depending on branch) and the patch number is reset to zero. The minor number is incremented each time the main branch is split to produce a release branch. The minor number is always even on a release branch and odd on the development branch (the latest development version minor number is one greater than the latest release version). The patch number is incremented each time bugs are fixed on the release branch, or each time a snapshot is produced on the development branch. The comment is a character string indicating the scope of the release and is the empty string for all public releases and snapshots. Library version numbers are printed as *i.j.k-c* where *i* is the major number, *j* is the minor number, *k* is the patch number, and *-c* is the comment string (the hyphen is printed only if the comment string is non-empty).

The API definition has a two-part version number which is the same as the major and minor version numbers of the header files and library. For any given release or snapshot the library must implement the corresponding version of the API. The API may document certain features as “not yet implemented”.

Database files will contain the library version number as an attribute named “SAF” attached to the group containing the VBT files. The attribute will be of compound type and contain all global SAF metadata.

Standard Comment Strings: The comment string for all development versions which have not yet passed the snapshot operation will be ‘devel’. When the main branch is split to create a release branch the comment string on the release branch will be cleared. Pre-releases will then be created from the release branch while holding the patch number at zero so the release can be tested by the developers. Such prereleases will be commented as ‘preN’ where *n* is a number beginning at zero. When a prerelease passes all developer tests the comment will be removed or changed to ‘beta’.

Almost all programs call *saf_init* and/or *saf_open_database* in order to do something useful. So we’ve chosen to wrap those functions in macros which also make a reference to a global variable whose name is derived from the SAF version number. This variable is declared in the SAF library so that if an application is compiled with SAF header files which have a different version than the SAF library a link-time error will result. A version mismatch will result in an error similar to *undefined reference to “SAF__version_0_1_0”* from the linker.

Members

Serial/Parallel-dependent variable

SAF_PARALLEL_VAR is a symbol defined in saf.h.

Synopsis:

SAF_PARALLEL_VAR

Description: This is simply a global integer variable whose name depends somehow on whether the library is being compiled for serial or parallel. It is used to check at link-time whether the header files used by an application match the SAF library to which the application is linked.

See Also:

- *Version Numbers*: Introduction for current chapter

Version Annotation

SAF_VERSION_ANNOT is a symbol defined in saf.h.

Synopsis:

SAF_VERSION_ANNOT

Description: The version annotation of the SAF header files. This indicates a restriction of the release (such as ‘beta’).

See Also:

- *Version Numbers*: Introduction for current chapter

Major version number

SAF_VERSION_MAJOR is a symbol defined in saf.h.

Synopsis:

SAF_VERSION_MAJOR

Description: The major version number of the SAF header files. If this number is not equal to the major version number of the SAF library with which the application was linked then the library will raise an error.

See Also:

- *Version Numbers*: Introduction for current chapter

Minor version number

SAF_VERSION_MINOR is a symbol defined in saf.h.

Synopsis:

SAF_VERSION_MINOR

Description: The minor version number of the SAF header files. If this number is not equal to the minor version number of the SAF library with which the application was linked then the library will raise an error.

See Also:

- *Version Numbers*: Introduction for current chapter

Release number

SAF_VERSION_RELEASE is a symbol defined in saf.h.

Synopsis:

SAF_VERSION_RELEASE

Description: The patch number of the SAF header files. If this number is not equal to the patch number of the SAF library with which the application was linked then the library will raise an error.

See Also:

- *Version Numbers*: Introduction for current chapter

Version-dependent variable

SAF_VERSION_VAR is a symbol defined in saf.h.

Synopsis:

SAF_VERSION_VAR

Description: This is simply a global integer variable whose name depends somehow on the SAF version numbers defined above. It is used to check at link-time whether the header files used by an application match the SAF library version number to which the application is linked.

See Also:

- *Version Numbers*: Introduction for current chapter

Returns string representation of version number

`saf_version_string` is a function defined in `utils.c`.

Synopsis:

```
char * saf_version_string (int verbose, char *buffer, size_t bufsize)
```

Description: Provides a function that should be used so version numbers all have a common format. If `verbose` is set then the returned string will be of the form ‘version 1.2 release 3 (comment)’, otherwise the returned string will be of the form ‘1.2.3-comment’. The ‘ (comment)’ or ‘-comment’ part of the string is omitted if there is no version annotation.

Return Value: `buffer`

See Also:

- [Version Numbers](#): Introduction for current chapter

Datatypes

No description available.

Members

Wildcards for searching

SAF is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

`SAF_ANY_INT:`

`SAF_ANY_DOUBLE:`

`SAF_ANY_FLOAT:`

`SAF_ANY_TOPODIM:`

`SAF_CELLTYPE_ANY:`

`SAF_ANY_RATIO:`

`SAF_ANY_NAME:`

`SAF_ANY_CAT:`

Description: In `saf_find` calls, the client may not want to limit the search to all of the available argument’s values. **SAF_** offers these wildcard values, all with the word *ANY* in them, to pass as the value for an argument that the client does NOT wish to use in limiting a search. For example, see `saf_find_matching_set`.

See Also:

- [Datatypes](#): Introduction for current chapter

Indexing scheme

`SAF_1DC` is a macro defined in `saf.h`.

Synopsis:

SAF_1DC (nx)

Description: One-dimensional C array of size nx. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_1DF is a macro defined in saf.h.

Synopsis:

SAF_1DF (nx)

Description: One-dimensional Fortran array of size nx. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_2DC is a macro defined in saf.h.

Synopsis:

SAF_2DC (nx, ny)

Description: Two-dimensional C array of size nx by ny. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_2DF is a macro defined in saf.h.

Synopsis:

SAF_2DF (nx, ny)

Description: Two-dimensional Fortran array of size nx by ny. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_3DC is a macro defined in saf.h.

Synopsis:

SAF_3DC (nx, ny, nz)

Description: Three-dimensional C array of size nx, ny, nz elements. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_3DF is a macro defined in saf.h.

Synopsis:

SAF_3DF (nx, ny, nz)

Description: Three-dimensional Fortran array of size nx, ny, nz elements. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Reserved attribute name keys

SAF_ATT is a collection of related C preprocessor symbols defined in saf.h.

Synopsis:

SAF_ATT_NAMES: If the client passes SAF__ATT_NAMES for the NAME arg in a call to *saf_get_attribute*, SAF will return a TYPE of string (if the TYPE return value is requested), a COUNT equal to the number of attributes (if the COUNT return value was requested), and a VALUE array containing the names of all attributes defined for the object.

SAF_ATT_COUNT: If the client passes SAF__ATT_COUNT for the NAME arg in a SAF call to *saf_get_attribute*, SAF will return the count of number of attributes defined for the given object in the COUNT. It is an error to request a count with SAF__ATT_COUNT, but pass NULL for the COUNT argument in a call to get attributes.

SAF_ATT_FIRST: If the client passes SAF__ATT_FIRST, for the NAME argument in a SAF call to *saf_get_attribute*, SAF will return the **first** attribute that was ever defined for the object. Thereafter, any call with SAF__ATT_NEXT will iterate through the list of attributes defined for the object.

SAF_ATT_NEXT: This reserved attribute name works in conjunction with SAF__ATT_FIRST, to allow the client to iterate through all attributes defined for a given object. It is an error to pass SAF__ATT_NEXT without at least one prior call with SAF__ATT_FIRST.

Description: There are some reserved attribute names. These reserved attribute names may be passed as the NAME argument in any calls to get attributes (see *saf_get_attribute*). The SAF__ATT_NAMES / SAF__ATT_COUNT pair of reserved names provide a mechanism to the client to determine the count of attributes defined for a given object and their names. Or, alternatively, the SAF__ATT_FIRST / SAF__ATT_NEXT provide a mechanism for the client to make repetitive calls to iterate through the attributes for a given object.

See Also:

- *saf_get_attribute*: 23.1: Read a non-sharable attribute
- *Datatypes*: Introduction for current chapter

Basis types

SAF_BasisConstants is a collection of related C preprocessor symbols defined in SAFbasis.h.

Synopsis:

SAF_UNITY: The basis set with a single basis vector; {1}

SAF_CARTESIAN: The basis set with *N* basis vectors; {e0, e1, ..., eN}

SAF_SPHERICAL: The basis set with 3 basis vectors {r, theta, phi}

SAF_CYLINDRICAL: The basis set with 3 basis vectors {r, theta, h}

SAF_UPPERTRI: The basis set of a symmetric tensor. Why do we need this if the algebraic type already captures it?

SAF_VARIYING: For a basis that is varying over the base space. Often needed if the basis is derived from local surface behavior such as surface normals. Although, shouldn't we use something like SAF__SURFACE_NORMAL for that?

SAF_ANY_BASIS: Wildcard for searching.

Description: For every field, not just coordinate fields, SAF needs to be told what are the basis vectors for identifying the field's values. For example, if we have a field of N pairs of floats representing complex numbers, do those floats represent the real and imaginary part of the complex number (e.g. cartesian basis) or do they represent the magnitude and phase (e.g. the polar basis).

Likewise, if we have N triples representing color of each pixel in image are they RGB triples, LUV triples, YIQ triples, etc.? The basis type is designed to indicate what the basis vectors for a given field are.

See Also:

- [Datatypes](#): Introduction for current chapter

Boundary set tri-state

SAF_BoundMode is an enumerated type defined in saf.h.

Synopsis:

SAF_BOUNDARY_FALSE:

SAF_BOUNDARY_TRUE:

SAF_BOUNDARY_TORF:

Description: To make each function call made by a client a little more self-documenting, we provide specific tri-state tags to represent the meaning of that particular boolean. The one here is used to indicate whether a one set in a subset relation is *the* boundary of another set. See [saf_declare_subset_relation](#) for more information.

See Also:

- [saf_declare_subset_relation](#): 12.5: *Declare a subset relation*
- [Datatypes](#): Introduction for current chapter

Indexing scheme

SAF_CORDER is a macro defined in saf.h.

Synopsis:

SAF_CORDER (N)

Description: C order array of N dimensions. (See **Constants**)

See Also:

- [Datatypes](#): Introduction for current chapter

Library properties

`SAF_DEFAULT_LIBPROPS` is a symbol defined in `SAFlibprops.h`.

Synopsis:

`SAF_DEFAULT_LIBPROPS`

Description: Identifiers for default properties for the library.

See Also:

- *Datatypes*: Introduction for current chapter

Decomposition tri-state

`SAF_DecompMode` is an enumerated type defined in `saf.h`.

Synopsis:

`SAF_DECOMP_FALSE:`

`SAF_DECOMP_TRUE:`

`SAF_DECOMP_TORF:`

Description: To make each function call made by a client a little more self-documenting, we provide specific tri-state tags to represent the meaning of that particular boolean. The one here is used to indicate whether a given collection is a decomposition of its containing set.

See Also:

- *Datatypes*: Introduction for current chapter

Error return modes

`SAF_ErrMode` is an enumerated type defined in `SAFlibprops.h`.

Synopsis:

`SAF_ERRMODE_RETURN:` (The default) Library will issue return codes rather than throw exceptions

`SAF_ERRMODE_THROW:` Library will throw exceptions rather than issue return codes

Description: see `saf_setProps_ErrMode`

See Also:

- *Datatypes*: Introduction for current chapter

Evaluation Types

`SAF_EvalConstants` is a collection of related C preprocessor symbols defined in `SAFevaluation.h`.

Synopsis:

`SAF_SPACE_CONSTANT:` identifies an evaluation method that is constant. This is really just an alias for piecewise constant in which there is only one piece.

`SAF_SPACE_PWCONST:` identifies an evaluation method that is piecewise constant. That is it is constant over each piece in the `EVAL_COLL` argument of *`saf_declare_field`*.

SAF_SPACE_PWLINER: identifies an evaluation method that is piecewise linear.

SAF_SPACE_UNIFORM: identifies an evaluation method that is a single piece of linear evaluation such as is common for *uniform* coordinate fields.

Description: SAF currently supports specification of a field's evaluation method by picking from a list of known methods. Currently, that list is relatively short. SAF provides tags for specifying constant, piecewise linear and piecewise constant evaluations of a field.

Eventually, this list of evaluation methods will be expanded to include many of the common spline, and spectral evaluation schemes and they will also be user-definable. However, in this first implementation of SAF, we provide only an enumeration of the most commonly used evaluation methods.

See Also:

- [Datatypes](#): Introduction for current chapter

Extendable set tri-state

SAF_ExtendMode is an enumerated type defined in saf.h.

Synopsis:

SAF_EXTENDIBLE_FALSE:

SAF_EXTENDIBLE_TRUE:

SAF_EXTENDIBLE_TORF:

Description: To make each function call made by a client a little more self-documenting, we provide specific tri-state tags to represent the meaning of that particular boolean. The one here is used to indicate whether a set is extendible or not. See [saf_declare_set](#) for more information.

See Also:

- [saf_declare_set](#): 9.3: *Declare a set*
- [Datatypes](#): Introduction for current chapter

Indexing scheme

SAF_FORDER is a macro defined in saf.h.

Synopsis:

SAF_FORDER (N)

Description: Fortran order array of N dimensions. (See **Constants**)

See Also:

- [Datatypes](#): Introduction for current chapter

Set find modes

SAF_FindSetMode is an enumerated type defined in saf.h.

Synopsis:

SAF_FSETS_TOP: find the top-level from the given set

SAF_FSETS_BOUNDARY: find the boundary of the given set

SAF_FSETS_SUBS: find the immediate subsets of the given set

SAF_FSETS_SUPS: find the immediate supersets of the given set

SAF_FSETS_LEAVES: find all the bottom most sets in the tree rooted at the given set

Description: These are the possible modes that `saf_find_set` can operate in.

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_IndexSchema is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

SAF_F_ORDER:

SAF_C_ORDER:

Description: Macros for dealing with common indexing schema (Fortran and C 1,2 and 3D arrays).

See Also:

- *Datatypes*: Introduction for current chapter

Field component interleave modes

SAF_Interleave is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

SAF_BLOCKED: An alias for `SAF___INTERLEAVE_COMPONENT`.

SAF_INTERLEAVED: An alias for `SAF___INTERLEAVE_VECTOR`.

Description: When fields have multiple components, the components can be stored in the field's blob in different ways relative to each other. For example, in a 3D coordinate field, we will have 3 components for the x, y and z components of each coordinate. These can be stored as three different component fields or as a single composite field. If they are stored as a single composite field, they may be stored interleaved or non-interleaved.

The `SAF___INTERLEAVE_`*`` constants are defined by the ```ss_interleave_t` enumeration type. In addition we define aliases `SAF___BLOCKED` and `SAF___INTERLEAVED`.

See Also:

- *Datatypes*: Introduction for current chapter

Indexing scheme

SAF_NA_INDEXSPEC is a symbol defined in `saf.h`.

Synopsis:

SAF_NA_INDEXSPEC

Description: Not applicable index scheme. (See **Constants**)

See Also:

- *Datatypes*: Introduction for current chapter

Not applicable

SAF_NOT_APPLICABLE_INT is a symbol defined in saf.h.

Synopsis:

SAF_NOT_APPLICABLE_INT

Description: This is used for arguments of type int that aren't applicable in the current context

See Also:

- *Datatypes*: Introduction for current chapter

Not implemented

SAF_NOT_IMPL is a symbol defined in saf.h.

Synopsis:

SAF_NOT_IMPL

Description: This is used in parts of the API that are not implemented yet.

See Also:

- *Datatypes*: Introduction for current chapter

Associating a role with a collection category

SAF_RoleConstants is a collection of related C preprocessor symbols defined in SAFrole.h.

Synopsis:

SAF_TOPOLOGY: This role is associated with collection categories whose purpose is to knit the fine grained *topology* of the mesh together.

SAF_PROCESSOR: This role is associated with collection categories whose purpose is to represent different processor's pieces

SAF_BLOCK: This role is associated with collection categories whose purpose is to represent different *blocks* (regions of homogenous cell type)

SAF_DOMAIN: This role is associated with collection categories whose purpose is to represent different domains; fundamental quanta of a mesh that can be assigned to or, perhaps, migrate between, different processors.

SAF_ASSEMBLY: This role is associated with collection categories whose purpose is to represent parts in an assembly of parts.

SAF_MATERIAL: This role is associated with collection categories whose purpose is to represent materials.

SAF_SPACE_SLICE:

SAF_PARAM_SLICE:

SAF_ANY_ROLE: Wildcard role for searching

Description: The Role object is used in calls to *saf_declare_category* to associate a *role* with a collection category. We use the role of a collection category to hint at the purpose or intent of collections created of a given category. Some collections are used to represent processor pieces. Some are used to knit individual computational elements together into a mesh. Some are used to represent different materials, etc. The list of roles here is by no means complete.

Issues: It is unclear whether any routines in **SAF** will be or ought to be sensitive to the value of *role* or whether **SAF** simply passes the role around without ever interpreting it. There are two clear cases in which **SAF** itself might need to interpret the role; topology and boundary information. It might also be useful if **SAF** could interpret the processor role as this could help to make **SAF** knowledgeable about what pieces of the mesh are on which processors.

See Also:

- *saf_declare_category*: 10.2: *Declare a collection category*
- *Datatypes*: Introduction for current chapter

Subset inclusion lattice roles

SAF_SilRole is a collection of related C preprocessor symbols defined in saf.h.

Synopsis:

SAF_TIME: For sets specifying pieces of time

SAF_SPACE: For sets specifying pieces of space

SAF_PARAM: For sets specifying pieces of some arbitrary, user defined parameter space

SAF_SUITE: for sets specifying whole suites

SAF_ANY_SILROLE: Wildcard role for searching

Description: Every subset inclusion lattice defines pieces of some all-encompassing space in which those pieces live. For example, the lattice may be specifying pieces of the time-base, or pieces of space, or pieces of some user defined parameter space.

In future versions of **SAF**, this information will be supplanted by the quantity associated with the coordinate field for a given base space.

See Also:

- *Datatypes*: Introduction for current chapter

String allocation modes

SAF_StrMode is an enumerated type defined in SAFlibprops.h.

Synopsis:

SAF_STRMODE_LIB: library allocates **but** client frees (zero is the default)

SAF_STRMODE_CLIENT: client allocates **and** client frees

SAF_STRMODE_POOL: library allocates **and** library frees using a least recently used strategy involving a pool of many strings

Description: see *saf_setProps_StrMode*

See Also:

- *saf_setProps_StrMode*: 5.9: *Set string allocation style*
- *Datatypes*: Introduction for current chapter

Subset relation representation types

SAF_SubsetRelRep is a collection of related C preprocessor symbols defined in SAFrelrep.h.

Synopsis:

SAF_HSLAB: Indicates a hyperslab which is stored as 3 N-tuples; N indices for the start value in each of the N dimensions, followed by N indices for the count in each of the N dimensions followed by N indices for stride in each of the N dimensions. Use a stride of 1 for each of the N dimensions if you do **not** have a hypersample.

SAF_TUPLES: Indicates a list of N-tuples. Each N-tuple identifies one member of an N dimensionally indexed collection.

SAF_TOTALITY: Indicates that all members of the collection are involved—which probably also means the subset is equal to the superset. Perhaps a better name for this value would be SAF__IDENTITY. However, that is being used elsewhere. Typically, this value is only ever used during a *saf_write_field* call.

Description: The subset relationship between a superset and a subset can take many forms. In theory, the subset relation identifies every member of the superset that is **in** the subset. In practice, depending on the nature of the indexing schemes used to identify members of collections on the superset and subset, there are a number of different ways a client may *represent* a subset relationship. In an unstructured gridded code, the natural thing to do is simply enumerate each member of the superset in the subset by listing them. In a structured gridded code, the natural approach is to specify a hyperslab (or hypersample). Another natural approach for a structured gridded code is to specify a chain-code boundary where everything surrounded by the boundary is in the subset. This latter form is **not yet** supported by SAF.

Issues: These representational issues raise a more fundamental question. Is the act of defining a subset one of enumerating **every** point of the superset that is in the subset or can it also be achieved by enumerating a boundary in the superset where everything *inside* the boundary is in the subset? In other words, do we deal only with solid representations or both solid and boundary representations for sets?

We do **not** support a list of hyperslabs (hypersamples) due to the confusion of this representation with the union of a number of individual sets which are hyperslab subsets of some parent superset.

See Also:

- *Datatypes*: Introduction for current chapter

Top mode tri-state

SAF_TopMode is an enumerated type defined in saf.h.

Synopsis:

SAF_TOP_FALSE:

SAF_TOP_TRUE:

SAF_TOP_TORF:

Description: To make each function call made by a client a little more self-documenting, we provide specific tri-state tags to represent the meaning of that particular boolean. The one here is used to limit a search to top level sets in a *saf_find_matching_sets* call.

See Also:

- *saf_find_matching_sets*: 9.5: Find set by matching criteria
- *Datatypes*: Introduction for current chapter

Topological dimensions

`SAF_TopoDim` is an enumerated type defined in `saf.h`.

Synopsis:

`SAF_TOPODIM_0D`: a zero dimensional topological dimension (e.g. a point)

`SAF_TOPODIM_1D`: a one dimensional topological dimension (e.g. a curve)

`SAF_TOPODIM_2D`: a two dimensional topological dimension (e.g. a surface)

`SAF_TOPODIM_3D`: a three dimensional topological dimension (e.g. a volume)

Description: These are really just more informative aliases for the numbers 0, 1, 2 and 3 so that when these are seen in `saf` function calls, the purpose of the argument will be more clear.

See Also:

- *Datatypes*: Introduction for current chapter

Relation representation types

`SAF_TopoRelRep` is a collection of related C preprocessor symbols defined in `SAFrelrep.h`.

Synopsis:

`SAF_STRUCTURED`: N-dimensional rectangular topology

`SAF_UNSTRUCTURED`: unstructured, finite element zoo topology

`SAF_ARBITRARY`: arbitrary topology

Description: There are three basic classes of topology supported by `SAF`; N-dimensional rectangular structured topology, unstructured, finite element zoo topology and completely arbitrary topology. These three tags are used to define which class is being used in a `saf_declare_topology_relation` call.

In future versions of `SAF`, user defined cell types will be supported. Thus, the *zoo* from which element types are used in defining topology will eventually be filled with whatever cell-types the client needs.

Also, in future versions of `SAF`, structured topology will be represented by a structuring template similar to the notion of a stencil in finite difference computations. This would permit the characterization of hexagonal grids, triangle-strips, etc.

See Also:

- *Datatypes*: Introduction for current chapter

Standard tri-state values

`SAF_TriState` is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

`SAF_TRISTATE_FALSE`:

`SAF_TRISTATE_TRUE`:

`SAF_TRISTATE_TORF`:

Description: In many portions of `SAF`'s API, there are boolean values to indicate if a particular feature of an object is true or false. In addition, it is possible to invoke searches using `saf_find...` kinds of functions that will search for

objects for which the given boolean feature is true or false or either. So, we've defined a standard tri-state enumeration for these three cases.

See Also:

- *Datatypes*: Introduction for current chapter

NULL aliases

`SAF_VoidPtr` is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

`SAF_IDENTITY:`

`SAF_NO_COMPONENTS:`

`SAF_NO_DATA:`

Description: A bunch of useful aliases for 'NULL'

See Also:

- *Datatypes*: Introduction for current chapter

Return codes

`SAF_return_t` is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

`SAF_FAILURE:`

`SAF_SUCCESS:`

Description: Not written yet.

See Also:

- *Datatypes*: Introduction for current chapter

Predefined scalar datatypes

`SAF_type_t` is a collection of related C preprocessor symbols defined in `saf.h`.

Synopsis:

`SAF_CHAR`: Character datatype.

`SAF_INT`: Integer datatype.

`SAF_LONG`: Long integer datatype.

`SAF_FLOAT`: Single-precision floating-point datatype.

`SAF_DOUBLE`: Double-precision floating-point datatype.

`SAF_HANDLE`: Object handle datatype.

Description: These C preprocessor symbols represent various *SAF* predefined scalar datatypes. (See **Constants**)

Issues: These constants are mostly for backward compatibility. *SAF* now uses HDF5's datatype interface instead of the DSL interface. Applications should eventually switch over to HDF5 constants.

See Also:

- *Datatypes*: Introduction for current chapter

Notes

Miscellaneous notes.

Members

Constants

Many constants defined in `saf.h` have similar limitations to some C programming language constants such as `'stderr'` and `'stdout'`. If you have a file-scope static variable initialized to `stderr` or `stdout`, you will find that your variable gets initialized with garbage. In C, the reason is that these constants don't really get defined **until run-time** and when you initialize a file-scope static, you are relying upon it having been defined **at compile time**.

The same is true for many of **SAF's** constants. Therefore, you should take care **not to use them** in a manner which assumes they are defined at compile time. We have made an effort to denote all such constants in the reference manual so that you can easily determine for which constants this is true.

Some constants require a database argument. This means the constant is not defined except within the scope of an open database. Thus, these constants are even more restricted in use than those that can be used at any run-time.

In summary, there are three classes of constants. Compile-time constants can be used anywhere. Run-time constants can be used only after the code has begun executing. Database-time constants can be used only after a database has been opened.

Properties

A *property* abstraction is used to control various behaviors of the library and its interactions with databases, supplemental files and other things. A specific set of properties is constructed **prior** to a scope-beginning call in which those properties are applied. For example, since library properties effect the behavior of the library, they are applied in the *saf_init* call. Likewise, database properties are applied in the *saf_open_database* call. Once applied, the properties remain in effect until a corresponding scope-ending call. For example, *saf_final* for library properties or *saf_close_database* for database properties. You cannot change properties mid-stream. If this is a desired modality, please contact saf-help@sourceforge.sandia.gov with such a request.

The general way to control properties is to build up the desired property set of properties by first calling a `saf_createProps_xxx` function where 'xxx' is, for example, 'lib' or 'database'. This creates a default set of properties. You can then adjust specific properties in this set by calling individual functions described in the properties chapters of the API. The resultant set of properties is applied when they are passed in a scope-beginning call such as *saf_init* or *saf_open_database*. See descriptions of the individual member property functions for a description of the properties supported.

Algebraic Types

No description available.

Members

Common algebraic types

SAF_ALGTYPE is a collection of related C preprocessor symbols defined in SAFalgebraic.h.

Synopsis:

SAF_ALGTYPE_SCALAR: Used to specify fields that obey properties of scalar algebra.

SAF_ALGTYPE_VECTOR: Used, generically, for fields that obey properties of vector algebra.

SAF_ALGTYPE_COMPONENT: Used, generically, for any component of a multi-component field. In many cases, it might be just as well to treat each component of a multi-component field as a scalar field. However, this is not entirely mathematically correct.

SAF_ALGTYPE_TENSOR: Used for general, non-symmetric tensor fields

SAF_ALGTYPE_SYMTENSOR: Used for general, symmetric tensor fields.

SAF_ALGTYPE_TUPLE: Used to identify a field which evaluates to a *group* of otherwise unrelated fields. Typically used in a *State* field.

SAF_ALGTYPE_FIELD: This algebraic type is used for fields that are, in reality, simply *references* to other fields. These are called *field indirections* or, *indirect*fields*. *Indirect fields are used, primarily for two kinds of fields; *inhomogeneous fields and cross-product fields*. An inhomogeneous field is represented as references to pieces of the field over subsets of its base-space over which each piece is homogenous. Likewise, a cross-product field is used to work around the fact that SAF does NOT deal with cross product sets in the base-spaces of fields. Thus, we represent such fields as references to fields over other base spaces.

SAF_ALGTYPE_ANY: Wildcard for find operations.

Description: SAF supports the characterization of various algebraic types for fields. An algebraic type specifies the algebraic properties of the field.

We should probably identify an algebraic type for a *Barycentric* field; a field whose components are between 0.0 and 1.0 and which sum to 1.0.

See Also:

- *Algebraic Types*: Introduction for current chapter

Declare a new algebraic type

saf_declare_algebraic is a function defined in algebraic.c.

Synopsis:

SAF_Algebraic * **saf_declare_algebraic** (SAF_ParMode *pmode*, SAF_Db **db*, const char **name*, const char **url*, hbool_t *indirect*, SAF_Algebraic **alg*)

Formal Arguments:

- *db*: The database in which to create the new algebraic type
- *name*: Name of the algebraic type
- *url*: An optional *url* to the algebraic documentation
- *indirect*: If true then field is indirection to another field
- *alg*: [OUT] Optional handle to initialize (and return)

Description: This function declares a new algebraic type with a unique identification number.

Return Value: A handle to the new algebraic type.

See Also:

- *Algebraic Types*: Introduction for current chapter

Describe an algebraic type

`saf_describe_algebraic` is a function defined in `algebraic.c`.

Synopsis:

```
int saf_describe_algebraic (SAF_ParMode pmode, SAF_Algebraic *alg, char **name, char **url,  
                           hbool_t *indirect)
```

Formal Arguments:

- `alg`: Algebraic to describe
- `name`: If non-null, on return points to malloc'd algebraic name if any
- `url`: If non-null, on return points to malloc'd url if any
- `indirect`: If non-null, on return points to non-zero if type is indirect

Description: Breaks `ALGEBRAIC` into its parts and returns them through pointers.

Preconditions:

- `alg` must be a valid algebraic handle. (low-cost)

Return Value: A non-negative value indicates success, and a negative value indicates failure. On return, the output arguments `name`, `url`, `indirect`, and `ID` will be initialized.

See Also:

- *Algebraic Types*: Introduction for current chapter

Find algebraic types

`saf_find_algebraics` is a function defined in `algebraic.c`.

Synopsis:

```
int saf_find_algebraics (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *url,  
                        htri_t indirect, int *num, SAF_Algebraic **found)
```

Formal Arguments:

- `db`: Database in which to limit the search.
- `name`: Optional name for which to search.
- `url`: Optional url for which to search.
- `indirect`: Optional indirect flag for which to search. The caller should pass a negative value if it is not interested in restricting the search.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function allows a client to search for algebraic types in the database. The search may be limited by one or more criteria such as the name of the algebraic type, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`

See Also:

- *Algebraic Types*: Introduction for current chapter

Find one algebraic type

`saf_find_one_algebraic` is a function defined in `algebraic.c`.

Synopsis:

```
SAF_Algebraic * saf_find_one_algebraic (SAF_Db      *database,      const   char      *name,
                                       SAF_Algebraic *buf)
```

Formal Arguments:

- `database`: The database in which to search
- `name`: The name for which to search
- `buf`: [OUT] Optional algebraic handle to initialize and return

Description: This is a convenience version of `saf_find_algebraic` that returns the first algebraic type it finds whose name matches that which is specified.

Return Value: A handle to a matching algebraic type on success; `SAF__ERROR_HANDLE` on failure.

See Also:

- *Algebraic Types*: Introduction for current chapter

Alternative Index Specification

The indexing specification of a collection is a characterization, more generally, of the name space used to identify members of the collection. For example, we might choose to refer to the members of a collection of 4 quads, `[Q, "Q", "Q", "Q"]`, using any of the following schemes:

a: 0,1,2,3

b: "Larry", "Mark", "Peter", "Ray"

c: 27, 13, 102, 77

d: 14, 36, 37, 92

e: (0,0), (0,1), (1,0), (1,1)

f: 0x00000000, 0x00000001, 0x00010000, 0x00010001

The *a* scheme might be considered the “default” or “natural” naming scheme. *b* is a naming scheme based upon strings. *c* is a naming scheme based upon some arbitrary integer enumeration. Likewise for *d*. *e* is a naming scheme based upon rectangular indexing. *f* is a naming scheme that might be used in a pyramid of resolution of quads with 16 or fewer layers in which a 32 bit quantity is broken into two 16-bit pieces, one for the row and column of each layer in the pyramid.

Some observations about these naming schemes. In some, *a*, *e* and *f* there is an easily specified rule for generating the names. In the others, the names must be explicitly enumerated. In some, *a*, *b*, *d*, *e* and *f* the names are sorted. In some, *a*, *e* and *f*, the names are “compact” meaning that given the names of any two successive members, there is no name that can be drawn from the same pool from which the other names come that falls between them.

From these observations, we conclude that an indexing spec can be either implicit or explicit. An implicit spec is one in which there is a simple rule for constructing each id in the name space. An explicit indexing spec is one in which each id in the name space must be explicitly specified. In addition, for an explicit spec, we also need to know if the names are sorted (and maybe even how to sort them by virtue of a call-back function to compare names), and if the names are compact.

SAF’s notion of an *indexing specification* should be evolved to include these notions. Nonetheless, immediate support for user-defined IDs is essential. Therefore, we have provided functions in SAF for a client to specify *alternative indexing* specifications for a given collection. These functions will permit a SAF client to declare/describe and write/read alternative IDs. However, all relations involving the collection must still be specified in terms of the default indexing. Later, we can enhance the relations interface for SAF to support a client that specifies its relations in terms of these alternative IDs.

Implementation Details

These are details that are probably of no concern to the general user. This info is for someone who cares about the lower levels of SAF and how Alternative Indexing was implemented. The two SAF object data types SAF__IndexSpec and SAF__AltIndexSpec both map to the same SSlib object, namely `ss_indexspec_t`. Every collection record has a variable length array of links to `ss_indexspec_t` objects. The first item in that array is the default index spec for that collection. If there are any alternate index specs for a collection, typically there would be only one, since these would be the one way that the client refers to their node ids (or elem ids or face ids, etc).

Members

Declare an Alternative Index Specification

`saf_declare_alternate_indexspec` is a function defined in `altindx.c`.

Synopsis:

```
SAF_AltIndexSpec * saf_declare_alternate_indexspec (SAF_ParMode pmode, SAF_Db *db,
                                                    SAF_Set          *containing_set,
                                                    SAF_Cat   *cat,  const char *name,
                                                    hid_t   data_type, hbool_t is_explicit,
                                                    SAF_IndexSpec      implicit_ispec,
                                                    hbool_t is_compact, hbool_t is_sorted,
                                                    SAF_AltIndexSpec *aspec)
```

Formal Arguments:

- `pmode`: The parallel mode
- `db`: Database to contain the new index spec.
- `containing_set`: The containing set of the collection.
- `cat`: The collection category.

- `name`: The name you wish to assign to this alt index spec
- `data_type`: The data type used to identify members of the collection
- `is_explicit`: Whether the indexing specification is explicit or implicit
- `implicit_ispec`: The alternate indexing scheme of the collection. Ignored for explicit specs. Pass `SAF__NA_INDEXSPEC` for explicit alternative index specs.
- `is_compact`: Whether the indexing specification is compact or not. Ignored for implicit specs.
- `is_sorted`: Whether the indexing specification is sorted or not. Ignored for implicit specs.
- `aspec`: [OUT] The optional returned alternate index spec handle. If the null pointer is passed for this argument then new memory is allocated and returned, otherwise this argument serves as the successful return value.

Description: There is already a default `SAF__IndexSpec` associated with the collection defined by `containing_set` and `cat`. This call registers another, alternate index specification. The default index spec associated with the collection is something that allows you to describe the collection IDs very easily by specifying the start index and how many you have (typically the start index is 0). If you have some other, arbitrary way to identify the members of the collection, then you need to write out a problem sized array describing the names you give to the members of that collection. This is an explicit alternate indexing scheme, since you need to explicitly list the id's for each member of the collection. An implicit index spec is something that can be captured by stating the start index and how many you have, so you don't need to explicitly list the collection ids.

Preconditions:

- `pmode` must be valid. (low-cost)
- `containing_set` must be a valid set handle. (low-cost)
- `cat` must be a valid cat handle. (low-cost)

Return Value: On success, returns either the `aspec` argument or a newly allocated index specification. Returns the null pointer on failure.

Issues: The `data_type` is just stored as the HDF5 `data_type` member of the `SAF__AltIndexSpec`. This is transient, in memory data, it is not written to the saf database until the `saf_write_alternate_indexspec` call. This means that if you do something like: `saf_declare_alternate_indexspec`, then `saf_find_alternate_index_spec`, then `saf_describe_alternate_indexspec`, (with no write call yet) you won't be able to recover the `data_type`.

See Also:

- *Alternative Index Specification*: Introduction for current chapter

Get a description of an alternate indexing spec

`saf_describe_alternate_indexspec` is a function defined in `altindx.c`.

Synopsis:

```
int saf_describe_alternate_indexspec (SAF_ParMode   pmode,   SAF_AltIndexSpec *as-
                                     pec,   SAF_Set   *containing_set,   SAF_Cat   *cat,
                                     char **name, hid_t *data_type, hbool_t *is_explicit,
                                     SAF_IndexSpec *implicit_ispec, hbool_t *is_compact,
                                     hbool_t *is_sorted)
```

Formal Arguments:

- `pmode`: The parallel mode
- `aspec`: The alternate index spec you want the description of.
- `containing_set`: [OUT] The containing set of the collection. Pass `NULL` if you do not want this returned.

- `cat`: [OUT] The collection category. Pass `NULL` if you do not want this returned.
- `name`: [OUT] The name of this alt index spec.
- `data_type`: [OUT] The data type used to identify members of the collection. Pass `NULL` if you do not want this returned.
- `is_explicit`: [OUT] Whether the indexing specification is explicit or implicit.
- `implicit_ispec`: [OUT] The alternate indexing scheme of the collection. If the index spec is explicit, then `SAF__NA_INDEXSPEC` will be returned. If the index spec is implicit, the implicit index spec will be returned here.
- `is_compact`: [OUT] Whether the indexing specification is compact or not. Ignored for implicit specs.
- `is_sorted`: [OUT] Whether the indexing specification is sorted or not. Ignored for implicit specs.

Description: Get a description of an alternate indexing spec

Preconditions:

- `pmode` must be valid. (low-cost)
- The `aspec` argument must be a valid handle. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Alternative Index Specification*: Introduction for current chapter

Find alternate index specs by matching criteria

`saf_find_alternate_indexspecs` is a function defined in `altindx.c`.

Synopsis:

```
int saf_find_alternate_indexspecs (SAF_ParMode   pmode,   SAF_Set      *containing_set,
                                   SAF_Cat      *cat,   const char *name_grep, int *num,
                                   SAF_AltIndexSpec **found)
```

Formal Arguments:

- `pmode`: The parallel mode
- `containing_set`: The containing set of the collection.
- `cat`: The collection category.
- `name_grep`: The name of the alt index spec you wish to search for. Pass `NULL` if you do not wish to limit the search via a name.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: Find alternate index specs by matching criteria.

If the `name_grep` argument begins with a leading “at sign” character, ‘@’, the remaining characters will be treated as a limited form of a regular expression akin to that supported in ‘ed’. Otherwise, it will be treated as a specific name for a set. If the name does not matter, pass `SAF__ANY_NAME`.

If the library was not compiled with -lgen support library, then if regular expressions are used, the library will behave as though `SAF__ANY_NAME` was specified.

Preconditions:

- `pmode` must be valid. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: This function does not follow the usual semantics of a *find* operation. Instead of searching through a database (or scope) and looking for index specifications that match a certain pattern, it instead looks at a collection (specified with the `containing_set` and `cat` arguments) and returns any index specifications of that collection that have the requested name or name pattern.

See Also:

- *Alternative Index Specification*: Introduction for current chapter

Read an alternate index specs from disk

`saf_read_alternate_indexspec` is a function defined in `altindx.c`.

Synopsis:

```
int saf_read_alternate_indexspec (SAF_ParMode pmode, SAF_AltIndexSpec *aspec, void **buf)
```

Formal Arguments:

- `pmode`: The parallel mode.
- `aspec`: The alternate index spec handle to read.
- `buf`: The buffer to be filled in with the data.

Description: Read an alternate index specs from disk, involves actual I/O

Preconditions:

- `pmode` must be valid. (low-cost)
- `aspec` must be a valid alt index spec handle. (low-cost)
- `buf` cannot be null. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Issues: There is no way for the caller to find out what datatype is being returned.

See Also:

- *Alternative Index Specification*: Introduction for current chapter

Write an alternate index specs to disk

`saf_write_alternate_indexspec` is a function defined in `altindx.c`.

Synopsis:

```
int saf_write_alternate_indexspec (SAF_ParMode pmode, SAF_AltIndexSpec *aspec,  
hid_t data_type, void *buf, SAF_Db *file)
```

Formal Arguments:

- `pmode`: The parallel mode.

- `aspec`: The alternate index spec to write.
- `data_type`: The datatype used to identify members of the collection, if not already supplied with [*saf_declare_alternate_indexspec*](#).
- `buf`: The buffer of data to write.
- `file`: The optional destination file to which to write the data. If this is a null pointer then the data is written to the same file as `aspec`.

Description: Write an alternate index specs to disk, involves actual I/O

Preconditions:

- `pmode` must be valid. (low-cost)
- `aspec` must be a valid alternate index spec handle. (low-cost)
- `buf` must not be null. (low-cost)
- You must pass a datatype either in the call to [*saf_declare_alternate_indexspec*](#) or here. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- [*saf_declare_alternate_indexspec*](#): 30.1: *Declare an Alternative Index Specification*
- [*Alternative Index Specification*](#): Introduction for current chapter

Basis Types

No description available.

Members

Declare a new basis type

`saf_declare_basis` is a function defined in `basis.c`.

Synopsis:

```
SAF_Basis * saf_declare_basis (SAF_ParMode pmode, SAF_Db *db, const char *name, const  
                             char *url, SAF_Basis *basis)
```

Formal Arguments:

- `name`: Name of the basis type
- `url`: An optional `url` to the basis documentation
- `basis`: [OUT] Optional basis handle to initialize (and return).

Description: This function declares a new basis type with a unique identification number.

Return Value: A handle to the new basis type.

See Also:

- [*Basis Types*](#): Introduction for current chapter

Describe a basis type

`saf_describe_basis` is a function defined in `basis.c`.

Synopsis:

```
int saf_describe_basis (SAF_ParMode pmode, SAF_Basis *basis, char **name, char **url)
```

Formal Arguments:

- `basis`: Basis to describe
- `name`: [OUT] If non-null, on return points to malloc'd basis name if any
- `url`: [OUT] If non-null, on return points to malloc'd url if any

Description: Breaks `basis` into its parts and returns them through pointers.

Preconditions:

- `basis` must be a valid basis handle. (low-cost)

Return Value: A non-negative value indicates success, and a negative value indicates failure. On return, the output arguments `name`, `url`, and `ID` will be initialized.

See Also:

- [Basis Types](#): Introduction for current chapter

Find bases

`saf_find_bases` is a function defined in `basis.c`.

Synopsis:

```
int saf_find_bases (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *url, int *num,  
                   SAF_Basis **found)
```

Formal Arguments:

- `db`: Database in which to limit the search.
- `name`: Optional name to which to limit the search.
- `url`: Optional url to which to limit the search.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function allows a client to search for bases in the database. The search may be limited by one or more criteria such as the name of the basis, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`.

See Also:

- *Basis Types*: Introduction for current chapter

Find one basis type

`saf_find_one_basis` is a function defined in `basis.c`.

Synopsis:

`SAF_Basis * saf_find_one_basis (SAF_Db *database, const char *name, SAF_Basis *buf)`

Formal Arguments:

- `database`: The database in which to search
- `name`: The name for which to search
- `buf`: [OUT] Optional basis handle to initialize and return.

Description: This is a convenience version of `saf_find_basis` that returns the first basis it finds whose name matches that which is specified.

Return Value: A handle to a matching basis on success; `SAF__ERROR_HANDLE` on failure.

See Also:

- *Basis Types*: Introduction for current chapter

Collection Roles

No description available.

Members

Declare a new collection role

`saf_declare_role` is a function defined in `role.c`.

Synopsis:

`SAF_Role * saf_declare_role (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *url,
SAF_Role *role)`

Formal Arguments:

- `pmode`: The parallel mode
- `db`: The database in which to create the new role
- `name`: Name of the role
- `url`: An optional `url` to the role documentation
- `role`: [OUT] Optional role handle to initialize (and return)

Description: This function declares a new collection role with a unique identification number.

Return Value: A handle to the new role.

See Also:

- *Collection Roles*: Introduction for current chapter

Describe a role

`saf_describe_role` is a function defined in `role.c`.

Synopsis:

```
int saf_describe_role (SAF_ParMode pmode, SAF_Role *role, char **name, char **url)
```

Formal Arguments:

- `role`: Role to describe
- `name`: If non-null, on return points to malloc'd role name if any
- `url`: If non-null, on return points to malloc'd url if any

Description: Breaks `role` into its parts and returns them through pointers.

Preconditions:

- `role` must be a valid role handle. (low-cost)

Return Value: A non-negative value indicates success, and a negative value indicates failure. On return, the output arguments `name`, `url`, and `ID` will be initialized.

See Also:

- [Collection Roles](#): Introduction for current chapter

Find one collection role

`saf_find_one_role` is a function defined in `role.c`.

Synopsis:

```
SAF_Role * saf_find_one_role (SAF_Db *database, const char *name, SAF_Role *buf)
```

Formal Arguments:

- `database`: The database in which to search
- `name`: The name for which to search
- `buf`: [OUT] Optional role handle to initialize and return.

Description: This is a convenience version of [saf_find_roles](#) that returns the first role it finds whose name matches that which is specified.

Return Value: A handle to a matching role on success; `SAF__ERROR_HANDLE` on failure.

See Also:

- [saf_find_roles](#): 32.4: *Find roles*
- [Collection Roles](#): Introduction for current chapter

Find roles

`saf_find_roles` is a function defined in `role.c`.

Synopsis:

```
int saf_find_roles (SAF_ParMode pmode, SAF_Db *db, const char *name, char *url, int *num,  
                   SAF_Role **found)
```

Formal Arguments:

- `db`: Database in which to limit the search.
- `name`: Optional name to which to limit the search.
- `url`: Optional `url` to which to limit the search.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: This function allows a client to search for roles in the database. The search may be limited by one or more criteria such as the name of the role, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

See Also:

- *Collection Roles*: Introduction for current chapter

Data Types

No description available.

Members**Convert a single value**

`_saf_convert` is a function defined in `utils.c`.

Synopsis:

```
void * _saf_convert (hid_t srctype, const void *srcbuf, hid_t dsttype, void *dstbuf)
```

Formal Arguments:

- `srctype`: Source datatype; type of `srcbuf` value.
- `srcbuf`: Source datum to be converted to a new type.
- `dsttype`: Destination datatype; type of `dstbuf` value.
- `dstbuf`: Optional destination buffer. If not supplied then a buffer is allocated.

Description: Converts a single value from one datatype to another. This is most often used to convert a runtime typed value into an integer to be used by the library.

Return Value: Returns `dstbuf` (or an allocated buffer if `dstbuf` is null) on success; returns null on failure.

Parallel Notes: Independent

See Also:

- *Data Types*: Introduction for current chapter

Determine if datatype is primitive

`_saf_is_primitive_type` is a function defined in `utils.c`.

Synopsis:

`hbool_t _saf_is_primitive_type (hid_t type)`

Description: A “primitive” datatype is anything that’s an integer or floating point type.

Return Value: Returns true if `type` is primitive; false otherwise.

Parallel Notes: Independent

See Also:

- [Data Types](#): Introduction for current chapter

Evaluation Types

No description available.

Members

Declare a new evaluation type

`saf_declare_evaluation` is a function defined in `evaluation.c`.

Synopsis:

`SAF_Eval * saf_declare_evaluation (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *url, SAF_Eval *buf)`

Formal Arguments:

- `name`: Name of the evaluation type
- `url`: An optional `url` to the evaluation documentation
- `buf`: [OUT] Optional buffer to fill in and return

Description: This function declares a new evaluation type with a unique identification number.

Preconditions:

- `pmode` must be valid. (low-cost)

Return Value: A handle to the new evaluation type.

See Also:

- [Evaluation Types](#): Introduction for current chapter

Describe an evaluation type

`saf_describe_evaluation` is a function defined in `evaluation.c`.

Synopsis:

`int saf_describe_evaluation (SAF_ParMode pmode, SAF_Eval *evaluation, char **name, char **url)`

Formal Arguments:

- `evaluation`: Evaluation to describe
- `name`: If non-null, on return points to malloc'd evaluation name if any
- `url`: If non-null, on return points to malloc'd url if any

Description: Breaks `evaluation` into its parts and returns them through pointers.

Preconditions:

- `pmode` must be valid. (low-cost)
- `evaluation` must be a valid evaluation handle. (low-cost)

Return Value: A non-negative value indicates success, and a negative value indicates failure. On return, the output arguments `name`, `url`, and `ID` will be initialized.

See Also:

- *Evaluation Types*: Introduction for current chapter

Find evaluation types

`saf_find_evaluations` is a function defined in `evaluation.c`.

Synopsis:

```
int saf_find_evaluations (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *url,
                        int *num, SAF_Eval **found)
```

Formal Arguments:

- `db`: Database in which to limit the search.
- `name`: Optional name for which to search.
- `url`: Optional url for which to search.
- `num`: For this and the succeeding argument [see Returned Handles].
- `found`: For this and the preceding argument [see Returned Handles].

Description: The function allows the client to search for evaluation types in the database. The search may be limited by one or more criteria such as the name of the unit, etc.

Preconditions:

- `pmode` must be valid. (low-cost)
- `db` must be a valid database. (low-cost)
- `num` and `found` must be compatible for return value allocation. (low-cost)

Return Value: The constant `SAF__SUCCESS` is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on `pmode`

See Also:

- *Evaluation Types*: Introduction for current chapter

Find one evaluation type

`saf_find_one_evaluation` is a function defined in `evaluation.c`.

Synopsis:

SAF_Eval * **saf_find_one_evaluation** (SAF_Db **database*, const char **name*, SAF_Eval **buf*)

Formal Arguments:

- *database*: The database in which to search
- *name*: The name for which to search
- *buf*: [OUT] Optional buffer to fill in and return

Description: This is a convenience version of `saf_find_evaluation` that returns the first evaluation type it finds whose name matches that which is specified.

Return Value: A handle to a matching evaluation type on success; `SAF__ERROR_HANDLE` on failure.

See Also:

- *Evaluation Types*: Introduction for current chapter

Relation Representation Types

No description available.

Members

Declare a new object

`saf_declare_relrep` is a function defined in `relrep.c`.

Synopsis:

SAF_RelRep * **saf_declare_relrep** (SAF_ParMode *pmode*, SAF_Db **db*, const char **name*, const char **url*, int *id*, SAF_RelRep **buf*)

Formal Arguments:

- *db*: Database in which to declare the new relation representation
- *name*: Name of the object
- *url*: An optional *url* to the documentation
- *id*: A unique non-negative identification number
- *buf*: [OUT] Optional handle to fill in and return

Description: This function declares a new topology relation representation type with a unique identification number.

Preconditions:

- *pmode* must be valid. (low-cost)

Return Value: A handle to the new object.

See Also:

- *Relation Representation Types*: Introduction for current chapter

Describe an object

`saf_describe_relrep` is a function defined in `relrep.c`.

Synopsis:

```
int saf_describe_relrep (SAF_ParMode pmode, SAF_RelRep *obj, char **name, char **url, int *id)
```

Formal Arguments:

- `obj`: object to describe
- `name`: If non-null, on return points to malloc'd name if any
- `url`: If non-null, on return points to malloc'd url if any
- `id`: If non-null, on return points to unique id

Description: Breaks `obj` into its parts and returns them through pointers.

Preconditions:

- `pmode` must be valid. (low-cost)
- `obj` must be a valid relation representation handle. (low-cost)

Return Value: A non-negative value indicates success, and a negative value indicates failure. On return, the output arguments `name`, `url`, and `id` will be initialized.

See Also:

- [*Relation Representation Types*](#): Introduction for current chapter

Find one object

`saf_find_one_relrep` is a function defined in `relrep.c`.

Synopsis:

```
SAF_RelRep * saf_find_one_relrep (SAF_Db *database, const char *name, SAF_RelRep *buf)
```

Formal Arguments:

- `database`: The database in which to search
- `name`: The name for which to search
- `buf`: [OUT] Optional buffer to initialize and return

Description: This is a convenience version of `saf_find_trelrep` that returns the first object it finds whose name matches that which is specified.

Return Value: A handle to a matching object on success; negative on failure.

See Also:

- [*Relation Representation Types*](#): Introduction for current chapter

Find relation representation types

`saf_find_relreps` is a function defined in `relrep.c`.

Synopsis:

```
int saf_find_relreps (SAF_ParMode pmode, SAF_Db *db, const char *name, const char *url, int id,
                    int *num, SAF_RelRep **found)
```

Formal Arguments:

- db: Database in which to limit the search.
- name: Optional name for which to search.
- url: Optional url for which to search.
- id: Optional id for which to search, or pass SAF__ANY_INT.
- num: For this and the succeeding argument [see Returned Handles].
- found: For this and the preceding argument [see Returned Handles].

Description: This function allows a client to search for relation representation types in the database. The search may be limited by one or more criteria such as the name of the type, etc.

Preconditions:

- pmode must be valid. (low-cost)
- db must be a valid database. (low-cost)
- num and found must be compatible for return value allocation. (low-cost)

Return Value: The constant SAF__SUCCESS is returned when this function is successful. Otherwise this function either returns an error number or throws an exception, depending on the value of the library's error handling property.

Parallel Notes: Depends on pmode

See Also:

- *Relation Representation Types*: Introduction for current chapter

1.2 Sets and Fields (SAF) Examples and Use Cases

Acknowledgements

1.2.1 Table of Contents

Birth and Death Use Case

This is testing code that demonstrates how to use [SAF](#) to output a mesh in which elements are being created and destroyed over the course of a simulation. The bulk of the code here is used simply to create some interesting meshes and has nothing to do with reading/writing from/to [SAF](#). The only routines in which [SAF](#) calls are made are [main](#), [OpenDatabase](#), [WriteCurrentMesh](#), [UpdateDatabase](#) and [CloseDatabase](#). As such, these are the only *Public* functions defined in this file. In addition, the function to parse a sequence of addition/deletion, [GetAddDelSequence](#), steps is also public so that a user can see how to program this client to create a variety of interesting databases.

The test is designed to be relatively flexible in the dimension of mesh (topological and geomertic dimensions are bound together) and in the various steps it goes through creating and deleting elements. However, elements are created and deleted in slabs (e.g. planes of elements). There is a default dimension and sequence of steps hardcoded into this test client. In addition, this test client is designed to also accept input from a text file that encodes the dimensionality of the mesh and the series of steps of deletions and additions (see [GetAddDelSequence](#)).

For each step, this test client will output the mesh set, its topology relation, its nodal coordinate field and the node and element IDs on each instance of the mesh. All mesh instances are collected together into a user-defined collection on

an aggregate set representing the union of the different mesh instances. In addition, the test client will create subsets (blocks) of the mesh for the various *half spaces* in which the mesh exists. For example, for a 2D mesh, it will create, at most, 4 blocks for the (+,+), (-,+), (-,-) and (+,-) quadrants of the 2D plane. Such a subset will only be created if, in fact, a portion of the mesh exists in that particular half-space. We do this primarily to add some interesting subsets to the mesh.

Next, unless ‘-noSimplices’ is specified on the command-line, some of these blocks are refined into simplices. Those blocks that are refined are those whose low order 2 bits of the half-space index yield 2 or 3. In two dimensions, each quad is refined into 2 tringles. In three dimensions, each hex is refined into 6 tets. We do this only to make the element type generated by the code non-uniform.

If no arguments are given, the database will consist of a single file and the four mesh steps depicted in “use case 4-1.gif” will be produced. The node and element IDs will be as defined in the diagram.

Members

Form the sequence of element additions and deletions

GetAddDelSequence is a function defined in birth_death_w.c.

Synopsis:

```
void GetAddDelSequence (const char *inFileName, int *numDims, int *numSteps, int **theOps, Elem-  
Slab_t **theSlabs)
```

Formal Arguments:

- inFileName: [IN] name of input file or NULL if no input file specified
- numDims: [OUT] the number of spatial and topological dimensions of the mesh
- numSteps: [OUT] the number of addition/deletion steps
- theOps: [OUT] array of length numSteps indicating the operation (add=+1,delete=-1)
- theSlabs: [OUT] array of element slabs, one for each step

Description: This function constructs the sequence of element additions or deletions. It either reads input from a file to construct the sequence or, if no file is specified, generates some default data.

The format of the file is described below...

```
1 ndims=<n>  
2 step +|- +|-<K0>,+|-<K1>, ...,+|-<Kn-1>  
3 step +|- +|-<K0>,+|-<K1>, ...,+|-<Kn-1>  
4 step +|- +|-<K0>,+|-<K1>, ...,+|-<Kn-1>  
5 .  
6 .  
7 .
```

For example, the file

```
1 ndims=2  
2 step + +2,+3  
3 step + +2,+0  
4 step - +0,-1  
5 step - -1,+0
```

creates several steps in the life of a 2 dimensional mesh of quads illustrated in “use case 4-1.gif”. .. figure:: use_case_4-1.gif The mesh depicted in “use case 4-1.gif” is, in fact, the default sequence if no input file is specified.

The “ndims=2” line specifies the fact that the mesh will be 2D. In turn, this means that every “step” line in the file will be a 2-tuple of integers. The only valid values for ndims are 1, 2 and 3. Each step line specifies elements to add or delete. A plus (+) sign immediately after “step” indicates an addition while a minus sign (-) indicates a deletion. The signs on the entries of the tuples indicate which *end* of the corresponding axis at which the addition or deletion will occur. The **first** step line is always an addition. For example, in the above steps, the first step line creates a mesh 2 elements wide extending in the positive ‘x’ direction by three elements high extending in the positive ‘y’ direction from the origin. A +0 or -0 entry means the addition or deletion **does not** involve elements along this axis.

See Also:

- *Birth and Death Use Case*: Introduction for current chapter

Open a new database and do some preparatory work on it

OpenDatabase is a function defined in birth_death_w.c.

Synopsis:

```
void OpenDatabase (char *dbname, hbool_t do_multifile, int numDims, DbInfo_t *dbInfo)
```

Formal Arguments:

- dbname: [IN] name of the database
- do_multifile: [IN] boolean to indicate if each step will go to a different supplemental file
- numDims: [IN] number of topological and geometric dimensions in the mesh
- dbInfo: [OUT] database info object

Description: This function creates the initial database and some key objects such as the top-most aggregate set, and the state suite.

The aggregate set **is extendible** because the infinity of points that comprise it can grow (or shrink).

See Also:

- *Birth and Death Use Case*: Introduction for current chapter

Write current mesh to the SAF database

WriteCurrentMesh is a function defined in birth_death_w.c.

Synopsis:

```
void WriteCurrentMesh (DbInfo_t *dbInfo, int theStep, int numDims, CurrentMeshParams_t theMesh,
    SAF_Field *fieldList, int *fieldListSize)
```

Formal Arguments:

- dbInfo: [IN/OUT] database info object
- theStep: current step number
- numDims: [IN] number of dimensions in mesh
- theMesh: [IN] current mesh parameters
- fieldList: [IN/OUT] list of fields we’ll append new fields too
- fieldListSize: [IN/OUT] On input, the current size of the field list. On output, its new size

Description: This function does all the work of writing the current mesh, including its topology relation, subset relations and fields, to the [SAF](#) database.

See Also:

- *Birth and Death Use Case*: Introduction for current chapter

Main program for Birth and Death of Elements Use Case

`main` is a function defined in `birth_death_w.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Formal Arguments:

- `argc`: command line argument count
- `argv`: command line arguments

Description: This is the [main](#) code for the birth and death use case. It gets the steps of additions/deletions of elements either from a file or uses the default steps, initializes the database, loops over all the steps and then closes the database.

If “`do_multifile`” is present on the command-line, each cycle output will be written to different files. Otherwise, it will all be written to one file.

Parallel Notes: Parallel and serial behavior is identical due to use of `SAF__ALL` mode in all calls.

Issues: because we are in a try block here, all failures in any code here or in functions we call from here will send us to the one and only catch block at the end of this test

See Also:

- *Birth and Death Use Case*: Introduction for current chapter

Storagew

This is code that demonstrates indirect fields used to represent fields stored on a domain decomposition. An indirect field is a field whose stored coefficients are handles to fields. Typically such a field is on a superset and the handles are to the “same” field on the subsets. This example generates a [SAF](#) database containing an unstructured mesh with two domains. Indirect fields are specified on the mesh which refer to fields actually stored in fields on the domains.

Parallel and serial behavior are identical due to use of `SAF__ALL` mode in all calls.

Members

Main entry point

`main` is a function defined in `storagew.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Description: See **Storagew**

See Also:

- *Storagew*: Introduction for current chapter

Create mesh

`make_base_space` is a function defined in `storagew.c`.

Synopsis:

void **make_base_space** (void)

Description: Constructs the mesh for `storagew` and writes it to the file.

See Also:

- [Storagew](#): Introduction for current chapter

Create direct coordinate field

`make_direct_coord_field` is a function defined in `storagew.c`.

Synopsis:

void **make_direct_coord_field** (void)

Description: Creates the global coordinate field defined directly on the two domains.

See Also:

- [Storagew](#): Introduction for current chapter

Create direct temperature field

`make_direct_temperature_field` is a function defined in `storagew.c`.

Synopsis:

void **make_direct_temperature_field** (void)

Description: Creates the temperature field defined directly on the two domains.

See Also:

- [Storagew](#): Introduction for current chapter

Create indirect coordinate field

`make_indirect_coord_field` is a function defined in `storagew.c`.

Synopsis:

void **make_indirect_coord_field** (void)

Description: Creates the global coordinate field on the mesh but defined indirectly on the two domains.

See Also:

- [Storagew](#): Introduction for current chapter

Create indirect temperature field

`make_indirect_temperature_field` is a function defined in `storagew.c`.

Synopsis:

void **make_indirect_temperature_field** (void)

Description: Creates the temperature field on the mesh but defined indirectly on the two domains.

See Also:

- [Storagew](#): Introduction for current chapter

Triangle Mesh

No description available.

Members

Main entry point

`main` is a function defined in `triangle_mesh.c`.

Synopsis:

int **main** (int *argc*, char ***argv*)

Description: See *Triangle Mesh*.

See Also:

- [Triangle Mesh](#): Introduction for current chapter

Construct triangle mesh

`make_base_space` is a function defined in `triangle_mesh.c`.

Synopsis:

void **make_base_space** (SAF_Db **db*, SAF_Set **mesh*, SAF_Cat **nodes*, SAF_Cat **elems*, int *edge_ct_x*,
int *edge_ct_y*)

Description: Constructs the triangle mesh and writes it to `SAF__FILE`.

See Also:

- [Triangle Mesh](#): Introduction for current chapter

Construct coordinate field

`make_coord_field` is a function defined in `triangle_mesh.c`.

Synopsis:

void **make_coord_field** (int *edge_ct_x*, int *edge_ct_y*, SAF_Db **db*, SAF_Set **mesh*, SAF_Cat **nodes*,
SAF_Cat **elems*, SAF_Db **saf_file*)

Description: Constructs the coordinate field on the mesh.

See Also:

- *Triangle Mesh*: Introduction for current chapter

Create coordinate field

`make_coord_field_dofs` is a function defined in `triangle_mesh.c`.

Synopsis:

```
double *make_coord_field_dofs (int edge_ct_x, int edge_ct_y)
```

Formal Arguments:

- `edge_ct_x`: number of edges in X direction
- `edge_ct_y`: number of edges in Y direction

Description: Creates the coordinate field for a rectangular triangle mesh. Number of nodes in X direction is `edge_ct_x`+1` and similarly for ``Y direction.

Preconditions:

- There must be at least 1 edge in the x direction. (low-cost)
- There must be at least 1 edge in the y direction. (low-cost)

See Also:

- *Triangle Mesh*: Introduction for current chapter

Create rectangular array of triangles

`make_mesh_connectivity` is a function defined in `triangle_mesh.c`.

Synopsis:

```
int *make_mesh_connectivity (int edge_ct_x, int edge_ct_y)
```

Formal Arguments:

- `edge_ct_x`: number of edges in X direction
- `edge_ct_y`: number of edges in Y direction

Description: Creates a rectangular array of triangles. The number of triangles in the X direction is $2^{edge_ct_x}$ and similarly for the Y direction.

Preconditions:

- There must be at least 1 edge in the x direction. (low-cost)
- There must be at least 1 edge in the y direction. (low-cost)

See Also:

- *Triangle Mesh*: Introduction for current chapter

Construct scalar field

`make_scalar_field` is a function defined in `triangle_mesh.c`.

Synopsis:

```
void make_scalar_field (int edge_ct_x, int edge_ct_y, SAF_Db *db, SAF_Set *mesh, SAF_Cat *nodes,  
                        SAF_Cat *elems, SAF_Db *saf_file)
```

Description: Construct the scalar field on the mesh.

See Also:

- [*Triangle Mesh*](#): Introduction for current chapter

Create scalar on mesh

`make_scalar_field_dofs` is a function defined in `triangle_mesh.c`.

Synopsis:

```
double * make_scalar_field_dofs (int edge_ct_x, int edge_ct_y)
```

Formal Arguments:

- `edge_ct_x`: number of edges in X direction
- `edge_ct_y`: number of edges in Y direction

Description: Creates a scalar on a rectangular triangle mesh. Number of nodes in X direction is `edge_ct_x``+1` and similarly for ``Y direction.

Preconditions:

- There must be at least 1 edge in the x direction. (low-cost)
- There must be at least 1 edge in the y direction. (low-cost)

See Also:

- [*Triangle Mesh*](#): Introduction for current chapter

Construct stress field

`make_stress_field` is a function defined in `triangle_mesh.c`.

Synopsis:

```
void make_stress_field (int edge_ct_x, int edge_ct_y, SAF_Db *db, SAF_Set *mesh, SAF_Cat *elems,  
                        SAF_Db *saf_file)
```

Description: Construct the stress field on the mesh.

See Also:

- [*Triangle Mesh*](#): Introduction for current chapter

Create stress field

`make_stress_field_dofs` is a function defined in `triangle_mesh.c`.

Synopsis:

```
double *make_stress_field_dofs (int edge_ct_x, int edge_ct_y)
```

Formal Arguments:

- `edge_ct_x`: number of edges in X direction
- `edge_ct_y`: number of edges in Y direction

Description: Creates a stress field on a rectangular array of triangles. Used to test instantiation of `st2` field; values of field are meaningless. Number of nodes in X direction is `edge_ct_x`+1` and similarly for ``Y` direction.

Preconditions:

- There must be at least 1 edge in the x direction. (low-cost)
- There must be at least 1 edge in the y direction. (low-cost)

See Also:

- [Triangle Mesh](#): Introduction for current chapter

Dynamic Load Balance Use Case

This is testing code that demonstrates how to use [SAF](#) to output a mesh in which elements are being shifted among processors between each restart *dump*. The bulk of the code here is used simply to create some interesting meshes and has nothing to do with reading/writing from/to [SAF](#). The only routines in which [SAF](#) calls are made are *main*, *OpenDatabase*, *WriteCurrentMesh*, *UpdateDatabase*, *CloseDatabase* and *ReadBackElementHistory*. As such, these are the only *Public* functions defined in this file. If you are interested in seeing the private stuff, and don't see it in what you are reading, you probably need to re-gen the documentation with *mkdoc* and specify an audience of *Private* (as well as *Public*) with the `-a` option. If you are viewing this documentation with an HTML browser, don't forget to following the links to the actual source subroutines described here.

This use case can generate 1, 2 or 3D, unstructured meshes of edge, quad or hex elements, respectively. Use the `-dims %d %d %d` command line option (always pass three args even if you want a 1 or 2D mesh in which case pass zero for the remaining arg(s)). The default mesh is a 2D mesh of 10 x 10 quads.

Upon each *cycle* of this use case, a certain number of elements are shifted from one processor to the next starting with processor 0. This shifting is continued through all the processors until the elements again appear on processor 0.

In the initial decomposition, elements are assigned round-robin to processors. This is, of course, a brain-dead decomposition but is sufficient for purposes of this use case. However, before the round-robin assignment begins, a certain number of elements are held-back from the list of elements to initially assign. This is the number of elements to be shifted between processors. By default, this number is 10. You can specify a different number using the `-numToShift %d` command line argument. These elements are the elements of highest numerical index in the global collection of elements on the whole mesh. Thus, in the default case, elements 90...99 wind up as the shifted elements. Initially, these elements are assigned to processor 0. Then, with each cycle output, they are shifted to the next processor. Consequently, in each *cycle* output by this use case, the element-based processor decomposition is, indeed, a *partition* of the whole. No elements are shared between processors. This decomposition is illustrated for the default case run on 3 processors in "loadbalance diagrams-2.gif".

Since each *cycle* in the output is a different decomposition of the whole, we create different *instances* of the whole mesh in an *self* collection on the whole. The interpretation is that the top-level set and every member of the *self*

collection on that set are equivalent point-sets. They are, indeed, the same set. However, each is decomposed into processor pieces differently.

Two fields are created. One is the coordinate field for the mesh; a piecewise-linear field with 1 dof for each node in the problem. The other is a pressure field with the following time-space behavior. . .

```

1      2
2      t
3      -----
4      2
5      (1 + r)

```

where t is time and r is distance from the origin. This is a piecewise constant field with 1 dof for each element in the problem.

Finally, both the the coordinate field on the whole on the given dump and the pressure field on the whole are written to a state-field. Due to the fact that the state/suite interface does not currently support subsuites (that is subsetting of a suite) we are forced to create a separate state/suite for each dump. This will be corrected in the [SAF](#) library shortly. A high-level diagram of the Subset Relation Graph (SRG) is illustrated in “loadbalance diagrams-2.gif”

Note that this diagram does **not** show the field indirections from a particular decomposed set to its processor pieces. That would have made the diagram even busier than it already is.

Optionally, this use case will output the *dump history* of a given element you specify with the `-histElem %d` command-line option. Within the context of this use case, we define the *dump history* of an element to be the sequence of processor assignments and pressure dofs for each *dump cycle* of the use case. If you specify an element number in the interval $[N\text{-numToShift} \dots N-1]$, where N is the total number of elements in the mesh, the element’s processor assignment should vary, increasing by one each step. Otherwise, it should remain constant. This so because high-numbered elements are shifted while low-numbered ones stay on the processor they were initially assigned to.

An element’s dump history is not output by default. You have to request it by specifying the `-histElem %d` command-line option. Also, to confirm that the use case does, indeed, query back from the database the correct dump history, it also computes and stores the history as the database is generated. At the end of the run, it then prints both the history as it was generated and the history as it was queried from the database for a visual inspection that the results are identical.

Members

Open a new database and do some preparatory work on it

`OpenDatabase` is a function defined in `loadbalance.c`.

Synopsis:

```
void OpenDatabase (char *dbname, hbool_t do_multifile, int numDims, int numProcs, DbInfo_t *dbInfo)
```

Formal Arguments:

- `dbname`: [IN] name of the database
- `do_multifile`: [IN] boolean to indicate if each step will go to a different supplemental file
- `numDims`: [IN] number of topological and geometric dimensions in the mesh
- `numProcs`: [IN] number of processors
- `dbInfo`: [OUT] database info object

Description: This function creates the initial database and some key objects such as the top-most aggregate set, the nodes, elems and procs collection categories and the state suite.

See Also:

- *Dynamic Load Balance Use Case:* Introduction for current chapter

Query an element's history from the database

ReadBackElementHistory is a function defined in loadbalance.c.

Synopsis:

```
void ReadBackElementHistory (DbInfo_t *dbInfo, int myProcNum, int histElem, int *numReadBack, ElementHistory_t **hist)
```

Formal Arguments:

- dbInfo: database info object
- myProcNum: processor rank in MPI_COMM_WORLD
- histElem: the element for which history
- numReadBack: number of dump for which history was read back
- hist: the resulting history buffer

Description: This function queries the pressure dump history for a specific element back out of the database.

Issues: For expediency in completing the use-case, this function was written to be fairly specific to what was written to the database. Note that the approach taken here is written assuming each dump is a different set in the *self* collection on the top-level set.

A more general dump history tool would involve the following...

```
1 Command-line arguments...
2   -elemID %d (0 or more times)    identify the element(s) you want dump history for
3   -nodeID %d (0 or more times)   identify the node(s) you want dump history for
4   -field %s (1 or more times)    identify the field(s) you want dumped for each
↪elem/node
5                                   use "all" for all fields
6   -IDfield                       If the node or element IDs you specified with -
↪elemID or
7                                   -nodeID are not native collection indices,
↪specify the name
8                                   of the field in which these ID's are stored
```

```
1 Possible output...
```


```
1 Dump History for Element 5...
```

stepIdx	step coord	lives in	pressure	velocity	gauss-pts
				vx vy vz	d0 d1 d2 d3
000	0.000	whole-005	2.4	0.1 2.0 2.2	2.0 2.2 2.2 2.3
020	0.059	whole-005	3.9	blah-blah-blah	

```
1 or in some decomposed database...
```

Dump History **for** Element 16...

stepIdx	step coord	lives in	pressure	velocity	gauss-pts
				vx vy vz	d0 d1 d2 d3
000	0.000	proc0-000	2.4	0.1 2.0 2.2	2.0 2.2 2.2 2.3
000	0.000	proc1-007	2.4	0.1 2.0 2.2	2.0 2.2 2.2 2.3
000	0.000	proc2-004	2.4	0.1 2.0 2.2	2.0 2.2 2.2 2.3
020	0.059	proc0-000	42.3	0.1 2.0 2.2	2.0 2.2 2.2 2.3
020	0.059	proc1-007	42.4	0.1 2.0 2.2	2.0 2.2 2.2 2.3
020	0.059	proc2-004	42.3	0.1 2.0 2.2	2.0 2.2 2.2 2.3

In this example, element 16 is shared by three processor pieces. And, this shows  example output where one processor doesn't agree with the others on the pressure value.

```

algorithm...
1. open the database
2. find all suites
3. order suites (by coord value associated with first member state or something)
4. For each suite...
  a. read a state field
  b. for each field in the state matching field(s) specified on command-line.
      a. if field is inhomog, find pieces on which it is homog
          1. locate the identified nodes and elements in each piece
             by reading subset relations and examining them
          2. for each piece...
              partially read the field to obtain the dofs for all
              specified node/elems
          3. capture dof values, names of pieces (sets) and local
             indexes on these sets
      b. otherwise, just partially read the field to obtain the dofs
         for all specified nodes/elems
      c. build up buffers of dof values, names of pieces (sets) and local
         indexes on these sets

```

See Also:

- *Dynamic Load Balance Use Case*: Introduction for current chapter

Write current mesh to the SAF database

WriteCurrentMesh is a function defined in loadbalance.c.

Synopsis:

```
void WriteCurrentMesh (DbInfo_t *dbInfo, int theStep, int numDims, int numProcs, int myProcNum, CurrentMeshParams_t *theMesh, SAF_Field *fieldList, int *fieldListSize)
```

Formal Arguments:

- dbInfo: [IN/OUT] database info object
- theStep: [IN] current step number
- numDims: [IN] number of dimensions in mesh
- numProcs: [IN] number of processors

- myProcNum: [IN] the rank of calling processor
- theMesh: [IN/OUT] current mesh parameters (relation and file handle updated)
- fieldList: [IN/OUT] list of fields we'll append newly written fields too
- fieldListSize: [IN/OUT] On input, the current size of the field list. On output, its new size

Description: This function does all the work of writing the current mesh, including its domain-decomposed topology relation, processor subset relations, coordinate field, and pressure field to the [SAF](#) database.

See Also:

- *Dynamic Load Balance Use Case*: Introduction for current chapter

Main program for Dynamic Load Balance Use Case

main is a function defined in loadbalance.c.

Synopsis:

```
int main (int argc, char **argv)
```

Formal Arguments:

- argc: command line argument count
- argv: command line arguments

Description: This is the [main](#) code for the dynamic load balance use case.

Here are the command-line options...

```

1  -multifile
2      each cycle output will be written to different files. Otherwise, it will all be
↪written to one file.
3  -numToShift %d
4      specify the number of elements to shift on each step [10].
5  -meshSize %d %d %d
6      specify size of mesh in 1, 2 or 3 dimensions. Specify 0 for each dimension you do
↪not want to have.
7      For example, -meshSize 5 0 0 specifies a 1D mesh of size 5 elements [10 10 0]
8  -histElem %d
9      specify an element, using a global element id, whose pressure history is to
↪displayed at the
10     end of the run. In this case, the database is closed and then re-opened. For each
↪instant of the mesh in
11     the database, the specified element's pressure is found by first finding which
↪processor set the
12     element was assigned to. This find step is done in parallel. Once the processor-
↪set is known, that
13     processor re-opens the database and reads the field, using partial I/O on that
↪specific set for the
14     specific element and prints a value. If you want to specify an element that you
↪know has been
15     shifted, use an element id within <numToShift> elements of the highest element
↪number.
```

Issues: Only two of the proc-to-top subset relations are different in each step. It would be nice to re-use the data already written when the relations are identical to some other previous step. A function such as `saf_usewritten_rel``(``SAF__Rel theRel, SAF__Rel alreadyWrittenRel);` would do the job.

It might be nice to provide a `-histNode` command-line option. Node history is a little different because some nodes are shared between processors.

This is intended to be only a parallel client. In serial, this example should be skipped.

This is really only a parallel test. It doesn't make much sense to run it in serial

See Also:

- *Dynamic Load Balance Use Case*: Introduction for current chapter

Example Utilities

No description available.

Members

Close the database

`CloseDatabase` is a function defined in `exampleutil.c`.

Synopsis:

```
void CloseDatabase (DbInfo_t dbInfo)
```

Formal Arguments:

- `dbInfo`: database info object

Description: This function performs any tasks associated with closing the database. Currently, the only thing this does is call `:file:`saf_close_database ../safapi_refman.rest/saf_close_database.rst``.

See Also:

- *Example Utilities*: Introduction for current chapter

Update the database to the current step

`UpdateDatabase` is a function defined in `exampleutil.c`.

Synopsis:

```
void UpdateDatabase (DbInfo_t *dbInfo, int stepNum, int numSteps, hbool_t do_multifile, int numFields,  
                    SAF_Field *theFields)
```

Formal Arguments:

- `dbInfo`: [IN/OUT] database info object (`currentFile` member can be modified)
- `stepNum`: [IN] the current step number in the sequence starting from zero
- `numSteps`: [IN] total number of steps to be output (≥ 1)
- `do_multifile`: [IN] boolean to indicate if each step will go to a different supplemental file
- `numFields`: [IN] the number of fields on the mesh
- `theFields`: [IN] array of length `numFields` of the field handles

Description: This function performs a number of tasks involved in updating the database to the current step in the sequence of additions and deletions.

Issues: We are forced to create a new state field each time this function is called. This can be viewed as either appropriate or counter-intuitive depending on how the SAF client *views* its data. Certainly, since each step in the sequence of additions and deletions changes the mesh, the state output by the client is different on each step. This follows the strict definition of state currently supported by SAF's states and suites API. However, if the list of fields that are output **do not change** and only their base-space varies with time, is the state output by the client, in the eyes of the client, really different? We could support this somewhat looser definition of state by instead of associating with each state the fields on the mesh instances, we used fields on the aggregate. However, this would require a minor alteration to our current interpretation of inhomogeneous fields. If interested, we can pursue further.

Because the fields from each state are defined on a different base-space, we need to define a new suite for every step in the simulation. This will be corrected when SAF supports sub-setting on suites. Then, we'll be able to define an inhomogeneous field on the suite for its different pieces.

we always write to the 0'th index of this new suite

See Also:

- *Example Utilities:* Introduction for current chapter

Hadaptive Use Case

This is a simple example of using SAF to represent adaptation in a parallel-decomposed mesh. The use-case is hard-coded to write the 6 mesh states illustrated in "use case 5-1.gif"

This example is designed to run on only 3 processors. It will abort with error if run in parallel on any other number of processors.

There are a total of 6 states output. The ending state is identical to the initial state. Other than a coordinate field, there are no other whole-mesh fields declared. This is simply due to expediency in completing this example code.

State 0

State 1

State 2

State 3

State 4

State 5

To give elements an immutable, global ID, we use something called the *LRC* index which is a triple of the level of adaptivity and then the row and column index within that level starting from the origin in the upper right. The largest elements are defined, arbitrarily, to be at level 1. So, for example, the lower-right child of the one element that is refined in state 1 has an LRC index of {2,1,3} for level 2, row 1 and column 3. The LRC indexes are written on each step as alternative indexing for the elements. In the dialog that follows, any reference to an element enclosed in '{' and '}' braces is the element's LRC index.

In each state, each processor enumerates changes in the mesh *relative* to the last known dump of the mesh to the database. Two broad categories of changes are tracked; refinements and re-balances. Refinements are tracked and enumerated in the *global* context. Re-balancing is tracked and enumerated in the *local* context. In other words, refinement information is stored *above* the processor decomposition while re-balancing information is stored *below*

it in the Subset Relation Graph (SRG). This was completely arbitrary and can be changed if desired. Given all the elements on a processor at a given state, each processor stores information to answer: “Which of my elements in the current state...”

- 1

a. ...did I get from another processor in the last known state.

2

b. ...are children of (refinements of) an element in the last known state.

In addition, we arbitrarily choose to store information on UNrefinements and no-changes as well as elements that are kept (as opposed to re-balanced). There is no particular reason to do this other than trying to make the example a little more interesting. It costs more data that is non-essential.

In some cases, between two state dumps an element may be refined **and** some or all of its children may be re-balanced to other processors. When this happens, from the point of view of the database receiving information about each state, the convention used is that the processor in the previous state that owned the parent made the decision to refine it **and** then re-balanced the elements to other processors. For example, between state 0 and state 1 in “use case 5-1.gif”, element {1,0,1} which lived on processor 0 in state 0 was refined **and** half (2) of its children were given to processor 1. Thus, in state 1, processor 1 treats these two elements as being **both** refinements **and** re-balances.

Next, for re-balances, a field is stored on the re-balances set which identifies the processor from which each element in the set came.

[Side note: Why not store this information as a set of subsets? That too, is completely appropriate. The approach chosen here is merely more convenient and storage efficient. The fact is, there is a duality in how certain kinds of information can be captured in SAF. This duality is a fundamental aspect of the mathematical interpretation of a field defined on member(s) of a Set Relation Graph (SRG). In short, if one wishes to enumerate a value for each element in a collection, one has a choice of saying (in natural language), “for each element, which value does it have...” or “for each value, which elements have that value...”. The former approach takes the form of a field while the latter approach takes the form of a set of subsets (a partition in fact). In fact, there is document that discusses these issues in detail available from the SAF web-pages at www.ca.sandia.gov/ASCI/sdm/SAF/docs/wips/fields_n_maps.ps.gz In summary, while one may have a natural way of *thinking* about this kind of data, there is clear mathematical theory to explain why either approach is appropriate **and** there are even theoretical storage and performance reasons to prefer one over the other depending on the situation. – end side note]

Since developing this initial use-case, a couple of enhancements have been identified that would be make the use-case more realistic and facilitate certain kinds of queries. First, we’ve identified a way to use SAF to capture differences in the sets from one state to the next as opposed or in addition to each specific state. Second, we’ve identified a way to make forward references (as apposed or in addition to to backward) to facilitate forward tracking of changes in refinement and rebalancing.

Issues: The ability to talk about the “difference” between two SRGs would be useful. If one is permitted only to enumerate a given state of the client’s data, it is difficult to store information at state I that captures what is changed in going to state I+1. For example, in going from state 0 to state 1, element {1,0,1} is refined into 4 children. However, the output for state 0 can’t mention any of these children because when state 0 is created, they don’t exist. Because they do exist in state 1, we can talk about where they came from relative to state 0. Thus, a causality is imposed, which the current implementation demonstrates, in the direction in which we can talk about changes (as mentioned above, I think we have identified as solution to this).

If one wishes to capture the differences between states, where does that information “live”? The differences represent what happened in making the transition from one state to the next. In some sense the differences represent actions on objects and not objects themselves. For example, “...these elements were added by refinement of that element...” or “...these elements were obtained by rebalancing from that processor...” are the kinds of statements one might like to make. It would be nice of such differences could be captured using the existing objects available in SAF rather than having to create new ones. I think I have identified a way of doing this. Given two states, ‘a’ and ‘b’, and two sets, S

and P where S is the subset of P in both states, we can talk about the difference of S_a and S_b (that is S in state a and S in state b) in P by introducing two subsets of S , one in state a , called D_{ab} and one in state b called D_{ba} where

```
1 Dab = Sa - Sb (all points in Sa but not in Sb)
2 Dba = Sb - Sa (all points in Sb but not in Sa)
```

Together, these two sets represent, in effect, additions and deletions of points in going from S_a to S_b or vice versa. D_{ab} is the set of points deleted from S_a in arriving at S_b and D_{ba} is the set of points added to S_a in arriving at S_b (or deleted from S_b to arrive at S_a in the reverse direction).

Both D_{ab} and D_{ba} are ordinary subsets in their respective SRGs. However, what we are missing from **SAF** is the ability to *declare* that D_{ab} is a difference subset and which set it is differenced with. There are two possible routes to take here. One is to simply add a `SAF__DIFFERENCE` option to the `:file:`saf_declare_subset_relation`../safapi_refman.rest/saf_declare_subset_relation.rst`` call so that some subsets can be defined that are differences with other sets. The other route is to add a new function to declare expressions involving sets such as... `saf_declare_set_expression``(``SAF__Set resultSet, char *expr)` along with functions to build up the string representation for the expression. This would then permit a client to find sets in the SRG according to a given expression (implementation details would require something like an `expr_blob_id` member of a set object in VBT which could be implemented as a `meta_blob`). The latter approach is more general in that it permits a variety of set expressions to be characterized, not just a difference.

Because **SAF** is targeted primarily as a data modeling and I/O library, it is typically used to output restart or plot dumps for states that are *far* apart relative to the physics time-step. For example, there may be many hundreds of time-steps from one state dump to the next. Consequently, the relationships that can be captured in such a scenario are how the two states **as told to the**I/O**system** are related. For example, if a state is dumped at time I where an element, say K , is on processor 0 and then this element migrates from processor 0, to 1, to 5, to 17 and finally to 22 before a new state is dumped to the I/O system, the only fact that the I/O system can capture is that, somehow, element K on processor 0 was given to processor 22. In order to capture the in-between information, each of those states must be enumerated to the I/O system. This might be where having the ability to enumerate state-transitions as opposed to just states would be useful. Then, it may be relatively simple to enumerate each of the states the code went through.

Members

Open a new database and do some preparatory work on it

`OpenDatabase` is a function defined in `hadaptive.c`.

Synopsis:

```
void OpenDatabase (char *dbname, hbool_t do_multifile, DbInfo_t *dbInfo)
```

Formal Arguments:

- `dbname`: [IN] name of the database
- `do_multifile`: [IN] boolean to indicate if each step will go to a different supplemental file
- `dbInfo`: [OUT] database info object

Description: This function creates the initial database and some key objects such as the top-most aggregate set, the nodes, elems and procs collection categories and the state suite.

See Also:

- [Adaptive Use Case](#): Introduction for current chapter

Write current mesh to the SAF database

`WriteCurrentMesh` is a function defined in `hadaptive.c`.

Synopsis:

```
void WriteCurrentMesh (DbInfo_t *dbInfo, int theStep, int numProcs, int myProcNum, CurrentMesh-  
Params_t theMesh, SAF_Field *fieldList, int *fieldListSize)
```

Formal Arguments:

- `dbInfo`: [IN/OUT] database info object
- `theStep`: [IN] current step number
- `numProcs`: [IN] number of processors
- `myProcNum`: [IN] the rank of calling processor
- `theMesh`: [IN] current mesh parameters
- `fieldList`: [IN/OUT] list of fields we'll append newly written fields too
- `fieldListSize`: [IN/OUT] On input, the current size of the field list. On output, its new size

Description: This function does all the work of writing the current mesh, including its domain-decomposed topology relation, processor subset relations, coordinate field, and pressure field to the [SAF](#) database.

See Also:

- *Hadaptive Use Case*: Introduction for current chapter

Main program for Hadaptive Use Case

`main` is a function defined in `hadaptive.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Formal Arguments:

- `argc`: command line argument count
- `argv`: command line arguments

Issues: This is really only a parallel test. It doesn't make much sense to run it in serial

See Also:

- *Hadaptive Use Case*: Introduction for current chapter

Larry Use Case

This is testing code that exercises Larry Schoof first use case. This code declares [SAF](#) objects and writes the raw data.

Members

Main entry point

`main` is a function defined in `larrylw.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Description: Implementation of Larry use case.

Return Value: Exit status is the number of failures.

See Also:

- *Larry Use Case*: Introduction for current chapter

Construct the mesh

`make_base_space` is a function defined in `larry1w.c`.

Synopsis:

```
void make_base_space (void)
```

Description: Constructs the mesh for Larry use case and writes it to a file.

See Also:

- *Larry Use Case*: Introduction for current chapter

Construct the displacement field

`make_displacement_field` is a function defined in `larry1w.c`.

Synopsis:

```
void make_displacement_field (void)
```

Description: Constructs the displacement field on the mesh. The field templates for the displacement field are the same as the templates for the global coordinate field.

See Also:

- *Larry Use Case*: Introduction for current chapter

Construct the distribution factors

`make_distribution_factors_on_ss2_field` is a function defined in `larry1w.c`.

Synopsis:

```
void make_distribution_factors_on_ss2_field (void)
```

Description: Constructs the distribution factors field on side set 2.

See Also:

- *Larry Use Case*: Introduction for current chapter

Construct the coordinate field

`make_global_coord_field` is a function defined in `larry1w.c`.

Synopsis:

void **`make_global_coord_field`** (void)

Description: Constructs the coordinate field on the mesh.

See Also:

- *Larry Use Case:* Introduction for current chapter

Create initial suite

`make_init_suite` is a function defined in `larry1w.c`.

Synopsis:

void **`make_init_suite`** (void)

Description: Create a suite for initial state (time step zero).

See Also:

- *Larry Use Case:* Introduction for current chapter

Construct the stress field

`make_pressure_on_ss1_field` is a function defined in `larry1w.c`.

Synopsis:

void **`make_pressure_on_ss1_field`** (void)

Description: Constructs the stress field on cell 1.

See Also:

- *Larry Use Case:* Introduction for current chapter

Construct the stress field

`make_stress_on_cell_1_field` is a function defined in `larry1w.c`.

Synopsis:

void **`make_stress_on_cell_1_field`** (void)

Description: Construct the stress field on cell 1.

See Also:

- *Larry Use Case:* Introduction for current chapter

Construct the temperature field

`make_temperature_on_cell_2_field` is a function defined in `larry1w.c`.

Synopsis:

void **`make_temperature_on_cell_2_field`** (void)

Description: Constructs the temperature field on node set 1.

See Also:

- *Larry Use Case:* Introduction for current chapter

Construct the temperature field

`make_temperature_on_ns1_field` is a function defined in `larry1w.c`.

Synopsis:

void **`make_temperature_on_ns1_field`** (void)

Description: Constructs the temperature field on node set 1.

See Also:

- *Larry Use Case:* Introduction for current chapter

Construct the time field

`make_time_base_field` is a function defined in `larry1w.c`.

Synopsis:

void **`make_time_base_field`** (void)

Description: Construct the time field on the time base.

See Also:

- *Larry Use Case:* Introduction for current chapter

Create time suite

`make_time_suite` is a function defined in `larry1w.c`.

Synopsis:

void **`make_time_suite`** (void)

Description: Create time suite.

See Also:

- *Larry Use Case:* Introduction for current chapter

N to 1 Remapping Use Case

No description available.

Members

Main entry point

`main` is a function defined in `remap_n21.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Description: See N to 1 Remapping Use Case.

See Also:

- *N to 1 Remapping Use Case*: Introduction for current chapter

Overloaded Definitions

These objects have multiple definitions.

Members

OpenDatabase

This object has overloaded definitions.

Members

WriteCurrentMesh

This object has overloaded definitions.

Members

`main`

This object has overloaded definitions.

Members

This program demonstrates the functions needed by a parallel EXODUS application.

`main` is a function defined in `exo_par_wt.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Description: Creates a [SAF](#) database in parallel, emulating a parallel EXODUS writing client.

The code may be somewhat confusing but the subset relation graph (SRG) is straightforward. The SRG looks like this:

```
TOP_SET || |—DOMAIN_0 || || |—BLOCK_0_DOMAIN_0 ||. ||. | |—BLOCK_num_blocks_DOMAIN_0
| |—DOMAIN_1 |. |. |. |—DOMAIN_num_domains || || |—BLOCK_0_DOMAIN_num_domains ||. |
|. | |—BLOCK_num_blocks_DOMAIN_num_domains || || |—BLOCK_0 || || |—BLOCK_0_DOMAIN_0
(same set as subset of DOMAIN_0) ||. ||. | |—BLOCK_0_DOMAIN_num_domains | |—BLOCK_1 |. |. |.
|—BLOCK_num_blocks
```

```
|—BLOCK_num_blocks_DOMAIN_0 |. |. |—BLOCK_num_blocks_DOMAIN_num_domains
```

Parallel Notes: This is a parallel client.

See Also:

- [Tests](#): Introduction for current chapter

Basic EXODUS test

`main` is a function defined in `exo_basic_wt.c`.

Synopsis:

```
int main (int argc, char **argv)
```

Description: The `exo_basic_wt` and `exo_basic_rd` test the [SAF](#) library to ensure that all serial EXODUS functionality is supported.

Parallel Notes: This tests [SAF](#) in serial. There are corresponding tests (`exo_par_wt` and `exo_par_rd`) to test [SAF](#) in parallel.

See Also:

- [Tests](#): Introduction for current chapter

make_base_space

This object has overloaded definitions.

Members

Tests

No description available.

Members

Permuted Index

Table 1.2: Permuted Index

Concept	Key	Reference
Form the sequence of element	additions and deletions	<i>GetAddDelSequence</i>
This program demonstrates the functions needed by a parallel EXODUS	application.	<i>main</i>
Create rectangular	array of triangles	<i>make_mesh_connectivity</i>
Main program for Dynamic Load	Balance Use Case	<i>main</i>
	Basic EXODUS test	<i>main</i>
Main program for	Birth and Death of Elements Use Case	<i>main</i>
Main program for Birth and Death of Elements Use	Case	<i>main</i>
Main program for Dynamic Load Balance Use	Case	<i>main</i>
Main program for Hadaptive Use	Case	<i>main</i>
	Close the database	<i>CloseDatabase</i>
	Construct coordinate field	<i>make_coord_field</i>
	Construct scalar field	<i>make_scalar_field</i>
	Construct stress field	<i>make_stress_field</i>
	Construct the coordinate field	<i>make_global_coord_field</i>
	Construct the displacement field	<i>make_displacement_field</i>
	Construct the distribution factors	<i>make_distribution_factors_on</i>
	Construct the mesh	<i>make_base_space</i>
	Construct the stress field	<i>make_stress_on_cell_1_field</i>
	Construct the stress field	<i>make_pressure_on_ss1_field</i>
	Construct the temparature field	<i>make_temperature_on_ns1_fi</i>
	Construct the temperature field	<i>make_temperature_on_cell_2</i>
	Construct the time field	<i>make_time_base_field</i>
	Construct triangle mesh	<i>make_base_space</i>
Construct	coordinate field	<i>make_coord_field</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
Construct the	coordinate field	<i>make_global_coord_field</i>
Create	coordinate field	<i>make_coord_field_dofs</i>
Create direct	coordinate field	<i>make_direct_coord_field</i>
Create indirect	coordinate field	<i>make_indirect_coord_field</i>
	Create coordinate field	<i>make_coord_field_dofs</i>
	Create direct coordinate field	<i>make_direct_coord_field</i>
	Create direct temperature field	<i>make_direct_temperature_field</i>
	Create indirect coordinate field	<i>make_indirect_coord_field</i>
	Create indirect temperature field	<i>make_indirect_temperature_field</i>
	Create initial suite	<i>make_init_suite</i>
	Create mesh	<i>make_base_space</i>
	Create rectangular array of triangles	<i>make_mesh_connectivity</i>
	Create scalar on mesh	<i>make_scalar_field_dofs</i>
	Create stress field	<i>make_stress_field_dofs</i>
	Create time suite	<i>make_time_suite</i>
Write	current mesh to the SAF database	<i>WriteCurrentMesh</i>
Write	current mesh to the SAF database	<i>WriteCurrentMesh</i>
Write	current mesh to the SAF database	<i>WriteCurrentMesh</i>
Update the database to the	current step	<i>UpdateDatabase</i>
Close the	database	<i>CloseDatabase</i>
Query an element's history from the	database	<i>ReadBackElementHistory</i>
Write current mesh to the SAF	database	<i>WriteCurrentMesh</i>
Write current mesh to the SAF	database	<i>WriteCurrentMesh</i>
Write current mesh to the SAF	database	<i>WriteCurrentMesh</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
Open a new	database and do some preparatory work on it	<i>OpenDatabase</i>
Open a new	database and do some preparatory work on it	<i>OpenDatabase</i>
Open a new	database and do some preparatory work on it	<i>OpenDatabase</i>
Update the	database to the current step	<i>UpdateDatabase</i>
Main program for Birth and	Death of Elements Use Case	<i>main</i>
Form the sequence of element additions and	deletions	<i>GetAddDelSequence</i>
This program	demonstrates the functions needed by a parallel EXODUS application.	<i>main</i>
Create	direct coordinate field	<i>make_direct_coord_field</i>
Create	direct temperature field	<i>make_direct_temperature_field</i>
Construct the	displacement field	<i>make_displacement_field</i>
Construct the	distribution factors	<i>make_distribution_factors_on</i>
Open a new database and	do some preparatory work on it	<i>OpenDatabase</i>
Open a new database and	do some preparatory work on it	<i>OpenDatabase</i>
Open a new database and	do some preparatory work on it	<i>OpenDatabase</i>
Main program for	Dynamic Load Balance Use Case	<i>main</i>
Form the sequence of	element additions and deletions	<i>GetAddDelSequence</i>
Query an	element's history from the database	<i>ReadBackElementHistory</i>
Main program for Birth and Death of	Elements Use Case	<i>main</i>
Main	entry point	<i>main</i>
Main	entry point	<i>main</i>
Main	entry point	<i>main</i>
Main	entry point	<i>main</i>
This program demonstrates the functions needed by a parallel	EXODUS application.	<i>main</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
Basic	EXODUS test	<i>main</i>
Construct the distribution	factors	<i>make_distribution_factors_on</i>
Construct coordinate	field	<i>make_coord_field</i>
Construct scalar	field	<i>make_scalar_field</i>
Construct stress	field	<i>make_stress_field</i>
Construct the coordinate	field	<i>make_global_coord_field</i>
Construct the displacement	field	<i>make_displacement_field</i>
Construct the stress	field	<i>make_stress_on_cell_1_field</i>
Construct the stress	field	<i>make_pressure_on_ss1_field</i>
Construct the temperature	field	<i>make_temperature_on_ns1_fi</i>
Construct the temperature	field	<i>make_temperature_on_cell_2</i>
Construct the time	field	<i>make_time_base_field</i>
Create coordinate	field	<i>make_coord_field_dofs</i>
Create direct coordinate	field	<i>make_direct_coord_field</i>
Create direct temperature	field	<i>make_direct_temperature_fiel</i>
Create indirect coordinate	field	<i>make_indirect_coord_field</i>
Create indirect temperature	field	<i>make_indirect_temperature_f</i>
Create stress	field	<i>make_stress_field_dofs</i>
	Form the sequence of element additions and deletions	<i>GetAddDelSequence</i>
This program demonstrates the	functions needed by a parallel EXODUS application.	<i>main</i>
Main program for	Hadaptive Use Case	<i>main</i>
Query an element's	history from the database	<i>ReadBackElementHistory</i>
Create	indirect coordinate field	<i>make_indirect_coord_field</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
Create	indirect temperature field	<i>make_indirect_temperature_f</i>
Create	initial suite	<i>make_init_suite</i>
Open a new database and do some preparatory work on	it	<i>OpenDatabase</i>
Open a new database and do some preparatory work on	it	<i>OpenDatabase</i>
Open a new database and do some preparatory work on	it	<i>OpenDatabase</i>
Main program for Dynamic	Load Balance Use Case	<i>main</i>
	Main entry point	<i>main</i>
	Main entry point	<i>main</i>
	Main entry point	<i>main</i>
	Main entry point	<i>main</i>
	Main entry point	<i>main</i>
	Main program for Birth and Death of Elements Use Case	<i>main</i>
	Main program for Dynamic Load Balance Use Case	<i>main</i>
	Main program for Hadaptive Use Case	<i>main</i>
Construct the	mesh	<i>make_base_space</i>
Construct triangle	mesh	<i>make_base_space</i>
Create	mesh	<i>make_base_space</i>
Create scalar on	mesh	<i>make_scalar_field_dofs</i>
Write current	mesh to the SAF database	<i>WriteCurrentMesh</i>
Write current	mesh to the SAF database	<i>WriteCurrentMesh</i>
Write current	mesh to the SAF database	<i>WriteCurrentMesh</i>
This program demonstrates the functions	needed by a parallel EXODUS application.	<i>main</i>
Open a	new database and do some preparatory work on it	<i>OpenDatabase</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
Open a	new database and do some preparatory work on it	<i>OpenDatabase</i>
Open a	new database and do some preparatory work on it	<i>OpenDatabase</i>
Open a new database and do some preparatory work	on it	<i>OpenDatabase</i>
Open a new database and do some preparatory work	on it	<i>OpenDatabase</i>
Open a new database and do some preparatory work	on it	<i>OpenDatabase</i>
Create scalar	on mesh	<i>make_scalar_field_dofs</i>
	Open a new database and do some preparatory work on it	<i>OpenDatabase</i>
	Open a new database and do some preparatory work on it	<i>OpenDatabase</i>
	Open a new database and do some preparatory work on it	<i>OpenDatabase</i>
This program demonstrates the functions needed by a	parallel EXODUS application.	<i>main</i>
Main entry	point	<i>main</i>
Main entry	point	<i>main</i>
Main entry	point	<i>main</i>
Main entry	point	<i>main</i>
Open a new database and do some	preparatory work on it	<i>OpenDatabase</i>
Open a new database and do some	preparatory work on it	<i>OpenDatabase</i>
Open a new database and do some	preparatory work on it	<i>OpenDatabase</i>
This	program demonstrates the functions needed by a parallel EXODUS application.	<i>main</i>
Main	program for Birth and Death of Elements Use Case	<i>main</i>
Main	program for Dynamic Load Balance Use Case	<i>main</i>
Main	program for Hadaptive Use Case	<i>main</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
	Query an element's history from the database	<i>ReadBackElementHistory</i>
Create	rectangular array of triangles	<i>make_mesh_connectivity</i>
Write current mesh to the	SAF database	<i>WriteCurrentMesh</i>
Write current mesh to the	SAF database	<i>WriteCurrentMesh</i>
Write current mesh to the	SAF database	<i>WriteCurrentMesh</i>
Construct	scalar field	<i>make_scalar_field</i>
Create	scalar on mesh	<i>make_scalar_field_dofs</i>
Form the	sequence of element additions and deletions	<i>GetAddDelSequence</i>
Open a new database and do	some preparatory work on it	<i>OpenDatabase</i>
Open a new database and do	some preparatory work on it	<i>OpenDatabase</i>
Open a new database and do	some preparatory work on it	<i>OpenDatabase</i>
Update the database to the current	step	<i>UpdateDatabase</i>
Construct	stress field	<i>make_stress_field</i>
Construct the	stress field	<i>make_stress_on_cell_1_field</i>
Construct the	stress field	<i>make_pressure_on_ss1_field</i>
Create	stress field	<i>make_stress_field_dofs</i>
Create initial	suite	<i>make_init_suite</i>
Create time	suite	<i>make_time_suite</i>
Construct the	temparature field	<i>make_temperature_on_ns1_fi</i>
Construct the	temperature field	<i>make_temperature_on_cell_2</i>
Create direct	temperature field	<i>make_direct_temperature_fiel</i>
Create indirect	temperature field	<i>make_indirect_temperature_f</i>
Basic EXODUS	test	<i>main</i>

Continued on next page

Table 1.2 – continued from previous page

Concept	Key	Reference
	This program demonstrates the functions needed by a parallel EXODUS application.	<i>main</i>
Construct the	time field	<i>make_time_base_field</i>
Create	time suite	<i>make_time_suite</i>
Construct	triangle mesh	<i>make_base_space</i>
Create rectangular array of	triangles	<i>make_mesh_connectivity</i>
	Update the database to the current step	<i>UpdateDatabase</i>
Main program for Birth and Death of Elements	Use Case	<i>main</i>
Main program for Dynamic Load Balance	Use Case	<i>main</i>
Main program for Hadaptive	Use Case	<i>main</i>
Open a new database and do some prepratory	work on it	<i>OpenDatabase</i>
Open a new database and do some prepratory	work on it	<i>OpenDatabase</i>
Open a new database and do some prepratory	work on it	<i>OpenDatabase</i>
	Write current mesh to the SAF database	<i>WriteCurrentMesh</i>
	Write current mesh to the SAF database	<i>WriteCurrentMesh</i>
	Write current mesh to the SAF database	<i>WriteCurrentMesh</i>

1.3 SAF Support Library (SSlib) Programming Reference Manual for SAF 2.0.0 and later

Acknowledgements

1.3.1 Table of Contents

Introduction

The SAF Support Library (SSlib) grew out of experience the Sets and Fields (SAF) team had with the former Vector Bundle Tables (VBT) layer and Data Sharability Layer (DSL) and to some extent with the Hierarchical Data Format version 5, HDF5 (support.hdfgroup.org/HDF5/doc/index.html) library from NCSA. It was decided that in order to

increase performance, generalize some underlying functionality, and improve code engineering that we would embark on an effort to rewrite most of VBT and DSL with these goals in mind:

Reduced Communication: We learned by experience that designing an API that requires underlying communication makes it extremely difficult to optimize for performance at a later time, and that algorithms that require communication can be substantially slower than those that don't. So algorithms will be used to reduce communication and the API will be designed so that cases of repeated communication in the old VBT/DSL API can be performed just once, and cases of related communication can be combined into single messages.

Variable Length Datatypes: The VBT design set aside a fixed size character array for every string, which resulted in substantial wasted file space and lower bandwidth and precluded the client from using arbitrary length strings. The SSlib will employ HDF5 variable length datatypes to avoid these problems.

Transient Objects: The original VBT specification had no provision for creating objects that exist only in memory, although eventually this was patched in using HDF5's `core` virtual file driver. Transient objects are designed into SSlib.

Object Deletion: VBT did not allow for easy deletion of objects from the database. Although SSlib probably won't allow individual objects to be deleted, it will allow entire scopes to be deleted, freeing up memory in the HDF5 file as provided by the HDF5 library and file format.

Every File a Database: SAF had a notion of supplemental data files that were pointed to by a single master file, collectively called the database. It was not possible to open just a supplemental file, but one always had to open the master file. SSlib will make no distinction between master and supplemental files, rather every file will be a self-contained database. SAF allowed supplemental files to be missing; SSlib allows databases to be missing.

Partial Metadata Reads: VBT always read all the object definitions from the database whenever a database was opened. SSlib will only read subsets of a file called "scopes" and only when those scopes are accessed and only by the tasks accessing those scopes.

Interfile Object References: A VBT file could only refer to objects that were also in the same file. SSlib files will have the capability to refer to objects that are in some other file.

Multiple References: In SSlib, two or more objects may make references to a common third object or to common raw data, thus reducing the required storage.

Object Copying: Tools such as `safdiff` formerly needed extensive coding in order to copy an object (e.g., a field) from one database to another. SSlib will provide that functionality at a much lower layer. This also simplifies the implementation of Object Registries in SAF by moving much of that functionality downward in the software stack.

Common Error Handling: A code engineering aspect of SSlib is to generalize the HDF5 error handling subsystem, turn it into a public programming interface, and use it for SSlib and eventually higher software layers. This unifies the error recording and reporting features of all layers involved.

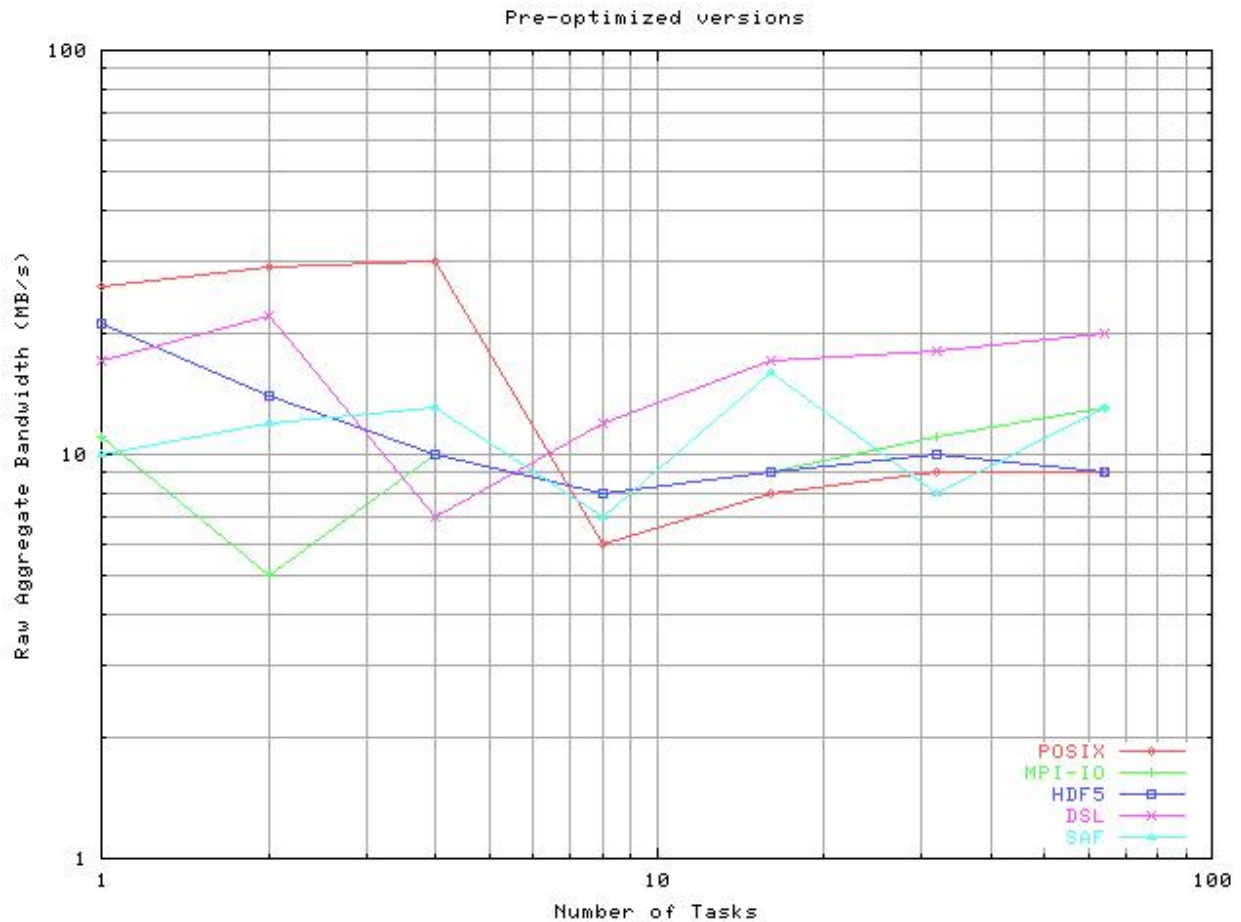
Flexible File Decomposition: As mentioned already, SAF required all object metadata to be stored in a single master file with optional supplemental files to hold raw field data. SSlib relaxes that constraint so that operational environments like SILO's multi-file output are possible, where the MPI job is partitioned into smaller subsets of tasks with each subset responsible for a single database, the databases being "sewed" together later.

Reduced Code Generation: SSlib replaces the more than 12,000 lines of `vbtgen` (a table parser and C code generator) with a few hundred lines of perl that does something very similar. In addition, the perl script parses standard C typedefs instead of a custom language.

BetterHDF5**Coupling:** The DSL datatype interface (more than 12,000 lines of library code) will be replaced with the HDF5 datatype interface plus a few additional functions that may migrate into the HDF5 library.

The plots below show the before and after scalability and performance improvements achieved.

Pre-optimized raw data I/O aggregate bandwidth scalability



Pre-optimized overall I/O aggregate bandwidth scalability

Optimized raw data I/O aggregate bandwidth scalability

Optimized overall I/O aggregate bandwidth scalability

Comparison of [SAF](#) and Silo A1e3d restart file dump timesComparison of [SAF](#) A1e3d restart file dump times by functionality

Library

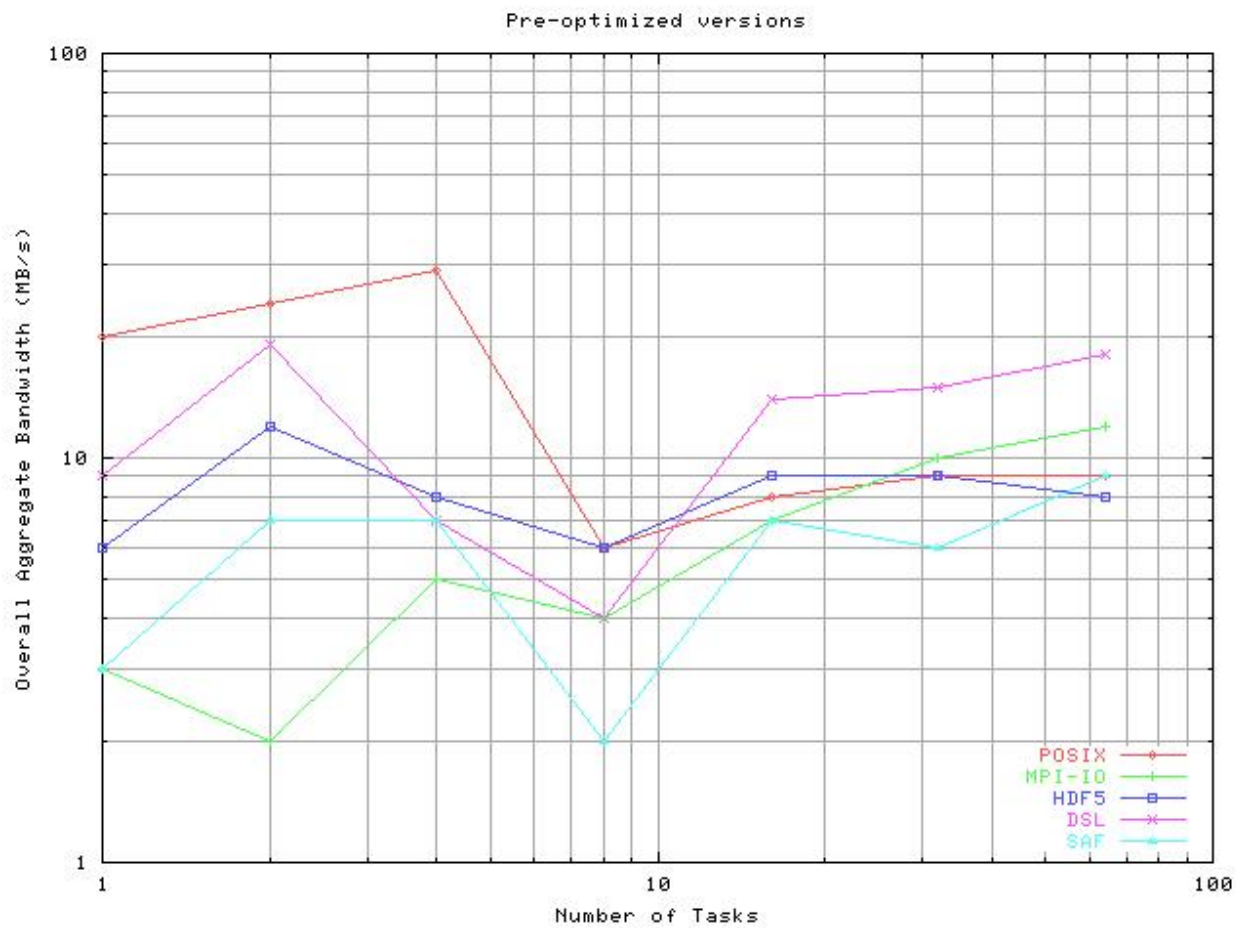
The SSlib library must be explicitly initialized before being used and should be finalized when the client is finished using it. In addition, this chapter contains some additional functions that operate on the library as a whole.

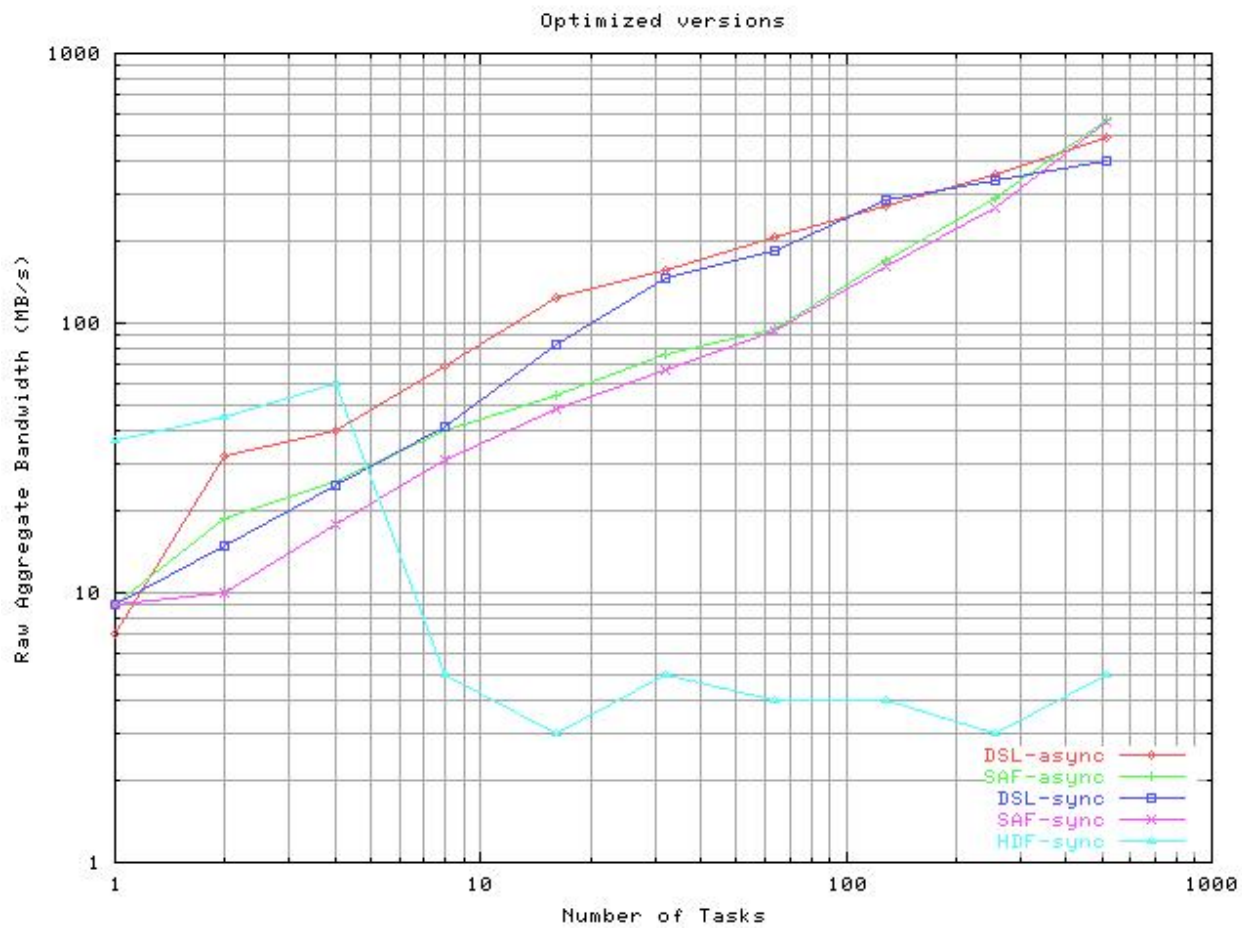
Members

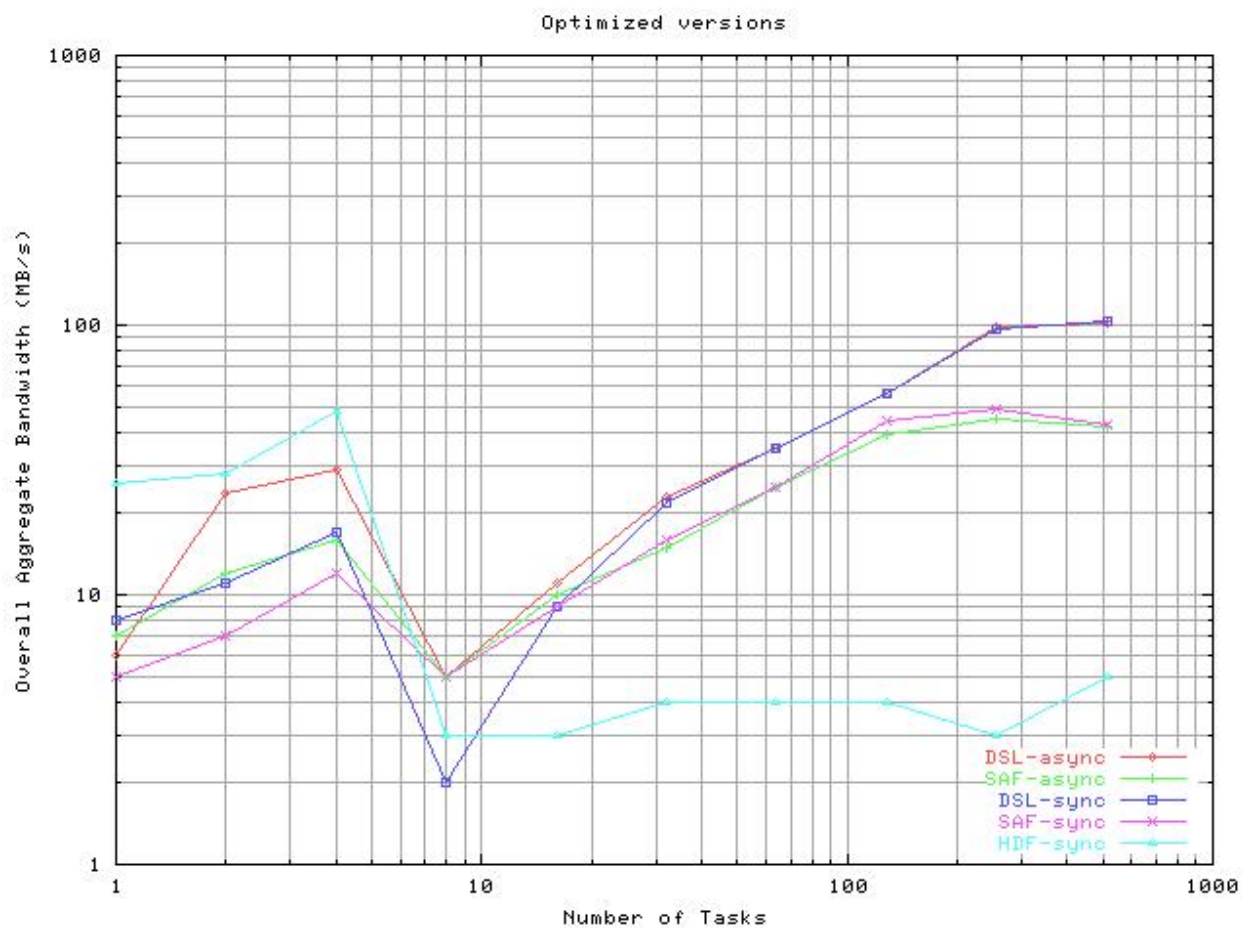
Initialize the library

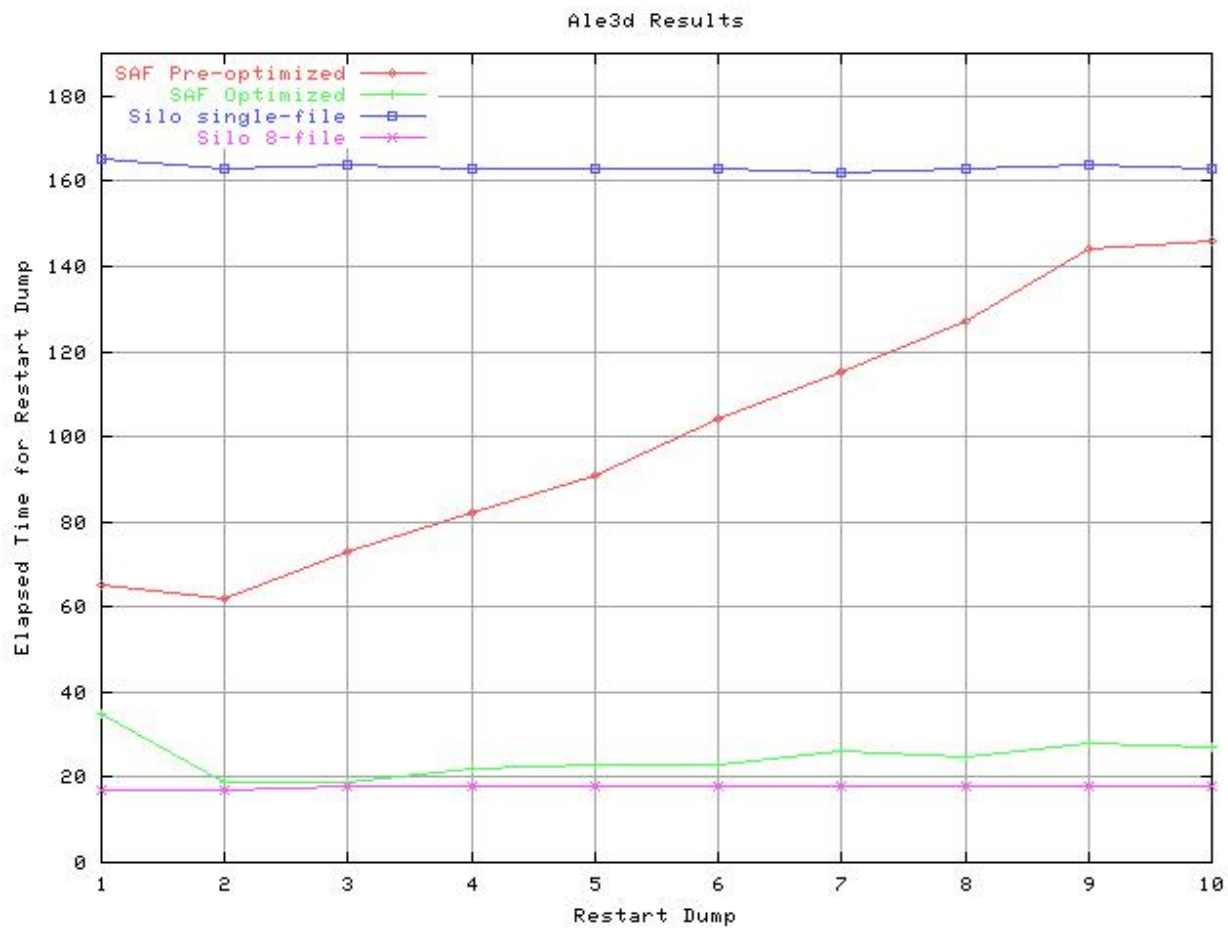
`ss_init_func` is a function defined in `sslib.c`.

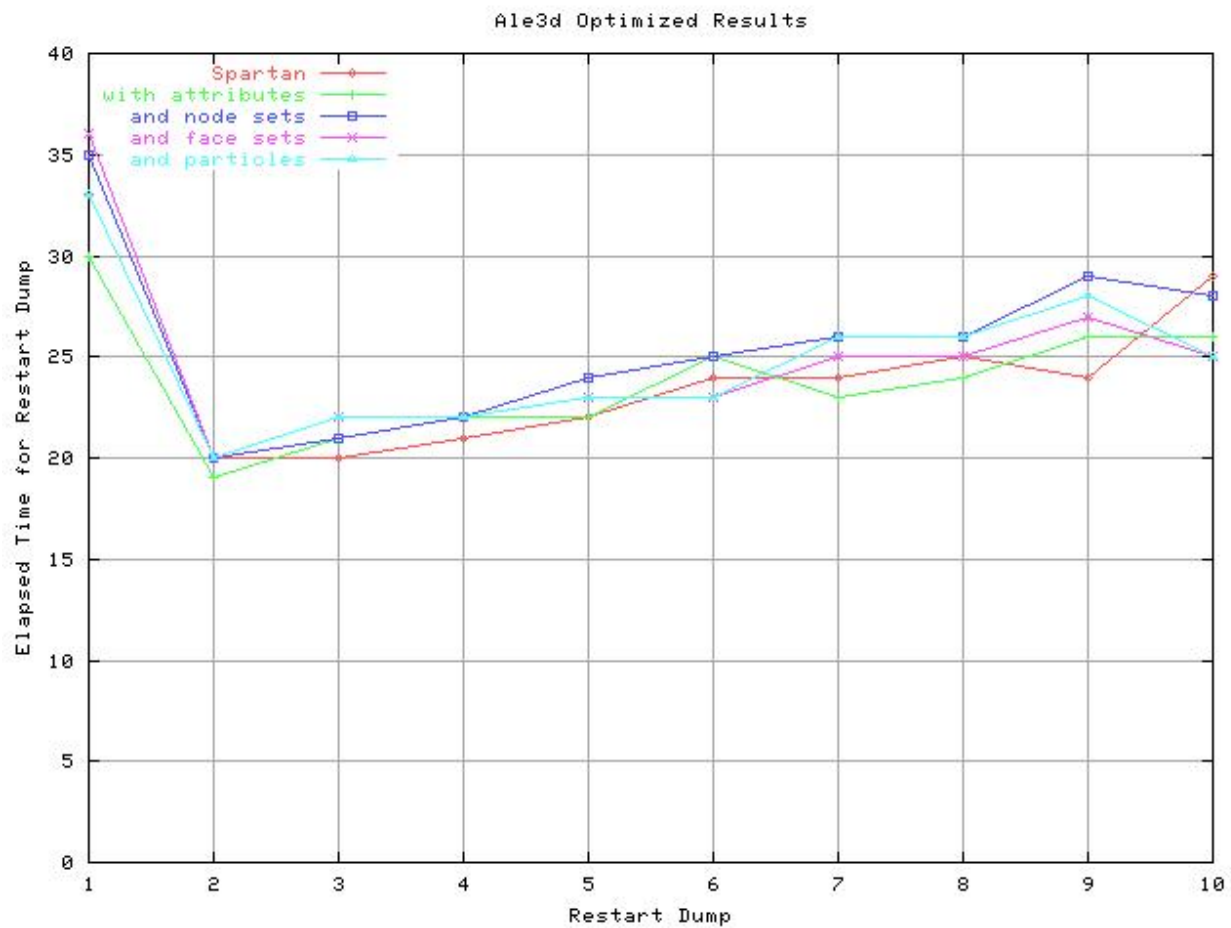
Synopsis:











herr_t **ss_init_func** (MPI_Comm *communicator*)

Formal Arguments:

- **communicator:** Library communicator defining the maximal set of MPI tasks that can be involved in various collective SSlib function calls. However, many collective SSlib functions can operate on a subset of this communicator. If SSlib is implicitly initialized then `MPI_COMM_WORLD` is assumed. When SSlib is compiled without MPI support then the `communicator` argument is just an integer that's ignored by this function.

Description: Call this function to initialize SSlib. It's use is entirely optional since SSlib generally initializes each layer of its software stack as it becomes necessary. However, if only a subset of the MPI tasks will be making calls to SSlib then this function can be invoked to define what tasks own the library.

This function initializes the collective parts of some other layers as well when those other layers are largely independent and might not have an opportunity to do their own collective initialization.

Normally the client initializes the library with the `ss_init` macro.

Return Value: Returns non-negative on success; negative on failure. It is an error to call this function more than one time or to call it after the library has been implicitly initialized.

Parallel Notes: Collective across the specified communicator.

Issues: We cannot pass things to this function with property lists since those property lists rely on SSlib having been initialized first.

See Also:

- [ss_init: 2.8: Initialize the library](#)
- [ss_initialized: 2.2: Test library initialization state](#)
- [Library](#): Introduction for current chapter

Test library initialization state

`ss_initialized` is a function defined in `sslib.c`.

Synopsis:

herr_t **ss_initialized** (void)

Description: Tests whether the library has been successfully initialized but not yet finalized.

Return Value: Returns true (positive) if the library has been initialized but not yet finalized and false otherwise. This function never fails and does not implicitly initialize the library.

Parallel Notes: Independent, although typically called collectively.

See Also:

- [Library](#): Introduction for current chapter

Terminate the library

`ss_finalize` is a function defined in `sslib.c`.

Synopsis:

herr_t **ss_finalize** (void)

Description: A call to this function will flush all pending data to the layers below SSLib, and then release as many resources as possible. This function is a no-op if called after a previous successful call or before the library is initialized (explicitly or implicitly). Obviously this function does not implicitly initialize the library.

Calling this function near the end of execution is strongly encouraged though not strictly necessary if all files have been explicitly flushed and/or closed. This function is suitable for registration with `atexit` provided care is taken to ensure that layers below SSLib are not finalized first.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the library's communicator.

See Also:

- [Library](#): Introduction for current chapter

Mark library as finalized

`ss_zap` is a function defined in `sslib.c`.

Synopsis:

`herr_t ss_zap (void)`

Description: Sometimes it's necessary to mark the library as finalized but without actually finalizing it. For instance, a call to `MPI_Abort` may eventually call `exit` (e.g., from `MPID_SHMEM_Abort`) which would cause SSLib's `atexit` handler to be invoked. But we really don't want that because the handler tries to do some MPI communication. So this function is supplied to make the `atexit` handler think `ss_finalize` has already been called.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the library's communicator, although this function does no MPI communication.

Issues: This function is here only because some libraries (e.g., MPICH's `MPI_Abort`) incorrectly call `exit` instead of calling `_exit`.

We should really set handlers back to their default values now and unmask them

See Also:

- [ss_finalize](#): 2.3: Terminate the library
- [Library](#): Introduction for current chapter

Start debugger for error

`ss_error` is a function defined in `sslib.c`.

Synopsis:

`herr_t ss_error (void)`

Description: This function gets called whenever SSLib pushes an error onto the error stack. If the error's identification number matches what the user wishes to debug as set with the `SSLIB_DEBUG` environment variable then the debugger is started. This function is also a useful place to set debugger breakpoints.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- [Library](#): Introduction for current chapter

Renders human readable numbers

`ss_bytes` is a function defined in `sslib.c`.

Synopsis:

```
char * ss_bytes (hsize_t nbytes, char *buf)
```

Formal Arguments:

- `buf`: Optional buffer to hold the results. If the user supplies the buffer then it should be large enough to hold the result. On a 64-bit machine that would be at least 62 bytes. If the caller passes the null pointer then one of six static buffers will be used (don't make more than six calls to this function in a single `printf` argument list).

Description: Often when printing a very large decimal number it's not obvious whether that number is some multiple of a power of 1024. This function breaks down the `nbytes` argument into GB, MB, and kB and stores it in a buffer supplied by the caller or a buffer allocated in this function. The output format will be something along the lines of:

```
4,197,386 (4M+3k+10)
```

If the `nbytes` has all bits set then it will be printed in hexadecimal. If it is less than 1024 then the parenthesised part will be omitted.

This function has no provision for limiting the size of the result string. The maximum string length on a 64-bit machine would be 62 bytes counting the NUL terminator:

```
##,###,###,###,###,###,### (##,###,###,###G+###M+###k+###)
```

Return Value: Returns a pointer to the NUL-terminated string containing the number on success; null on failure.

Parallel Notes: Independent

See Also:

- *Library*: Introduction for current chapter

Insert commas into an integer

`ss_insert_commas` is a function defined in `sslib.c`.

Synopsis:

```
char * ss_insert_commas (char *buf)
```

Description: Given a string containing a decimal integer, this function will insert commas between every third digit in order to make it more human readable.

Return Value: Returns `buf` after having modified it in place. Returns null on failure.

Parallel Notes: Independent

See Also:

- *Library*: Introduction for current chapter

Initialize the library

`ss_init` is a macro defined in `sslib.h`.

Synopsis:

ss_init (COMM)

Description: This is a macro around *ss_init_func* that also checks header/library compatibility.

Return Value: Same as *ss_init_func*

Parallel Notes: Same as *ss_init_func*

See Also:

- *ss_init_func*: 2.1: *Initialize the library*
- *Library*: Introduction for current chapter

Environment Variables

No description available.

Members

Environment Variables

SSLIB is a collection of related C preprocessor symbols defined in *ssenv.h*.

Synopsis:

SSLIB_DEBUG: See *ss_debug_env* for details

SSLIB_2PIO: See *ss_blob_set_2pio* for details

Description: SSlib is controlled by various environment variables documented here.

See Also:

- *Environment Variables*: Introduction for current chapter

Error Handling

SSlib functions and the functions that SSlib calls all indicate errors by returning special values. SSlib code inspects return values and enters an error recovery mode when an error is detected. In order to make the SSlib code base cleaner and to facilitate changes in error handling policies, we use a number of macros. The goal is to have a system that is easy to program, easy to optimize, and easy to read (lean, mean, and clean).

Almost every function will begin and end with *SS_ENTER* and *SS_LEAVE* calls. Somewhere in between them will be an *SS_CLEANUP* label that marks the boundary between normal flow of control and error recovery code. Inside the normal flow of control will be calls to *SS_ERROR* or *SS_ERROR_FMT* when an error is detected, or various calls to *SS_STATUS* macros to get information about an error.

Sometimes, particularly during a parallel run, error recovery is impossible, prohibitively expensive, or unusually complex. In such cases SSlib may call *MPI_Abort*. (A version of SSlib compiled with MPI support but run with a single MPI task as the library communicator is considered a serial run.)

Members

Minor error numbers

SS_MINOR is a collection of related C preprocessor symbols defined in *sserr.h*.

Synopsis:

SS_MINOR_ASSERT: assertion failed
 SS_MINOR_CONS: constructor failed
 SS_MINOR_CORRUPT: file file corruption
 SS_MINOR_DOMAIN: value outside valid domain
 SS_MINOR_EXISTS: already exists
 SS_MINOR_FAILED: a catch-all
 SS_MINOR_HDF5: HDF5 call failed
 SS_MINOR_INIT: not initialized
 SS_MINOR_MPI: MPI call failed
 SS_MINOR_NOTFOUND: not found
 SS_MINOR_NOTIMP: not implemented
 SS_MINOR_NOTOPEN: not open
 SS_MINOR_OVERFLOW: arithmetic or buffer overflow
 SS_MINOR_PERM: not permitted
 SS_MINOR_RESOURCE: insufficient resources
 SS_MINOR_SKIPPED: operation skipped by request
 SS_MINOR_TYPE: bad datatype
 SS_MINOR_USAGE: incorrect usage

Issues: These C preprocessor symbols would normally just be defined as the corresponding global variable, however error class numbers are generated at runtime with HDF5 calls and thus they must be initialized as a side effect of referencing them. The initialization is done by calling `ss_err_init1` and passing the address of the global minor error class variable. That function does the initialization and then returns the contents of that global variable.

See Also:

- [Error Handling](#): Introduction for current chapter

Asserts object runtime type

SS_ASSERT_TYPE is a macro defined in `sserr.h`.

Synopsis:

SS_ASSERT_TYPE (`_obj_`, `_type_`)

Description: The first argument should be an object of the specified type, valid at runtime. If it isn't then a TYPE error is raised. The `_type_` argument is a C datatype like `ss_fieldtmpl_t`.

See Also:

- [Error Handling](#): Introduction for current chapter

Asserts object runtime type and existence

`SS_ASSERT_MEM` is a macro defined in `sserr.h`.

Synopsis:

`SS_ASSERT_MEM` (`_obj_`, `_type_`)

Description: The first argument should be an object of the specified type, valid at runtime. If it isn't then a `TYPE` error is raised. The `_type_` argument is a C datatype like `ss_fieldtmpl_t`. This function is like `SS_ASSERT_TYPE` except it also checks that the `_obj_` exists in memory (i.e., the object can be dereferenced).

See Also:

- [SS_ASSERT_TYPE](#): 4.2: Asserts object runtime type
- [Error Handling](#): Introduction for current chapter

Asserts object runtime class

`SS_ASSERT_CLASS` is a macro defined in `sserr.h`.

Synopsis:

`SS_ASSERT_CLASS` (`_obj_`, `_cls_`)

Description: The first argument should be an object of the specified class, valid at runtime. If it isn't then a `TYPE` error is raised. The `_cls_` argument should be a C class datatype like `ss_pers_t`.

See Also:

- [Error Handling](#): Introduction for current chapter

Begin a functionality test

`SS_CHECKING` is a macro defined in `sserr.h`.

Synopsis:

`SS_CHECKING` (`_what_`)

Description: This family of macros can be used in the SSlib test suite to perform a test of some functionality. The `SS_CHECKING` and `SS_END_CHECKING` macros should be paired with curly braces. Inside the body of that construct may be zero or more calls to `SS_FAILED` or `SS_FAILED_WHEN`. If either of the failure macros is executed flow control branches to the `SS_END_CHECKING` macro.

The `SS_END_CHECKING_WITH` macro can be used in place of `SS_END_CHECKING`. It takes a single argument which is arbitrary code to execute if an error was detected in the body of the `SS_CHECKING` construct. Typically the argument will be something along the lines of `return "FAILURE"` or `'goto error'`.

The argument for `SS_CHECKING` should be a string that will be printed to `stderr` after the word "checking". The string is printed only if `_print` is non-zero (similarly for the output from related macros).

```
1 FILE *_print = 0==self ? stderr : NULL;
2 int nerrors=0;
3 SS_CHECKING("file opening operations") {
4     file1 = ss_file_create(...);
5     if (!file1) SS_FAILED_WHEN("creating");
6     file2 = ss_file_open(...);
```

(continues on next page)

(continued from previous page)

```

7      if (!file2) SS_FAILED_WHEN("opening");
8  } SS_END_CHECKING_WITH(nerrors++);

```

```

1  SS_CHECKING("file close") {
2      if (ss_file_close(...)<0) SS_FAILED;
3  } SS_END_CHECKING;

```

See Also:

- [*SS_END_CHECKING*](#): 4.10: *End functionality test*
- [*SS_END_CHECKING_WITH*](#): 4.11: *End functionality test*
- [*SS_FAILED*](#): 4.6: *Indicate functionality test failure*
- [*SS_FAILED_WHEN*](#): 4.7: *Indicate functionality test failure*
- [*Error Handling*](#): Introduction for current chapter

Indicate functionality test failure

SS_FAILED is a symbol defined in sserr.h.

Synopsis:

SS_FAILED

Description: See [*SS_CHECKING*](#)

See Also:

- [*SS_CHECKING*](#): 4.5: *Begin a functionality test*
- [*Error Handling*](#): Introduction for current chapter

Indicate functionality test failure

SS_FAILED_WHEN is a macro defined in sserr.h.

Synopsis:

SS_FAILED_WHEN (_mesg_)

Description: See [*SS_CHECKING*](#)

See Also:

- [*SS_CHECKING*](#): 4.5: *Begin a functionality test*
- [*Error Handling*](#): Introduction for current chapter

Indicate functionality test skipped

SS_SKIPPED is a symbol defined in sserr.h.

Synopsis:

SS_SKIPPED

Description: See *SS_CHECKING*

See Also:

- *SS_CHECKING*: 4.5: *Begin a functionality test*
- *Error Handling*: Introduction for current chapter

Indicate functionality test skipped

`SS_SKIPPED_WHEN` is a macro defined in `sserr.h`.

Synopsis:

`SS_SKIPPED_WHEN` (`_mesg_`)

Description: See *SS_CHECKING*

See Also:

- *SS_CHECKING*: 4.5: *Begin a functionality test*
- *Error Handling*: Introduction for current chapter

End functionality test

`SS_END_CHECKING` is a symbol defined in `sserr.h`.

Synopsis:

`SS_END_CHECKING`

Description: See *SS_CHECKING*

See Also:

- *SS_CHECKING*: 4.5: *Begin a functionality test*
- *Error Handling*: Introduction for current chapter

End functionality test

`SS_END_CHECKING_WITH` is a macro defined in `sserr.h`.

Synopsis:

`SS_END_CHECKING_WITH` (`_code_`)

Description: See *SS_CHECKING*

See Also:

- *SS_CHECKING*: 4.5: *Begin a functionality test*
- *Error Handling*: Introduction for current chapter

Magic Numbers

Many of the SSlib data structures have an `unsigned int` member that will contain a magic number while the struct is allocated. The magic number serves to run-time type the struct. The most significant 12 bits are 0x5af (looks sort of like “Saf”). The next eight bits are a type class number (e.g., all storable object handles belong to a certain class). The least significant 12 bits are a unique sequence number for that particular type class and are sometimes used as indices into various arrays.

Members

Miscellaneous (class 0x5af01000)

`SS_MAGIC_ss` is a collection of related C preprocessor symbols defined in `ssobj.h`.

Synopsis:

```
SS_MAGIC_ss_prop_t:
SS_MAGIC_ss_table_t:
SS_MAGIC_ss_string_table_t:
SS_MAGIC_ss_gblob_t:
```

See Also:

- *Magic Numbers*: Introduction for current chapter

Persistent object links (class 0x5af02000)

`SS_MAGIC_ss` is a collection of related C preprocessor symbols defined in `ssobj.h`.

Synopsis:

```
SS_MAGIC_ss_pers_t: just the class part
SS_MAGIC_ss_scope_t:
SS_MAGIC_ss_field_t:
SS_MAGIC_ss_role_t:
SS_MAGIC_ss_basis_t:
SS_MAGIC_ss_algebraic_t:
SS_MAGIC_ss_evaluation_t:
SS_MAGIC_ss_relrep_t:
SS_MAGIC_ss_quantity_t:
SS_MAGIC_ss_unit_t:
SS_MAGIC_ss_cat_t:
SS_MAGIC_ss_collection_t:
SS_MAGIC_ss_set_t:
SS_MAGIC_ss_rel_t:
SS_MAGIC_ss_fieldtmpl_t:
```

SS_MAGIC_ss_tops_t:
SS_MAGIC_ss_blob_t:
SS_MAGIC_ss_indexspec_t:
SS_MAGIC_ss_file_t:
SS_MAGIC_ss_attr_t:

Description: These are the magic numbers for persistent object links, which are the handles to persistent objects that the client usually works with.

Issues: These magic numbers must be in the same order as the persistent object magic numbers (class 0x5af03000).

See Also:

- *Magic Numbers*: Introduction for current chapter

Persistent objects (class 0x5af03000)

SS_MAGIC_ss is a collection of related C preprocessor symbols defined in ssobj.h.

Synopsis:

SS_MAGIC_ss_persobj_t: just the class part
SS_MAGIC_ss_scopeobj_t:
SS_MAGIC_ss_fieldobj_t:
SS_MAGIC_ss_roleobj_t:
SS_MAGIC_ss_basisobj_t:
SS_MAGIC_ss_algebraicobj_t:
SS_MAGIC_ss_evaluationobj_t:
SS_MAGIC_ss_relrepobj_t:
SS_MAGIC_ss_quantityobj_t:
SS_MAGIC_ss_unitobj_t:
SS_MAGIC_ss_catobj_t:
SS_MAGIC_ss_collectionobj_t:
SS_MAGIC_ss_setobj_t:
SS_MAGIC_ss_relobj_t:
SS_MAGIC_ss_fieldtmplobj_t:
SS_MAGIC_ss_topsobj_t:
SS_MAGIC_ss_blobobj_t:
SS_MAGIC_ss_indexspecobj_t:
SS_MAGIC_ss_fileobj_t:
SS_MAGIC_ss_attrobj_t:

Description: These are the magic numbers for the persistent objects themselves. They do not appear in the file but are part of the transient information for an object. The order of things here is such that when synchronizing a scope we

minimize the number of forward references. That is, if objects of type A can point to objects of type B then we should synchronize type B before type A.

Issues: These magic numbers must be in the same order as the persistent object link magic numbers (class 0x5af02000). Also, they are mentioned in `ss_pers_init` when constructing an HDF5 enumeration datatype.

If you add items here and they don't show up as tables in the files then the `SS_PERS_NCLASSES` constant defined in `sspers.h` is probably not large enough.

See Also:

- *Magic Numbers*: Introduction for current chapter

Obtain magic number for type

`SS_MAGIC` is a macro defined in `ssobj.h`.

Synopsis:

`SS_MAGIC` (`_type_`)

Description: Returns the magic number for the specified SSlib datatype.

Return Value: Returns an unsigned magic number.

See Also:

- *Magic Numbers*: Introduction for current chapter

Determine magicness

`SS_MAGIC_OK` is a macro defined in `ssobj.h`.

Synopsis:

`SS_MAGIC_OK` (`M`)

Description: Determines if number or class `M` looks magic.

Return Value: True if `M` is probably a magic number; false otherwise.

See Also:

- *Magic Numbers*: Introduction for current chapter

Obtain magic number class

`SS_MAGIC_CLASS` is a macro defined in `ssobj.h`.

Synopsis:

`SS_MAGIC_CLASS` (`M`)

Description: Given a magic number `M`, return the class part by masking off the low-order 12 bits.

Return Value: An unsigned int magic class number.

See Also:

- *Magic Numbers*: Introduction for current chapter

Obtain magic sequence number

`SS_MAGIC_SEQUENCE` is a macro defined in `ssobj.h`.

Synopsis:

`SS_MAGIC_SEQUENCE` (M)

Description: Given a magic number M, return the sequence number stored in the 12 low-order bits.

Return Value: An `unsigned int` magic sequence number.

See Also:

- *Magic Numbers*: Introduction for current chapter

Construct a magic number

`SS_MAGIC_CONS` is a macro defined in `ssobj.h`.

Synopsis:

`SS_MAGIC_CONS` (C, S)

Description: Given a magic class number like what is returned by *`SS_MAGIC_CLASS`* and a sequence number like what is returned by *`SS_MAGIC_SEQUENCE`*, construct a magic number. The class, C, and sequence, S, don't have to be purely a class or sequence because they'll be filtered.

Return Value: An `unsigned int` magic number constructed from a class and sequence number.

See Also:

- *`SS_MAGIC_CLASS`*: 5.6: Obtain magic number class
- *`SS_MAGIC_SEQUENCE`*: 5.7: Obtain magic sequence number
- *Magic Numbers*: Introduction for current chapter

Obtain magic number from a pointer

`SS_MAGIC_OF` is a macro defined in `ssobj.h`.

Synopsis:

`SS_MAGIC_OF` (OBJ)

Description: Given a pointer to any object, return the magic number stored in that object.

Return Value: An `unsigned int` magic number or zero if OBJ is the null pointer.

See Also:

- *Magic Numbers*: Introduction for current chapter

Properties

SSlib uses property lists to pass miscellaneous information to various functions, to pass properties through entire layers of the library, and as a mechanism to message pass information between MPI tasks. It relies heavily on HDF5's datatype description interface and data conversion functionality. The property list class `ss_prop_t` is a subclass of `ss_obj_t` and therefore has a magic number (see **Magic Numbers**).

Property lists are created and destroyed with *ss_prop_new*, *ss_prop_dup*, *ss_prop_cons*, and *ss_prop_dest*.

Once a property list exists, properties can be added and initialized to default values with *ss_prop_add*. The various *ss_prop_set* and *ss_prop_get* functions and their variants set and retrieve values of individual properties. The *ss_prop_buffer* function is similar to the *ss_prop_get* functions, but returns a pointer without copying the memory, and *ss_prop_type* returns a datatype instead of the value.

The *ss_prop_appendable*, *ss_prop_modifiable*, and *ss_prop_immutable* define (or query) what operations can be performed on a property list.

Members

Create a new property list from scratch

ss_prop_new is a function defined in *ssprop.c*.

Synopsis:

```
ss_prop_t * ss_prop_new (const char *name)
```

Formal Arguments:

- *name*: optional name for debugging

Description: Creates a new property list and initializes it. An optional *name* may be specified and is used only for debugging.

Return Value: Returns new property list on success; null on failure.

Parallel Notes: Independent

See Also:

- *Properties*: Introduction for current chapter

Create a new property list from an existing list

ss_prop_dup is a function defined in *ssprop.c*.

Synopsis:

```
ss_prop_t * ss_prop_dup (ss_prop_t *prop, const char *name)
```

Formal Arguments:

- *prop*: source property list
- *name*: optional name for debugging

Description: Duplicates an existing property list, giving it a new name (or a name generated from the source property list). The new property list's property names, values, and datatypes are copied from the specified *prop* list. The new list is marked as appendable (new properties can be added) and modifiable (property values can be changed).

Return Value: Returns a new property list on success; null on failure.

Parallel Notes: Independent

See Also:

- *Properties*: Introduction for current chapter

Property constructor

`ss_prop_cons` is a function defined in `ssprop.c`.

Synopsis:

`ss_prop_t * ss_prop_cons (hid_t type, void *values, const char *name)`

Formal Arguments:

- `type`: Property datatype (copied by this function)
- `values`: Optional initial values, of type `type`
- `name`: Optional property list name

Description: Construct a property from a datatype and optional memory. The datatype must be an HDF5 compound datatype and the memory must match that datatype. If a `values` buffer is supplied then the property list will be marked as non-appendable (new properties cannot be added) but modifiable (property values can be changed) and the `ss_prop_dest` function will not free the memory. But if `values` is null then a buffer will be allocated and initialized to zeros.

Return Value: Returns a new property list on success; null on failure.

Parallel Notes: Independent

See Also:

- `ss_prop_dest`: 6.4: *Destroy a property list*
- *Properties*: Introduction for current chapter

Destroy a property list

`ss_prop_dest` is a function defined in `ssprop.c`.

Synopsis:

`herr_t ss_prop_dest (ss_prop_t *prop)`

Description: All resources associated with the specified property list are released and the property list is marked as invalid and should not be referenced. If the caller had not supplied a buffer to hold the values then that memory is also freed.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Properties*: Introduction for current chapter

Add new property to a list

`ss_prop_add` is a function defined in `ssprop.c`.

Synopsis:

`herr_t ss_prop_add (ss_prop_t *prop, const char *name, hid_t type, const void *value)`

Formal Arguments:

- `prop`: property list to which is added a property

- `name`: name of property to add
- `type`: datatype for stored property value
- `value`: optional initial property value

Description: A new property called `name` is added to property list `prop`. The new property can be given an initial value (or else all bits are cleared). If a value is specified it must be of type `type`, which is the datatype of the property as stored in the property list.

Return Value: Returns non-negative on success; negative on failure. It is an error to add a property to the list if a property by that name already exists in the list. Only appendable property lists can have new properties added.

See Also:

- [Properties](#): Introduction for current chapter

Determine if property exists

`ss_prop_has` is a function defined in `ssprop.c`.

Synopsis:

```
htri_t ss_prop_has (ss_prop_t *prop, const char *name)
```

Formal Arguments:

- `prop`: property list being queried
- `name`: name of property to be tested

Description: This function determines if a property by the specified `name` exists in the property list.

Return Value: Returns true (positive) if `name` exists in `prop` and false if not. Returns negative on failure.

Parallel Notes: Independent

See Also:

- [Properties](#): Introduction for current chapter

Change a property value

`ss_prop_set` is a function defined in `ssprop.c`.

Synopsis:

```
herr_t ss_prop_set (ss_prop_t *prop, const char *name, hid_t type, const void *value)
```

Formal Arguments:

- `prop`: property list to be modified
- `name`: optional name of property to be modified
- `type`: optional datatype for supplied value
- `value`: optional new property value

Description: A property's value can be modified by supplying a new `value` with this function. If `type` is specified then the `value` will be of this type, which must be conversion-compatible with the type declared when the property was added to the list. Otherwise, when no `type` is specified the `value` should be of the type originally specified to [ss_prop_add](#). If `name` is null then `type` and `value` refer to the entire property list rather than a single property. The

`value` is copied into the property list, and if `value` is a null pointer then the property (or entire property list if `name` is null) is reset to zero.

Return Value: Returns non-negative on success; negative on failure. The property list must be marked as “modifiable” in order to change a property value. If the specified `type` is not conversion compatible with the stored type then the property is not modified.

Parallel Notes: Independent

See Also:

- [*ss_prop_add*](#): 6.5: *Add new property to a list*
- [*Properties*](#): Introduction for current chapter

Change a signed integer property value

`ss_prop_set_i` is a function defined in `ssprop.c`.

Synopsis:

```
herr_t ss_prop_set_i (ss_prop_t *prop, const char *name, int value)
```

Formal Arguments:

- `prop`: property list to be modified
- `name`: name of property to be modified
- `value`: new integer value

Description: This is a convenience function for modifying an integer-valued property. See [*ss_prop_set*](#) for details.

Return Value: See [*ss_prop_set*](#).

Parallel Notes: Independent

See Also:

- [*ss_prop_set*](#): 6.7: *Change a property value*
- [*Properties*](#): Introduction for current chapter

Change an unsigned integer property value

`ss_prop_set_u` is a function defined in `ssprop.c`.

Synopsis:

```
herr_t ss_prop_set_u (ss_prop_t *prop, const char *name, size_t value)
```

Formal Arguments:

- `prop`: property list to be modified
- `name`: name of property to be modified
- `value`: new unsigned (`size_t`) value

Description: This is a convenience function for modifying an unsigned integer-valued property. See [*ss_prop_set*](#) for details.

Return Value: See [*ss_prop_set*](#).

Parallel Notes: Independent

See Also:

- *ss_prop_set*: 6.7: *Change a property value*
- *Properties*: Introduction for current chapter

Change a floating-point property value

`ss_prop_set_f` is a function defined in `ssprop.c`.

Synopsis:

```
herr_t ss_prop_set_f (ss_prop_t *prop, const char *name, double value)
```

Formal Arguments:

- `prop`: property list to be modified
- `name`: name of property to be modified
- `value`: new floating-point value

Description: This is a convenience function for modifying a floating-point property. See *ss_prop_set* for details.

Return Value: See *ss_prop_set*.

Parallel Notes: Independent

See Also:

- *ss_prop_set*: 6.7: *Change a property value*
- *Properties*: Introduction for current chapter

Query a property value

`ss_prop_get` is a function defined in `ssprop.c`.

Synopsis:

```
void * ss_prop_get (ss_prop_t *prop, const char *name, hid_t type, void *buffer)
```

Formal Arguments:

- `prop`: property list to be queried
- `name`: name of queried property
- `type`: optional type of data to return
- `buffer`: optional result buffer

Description: The value of a property (or entire property list) can be queried by providing a handle to the property list and the name of the property (or `NULL`). If a datatype is supplied then it must be conversion-compatible with the declared property (or property list) datatype, and the result will be returned as the specified datatype, otherwise the result is returned in the original datatype. If a buffer is supplied then the value is copied into the buffer (the caller must ensure that the buffer is large enough), otherwise a buffer is allocated for the result.

Return Value: On success, returns either the supplied buffer or an newly allocated buffer which the caller should eventually free. Returns the null pointer on failure.

Parallel Notes: Independent

See Also:

- *Properties*: Introduction for current chapter

Query an integer property

`ss_prop_get_i` is a function defined in `ssprop.c`.

Synopsis:

```
int ss_prop_get_i (ss_prop_t *prop, const char *name)
```

Formal Arguments:

- `prop`: property list to be queried
- `name`: name of queried property

Description: This is a convenience function for querying integer-valued properties. See [*ss_prop_get*](#) for details.

Return Value: Returns -1 on failure. Since this can also be a valid property value, the caller should examine the error stack to determine if an error in fact occurred.

Parallel Notes: Independent

See Also:

- [*ss_prop_get*](#): 6.11: *Query a property value*
- *Properties*: Introduction for current chapter

Query an unsigned integer property

`ss_prop_get_u` is a function defined in `ssprop.c`.

Synopsis:

```
size_t ss_prop_get_u (ss_prop_t *prop, const char *name)
```

Formal Arguments:

- `prop`: property list to be queried
- `name`: name of queried property

Description: This is a convenience function for querying unsigned integer-valued properties. See [*ss_prop_get*](#) for details.

Return Value: Returns `SS_NOSIZE` on failure. Since this can also be a valid property value, the caller should examine the error stack to determine if an error in fact occurred.

Parallel Notes: Independent

See Also:

- [*ss_prop_get*](#): 6.11: *Query a property value*
- *Properties*: Introduction for current chapter

Query a floating point property

`ss_prop_get_f` is a function defined in `ssprop.c`.

Synopsis:

double **ss_prop_get_f** (ss_prop_t *prop, const char *name)

Formal Arguments:

- prop: property list to be queried
- name: name of queried property

Description: This is a convenience function for querying floating point properties. See [ss_prop_get](#) for details.

Return Value: Returns negative on failure. Since this can also be a valid property value, the caller should examine the error stack to determine if an error in fact occurred.

Parallel Notes: Independent

See Also:

- [ss_prop_get](#): 6.11: *Query a property value*
- [Properties](#): Introduction for current chapter

Obtain pointer direct to value

ss_prop_buffer is a function defined in ssprop.c.

Synopsis:

void * **ss_prop_buffer** (ss_prop_t *prop, const char *name)

Formal Arguments:

- prop: property list to be queried
- name: optional property name

Description: This function is similar to [ss_prop_get](#) except instead of copying the value or values into a new buffer, it returns a pointer directly into the property list values buffer. If a property name is specified then the pointer is to the beginning of the specified property, otherwise the pointer is to the beginning of the entire property values buffer.

Return Value: Returns a pointer into the buffer holding property values on success; null on failure. If the client supplied the buffer for the values via the [ss_prop_cons](#) function then the returned pointer is valid at least until the property list is destroyed, otherwise the pointer is valid only until the list is destroyed or a new property is added, whichever occurs first.

Parallel Notes: .. _SC_ss_prop_buffer:

1 *

See Also:

- [ss_prop_cons](#): 6.3: *Property constructor*
- [ss_prop_get](#): 6.11: *Query a property value*
- [Properties](#): Introduction for current chapter

Query the datatype of a property or property list

ss_prop_type is a function defined in ssprop.c.

Synopsis:

hid_t **ss_prop_type** (ss_prop_t *prop, const char *name)

Formal Arguments:

- `prop`: property list to be queried
- `name`: optional property name

Description: When given a property list and a property name, return the HDF5 datatype that was originally used to declare the property. If no property name is specified then return the HDF5 datatype of the whole property list (which is guaranteed to be a compound datatype).

Return Value: Returns an HDF5 datatype on success; negative on failure. The client should eventually call `H5Tclose` on the returned datatype. It is a failure to try to obtain the datatype of an empty property.

Parallel Notes: Independent

See Also:

- [Properties](#): Introduction for current chapter

Queries/sets property list appendability

`ss_prop_appendable` is a function defined in `ssprop.c`.

Synopsis:

`htri_t ss_prop_appendable (ss_prop_t *prop, htri_t new_value)`

Description: Queries and/or sets whether the property list is appendable. That is, whether new properties can be added to the property list. If `new_value` is negative then the appendability status remains unchanged, otherwise it is set to the new value. Once a property list is marked as not appendable it cannot be later marked as appendable.

Return Value: Returns true (positive) if the property list was appendable before this call, false otherwise. Returns negative on failure. It is an error to attempt to make a non-appendable property list appendable.

Parallel Notes: Independent

See Also:

- [Properties](#): Introduction for current chapter

Queries/sets property list modifiability

`ss_prop_modifiable` is a function defined in `ssprop.c`.

Synopsis:

`htri_t ss_prop_modifiable (ss_prop_t *prop, htri_t new_value)`

Description: Queries and/or sets whether the property list is modifiable. That is, whether property values can be changed. If `new_value` is negative then the modifiability status remains unchanged, otherwise it is set to the new value. Once a property list is marked as not modifiable it cannot be later marked as modifiable.

Return Value: Returns true (positive) if the property list was modifiable before this call, false otherwise. Returns negative on failure. It is an error to attempt to make a non-modifiable property list modifiable.

Parallel Notes: Independent

See Also:

- [Properties](#): Introduction for current chapter

Make a property list immutable

`ss_prop_immutable` is a function defined in `ssprop.c`.

Synopsis:

```
herr_t ss_prop_immutable (ss_prop_t *prop)
```

Description: This function that marks a property list as non-appendable and non-modifiable and non-destroyable. Once a property list is marked this way it is said to be immutable and cannot be changed in any way (not even destroyed except when the library is closed).

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Properties*: Introduction for current chapter

Persistent Objects

A persistent object is anything that typically gets stored in an SSlib database. Examples are sets, fields, relations, templates thereof, etc. For each class of persistent object SSlib creates a *link* datatype, which is akin to a C pointer in that it's a lightweight piece of data (a small C struct with no dynamically allocated components). But it's also more than a C pointer because it can point to disk-resident objects, and the links can be stored in files. Also like C pointers, the object to which a link points must be dereferenced in order to get to the actual object, and SSlib provides macros for doing such. In fact, SSlib provides three macros for each persistent object class.

SS_FIELD: Takes one argument, *field*, of type `ss_field_t` and dereferences that link to obtain a pointer to the C struct that implements the field object. The returned value is of type `ss_fieldobj_t`. The same pattern is followed for other persistent classes.

SS_FIELD_M: Just like **SS_FIELD** except it also takes a member name whose value it returns via C's arrow operator. There really isn't much point in using this macro but it's supplied for completeness.

SS_FIELD_P: Just like **SS_FIELD_M** except it returns the address of the member instead of the member value. This macro is used most often to obtain a pointer to a link that's stored in some object because objects store links but almost all SSlib functions take pointers to links. It's only supplied for completeness because one can do the same thing by using the "address of" (ampersand) C operator in front of **SS_FIELD_M**.

Generally the [SAF](#) library will pass around links to objects and dereference the link whenever access to members of the actual object's struct is necessary. The primary reason for passing around links instead of object pointers is that it allows SSlib to relocate objects in memory in order to do certain memory management tasks and optimizations. However, it's not really a big performance penalty to repeatedly dereference links because the link caches the pointer in such a way that most dereferencing operations will incur only two memory comparisons. And precisely for that reason it is normal practice to pass non-const-qualified pointers to links when calling most functions: it allows SSlib library to update the link itself and propagate that change up through the call stack.

Members

Create a new persistent object

`ss_pers_new` is a function defined in `sspers.c`.

Synopsis:

```
ss_pers_t * ss_pers_new(ss_scope_t *scope, unsigned tableid, const ss_persobj_t *init, unsigned flags,  
                        ss_pers_t *buf, ss_prop_t UNUSED *props)
```

Formal Arguments:

- `scope`: The scope that will own this new object.
- `tableid`: A magic number whose sequence part defines a table
- `init`: Optional initial data of type `ss_persobj_t` or a type derived therefrom. The type must be appropriate for the class of object being created. This argument can be used to copy a persistent object. *ISSUE*: Should this be a link instead?
- `flags`: Creation flags, like *SS_ALLSAME*
- `buf`: Optional buffer for return value
- `props`: Additional properties (none defined yet)

Description: Creates a new persistent object of the specified object type in the specified scope. Normally this function assumes that every caller could be creating its own object and the table synchronization algorithm will determine later how many objects were actually created by comparing their contents. However, synchronization can be an expensive operation which can be avoided when the caller knows that all tasks of the scope's communicator are participating to create a single object. This situation is noted by passing the *SS_ALLSAME* bit in the `flags` argument.

Return Value: Returns a link to the new object on success; the null pointer on failure. If `buf` is supplied then that will be the successful return value, otherwise a persistent object link will be allocated.

Parallel Notes: Independent or collective. This function must be collective across the scope's communicator (although communication-free) if the *SS_ALLSAME* bit is passed in the `flags` argument. In other words, if all tasks are participating to create one single object, then the call must be collective if we wish to avoid the synchronization costs later. However, it is still possible for all tasks to create one single object independently (i.e., creation order doesn't matter) if they don't pass *SS_ALLSAME* and they don't mind paying the synchronization cost later.

See Also:

- *ss_pers_copy*: 7.2: Copy an object
- *Persistent Objects*: Introduction for current chapter

Copy an object

`ss_pers_copy` is a function defined in `sspers.c`.

Synopsis:

```
ss_pers_t * ss_pers_copy(ss_pers_t *pers, ss_scope_t *scope, unsigned flags, ss_pers_t *buf,  
                        ss_prop_t *props)
```

Formal Arguments:

- `pers`: The object to be copied.
- `scope`: The destination scope that will own the new object.
- `flags`: Creation flags like *SS_ALLSAME* (see *ss_pers_new*).
- `buf`: Optional buffer for return value.
- `props`: Additional properties (none defined yet)

Description: Copy the given object and return a link to it. If the object contains memory that needs to be copied (like character strings or variable length arrays) then those are copied also. Other objects to which the original pointed are not copied – the new object has links to the same ones.

Return Value: Returns a link to the new object on success; the null pointer on failure. If `buf` is supplied then that will be the successful return value, otherwise a persistent object link will be allocated.

Parallel Notes: Independent or collective. This function must be collective across the scope's communicator (although communication-free) if the `SS_ALLSAME` bit is passed in the `flags` argument. In other words, if all tasks are participating to create one single new object then the call must be collective if we wish to avoid the synchronization costs later. However, it is still possible for all tasks to create one single object independently (i.e., creation order doesn't matter) if they don't pass `SS_ALLSAME` and they don't mind paying the synchronization cost later.

See Also:

- *ss_pers_new*: 7.1: Create a new persistent object
- *Persistent Objects*: Introduction for current chapter

Sets persistent object to initial state

`ss_pers_reset` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_reset (ss_pers_t *pers, unsigned flags)
```

Formal Arguments:

- `pers`: The object to be reset
- `flags`: Bit flags such as `SS_ALLSAME`

Description: This function sets a persistent object to an initial state of an all-zero bit pattern. The `ss_foo`*obj*`*_tm` and `ss_foo`*obj*`*_tf` parts of the C struct are set to zero but the other stuff is left unmodified (except the dirty bit is set).

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent. However, if the `SS_ALLSAME` bit flag is set then this function should be called collectively across the communicator of the scope that owns the object.

See Also:

- *Persistent Objects*: Introduction for current chapter

Destructor

`ss_pers_dest` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_dest (ss_pers_t *pers)
```

Description: Destroys a persistent object link by releasing those resources used by a persistent object link. If the link was allocated on the heap then the caller should free it.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Dereference an object link

`ss_pers_deref` is a function defined in `sspers.c`.

Synopsis:

```
ss_persobj_t * ss_pers_deref (ss_pers_t *pers)
```

Description: Given a link to a persistent object, dereference that link and return a pointer to the object itself. This may involve reading a table into memory if this is the first dereference into that table.

This function is almost never invoked directly by client code. Instead, the client will use macros appropriate for each object class which will check the link class and cast the return value to the appropriate type. For instance, `SS_FIELD` is a macro that takes a field object link as an argument, compile-time checks that the argument is an `ss_field_t` pointer, run-time check that the pointer is valid, and return an `ss_fieldobj_t` pointer.

Return Value: Returns an object pointer on success; the null pointer on failure.

Parallel Notes: Independent

Issues: We should probably accumulate some sort of statistics here to make sure that the object caching is performing as expected.

See Also:

- *Persistent Objects*: Introduction for current chapter

Updates an object link

`ss_pers_update` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_update (ss_pers_t *pers)
```

Description: This function makes information in the object link as current as possible. If the object has a permanent home in the table but the object index stored in the link is indirect then it is converted to a direct index. If the object is in memory then the link is moved to the ‘memory’ state. If the ‘mapidx’ value stored in the object is different than the object index in the link then the object index is updated in the link.

This function is essentially a weaker version of *ss_pers_deref*.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *ss_pers_deref*: 7.5: Dereference an object link
- *Persistent Objects*: Introduction for current chapter

Create an object link

`ss_pers_refer` is a function defined in `sspers.c`.

Synopsis:

```
ss_pers_t * ss_pers_refer (ss_scope_t *scope, ss_persobj_t *persobj, ss_pers_t *pers)
```

Formal Arguments:

- `scope`: The scope to which `persobj` belongs.

- `persobj`: The object to which the new link will point.
- `pers`: Optional memory for the link.

Description: This function creates (or fills in) a link to a new object that exists in memory.

Return Value: Returns a pointer to a persistent object link (either the supplied `pers` or a newly allocated one) on success; the null pointer on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Obtain scope for an object

`ss_pers_scope` is a function defined in `sspers.c`.

Synopsis:

```
ss_scope_t * ss_pers_scope (ss_pers_t *pers, ss_scope_t *buf)
```

Formal Arguments:

- `pers`: Persistent object link to query
- `buf` [OUT]: Optional buffer for the result scope link

Description: Every persistent object belongs to a scope and this function returns a link to that scope.

Return Value: On success, returns a link to the scope containing `pers`. If the caller supplied a buffer for the result in the `buf` argument then that's the success pointer returned. Returns the null pointer on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Obtain file for an object

`ss_pers_file` is a function defined in `sspers.c`.

Synopsis:

```
ss_file_t * ss_pers_file (ss_pers_t *pers, ss_file_t *file)
```

Formal Arguments:

- `pers`: Persistent object link to query
- `file` [OUT]: Optional buffer for the result file link

Description: Every persistent object belongs to a file and this function returns a link to that file.

Return Value: On success, returns a link to the file containing `pers`. If the caller supplied a buffer for the result in the `file` argument then that's the success pointer returned. Returns the null pointer on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Test whether an object can be modified

`ss_pers_iswritable` is a function defined in `sspers.c`.

Synopsis:

```
htri_t ss_pers_iswritable (ss_pers_t *pers)
```

Description: An object can be modified if it belongs to a scope that is modifiable. This function tests whether that is true.

Return Value: Returns true (positive) if `pers` points to a persistent object that exists in a writable scope and false otherwise. If `pers` points to a persistent object that is not in memory (e.g., it's file is not open or its file has been subsequently closed) then this function returns false. Errors are indicated with a negative return value.

Parallel Notes: Independent

Issues: Since the client can always obtain a pointer into memory for the object by dereferencing the link, there is nothing stopping the client from modifying that memory and setting the object's dirty bit and since all of that can be done with straight C code (without SSLib assistance) it is impossible for SSLib to warn about that situation. However, a synchronization should be able to detect and report it.

See Also:

- *Persistent Objects*: Introduction for current chapter

Obtain top scope for an object

`ss_pers_topscope` is a function defined in `sspers.c`.

Synopsis:

```
ss_scope_t * ss_pers_topscope (ss_pers_t *pers, ss_scope_t *buf)
```

Formal Arguments:

- `pers`: Persistent object link to query
- `buf [OUT]`: Optional buffer for the result scope link

Description: Every persistent object belongs to a scope, every scope belongs to a file, and every file has one top-scope. This function returns a link to that top-scope.

Return Value: On success, returns a link to the top-scope for `pers`. If the caller supplied a buffer for the result in the `buf` argument then that's the success pointer returned. Returns the null pointer on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Determine link equality

`ss_pers_eq` is a function defined in `sspers.c`.

Synopsis:

```
htri_t ss_pers_eq (ss_pers_t *pers1, ss_pers_t *pers2)
```

Description: This function determines if two links point to the same object.

Return Value: Returns true (positive) if PERS1 and PERS2 refer to the same object without actually dereferencing the link, false if not, and negative on error. The names `ss_pers_eq` and *ss_pers_equal* come from LISP where the `eq` function tests whether its operands refer to the same object and `equal` that recursively compares its operands to determine if they have the same value.

Parallel Notes: Independent

See Also:

- *ss_pers_equal*: 7.13: *Determine object equality*
- *Persistent Objects*: Introduction for current chapter

Determine object equality

`ss_pers_equal` is a function defined in `sspers.c`.

Synopsis:

```
htri_t ss_pers_equal (ss_pers_t *pers1, ss_pers_t *pers2, ss_prop_t UNUSED *props)
```

Formal Arguments:

- `props`: A property list to indicate how the comparison should proceed. No properties are currently defined. If an object contains persistent object links then the pointed-to objects will be compared with *ss_pers_eq* instead of recursively calling `ss_pers_equal`.

Description: This function determines if two links point to objects that could be considered to be the same object even if they don't point to the same memory. In other words, a "meter" in one database is almost certainly the same to a "meter" in some other database even though the two object handles point to distinct objects in memory (i.e., `ss_pers_equal` is true but *ss_pers_eq* is false).

Return Value: Returns true (positive) if PERS1 and PERS2 refer to objects whose internals are equal or if PERS1 and PERS2 both point to the same object or both are null pointers.

Parallel Notes: Independent

See Also:

- *ss_pers_eq*: 7.12: *Determine link equality*
- *Persistent Objects*: Introduction for current chapter

Change link state

`ss_pers_state` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_state (ss_pers_t *pers, ss_pers_link_state_t state)
```

Formal Arguments:

- `pers`: The persistent object link whose state is to be changed.
- `state`: Desired state for the link, one of the `SS_PERS_LINK` constants.

Description: A persistent object link can be in either a Closed or Memory state (not including the Filed state of links as they appear in a file). This function moves a link from state to state and also makes sure all the information in the link is up to date.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Compares two persistent objects

`ss_pers_cmp` is a function defined in `sspers.c`.

Synopsis:

```
int ss_pers_cmp (ss_pers_t *p1, ss_pers_t *p2, const ss_persobj_t *mask)
```

Formal Arguments:

- `p1`: First of two objects to compare.
- `p2`: Second of two objects to compare.
- `mask`: Optional mask to use for deep comparisons. A null value implies a shallow comparison, which means that the comparison should only look at the object handles and not the objects themselves.

Description: Compares two objects, `P1` and `P2`, and returns a value similar to `memcmp` or `strcmp`.

If `mask` is the null pointer then only the contents of the links `P1` and `P2` are consulted and the underlying objects are never referenced, a so called *shallow* comparison. The return value will be one of the following:

```
1 | 1 if A > B by some well defined global ordering
2 | 0 if A and B point to the same object
3 | -1 if A < B
4 | -2 on error
```

On the other hand, if `mask` is not null then a *deep* comparison is performed and `mask` should be a block of memory the same size as the objects to which `P1` and `P2` point (`mask` is not referenced if `P1` and `P2` point to objects of different types). The block of memory should be initialized to zero except where a comparison in the two underlying objects is desired. For instance, when comparing the roles and topological dimensions of two categories (`SAF__Cat`) you would do the following:

```
1 | SAF_Cat a=...; b=...;
2 | ss_catobj_t mask;
3 | memset(&mask, 0, sizeof mask);
4 | memset(&mask.role, SS_VAL_CMP_DFLT, 1); // default comparison mode for roles
5 | memset(&mask.tdim, SS_VAL_CMP_DFLT, 1); // default comparison mode for topological_
6 | ss_pers_cmp((ss_pers_t*)&a, (ss_pers_t*)&b, (ss_persobj_t*)&mask);
```

Note two things: (1) only the first non-zero byte in the mask corresponding to any particular member is consulted to determine the kind of comparison, and (2) `ss_pers_t` and `ss_persobj_t` are the types from which all persistent object links and objects are derived and are binary compatible with all links and objects.

To compare all fields of two categories you could just set the whole mask to the desired comparison because `ss_pers_cmp` will skip over parts of the mask that don't actually correspond to things that can be compared (e.g., padding bytes inserted by the compiler between members of the object and parts of the objects that are SSlib's private bookkeeping records):

```
memset (&mask, SS_VAL_CMP_DFLT, sizeof mask);
```

The various flags defining comparisons are defined by the *ss_val_cmp_t* type.

Return Value: Similar to `memcmp` except successful return value is one of: -1, 0, or 1 instead of arbitrary negative and positive values. This allows -2 (or less) to indicate failure, which is standard practice in SSlib for comparison functions.

Parallel Notes: Independent

Issues: Deep comparisons are not yet fully recursive. I.e., if P1 and P2 are being deeply compared and the objects to which P1 and P2 point contain object links which are being compared because they correspond to non-zero bits in `mask`, then only a shallow comparison is performed on those links. We plan to add a property list argument to this function that would allow finer-grained control of the deep comparison recursion.

See Also:

- *SS_PERS_EQ: 7.27: Determine link equality*
- *SS_PERS_EQUAL: 7.28: Determine object equality*
- *ss_pers_eq: 7.12: Determine link equality*
- *ss_pers_equal: 7.13: Determine object equality*
- *Persistent Objects: Introduction for current chapter*

Compares two persistent objects

`ss_pers_cmp_` is a function defined in `sspers.c`.

Synopsis:

```
int ss_pers_cmp_ (ss_persobj_t *p1, ss_persobj_t *p2, const ss_persobj_t *mask)
```

Formal Arguments:

- `p1`: First of two objects to compare. This is normally considered to be the “haystack”.
- `p2`: Second of two objects to compare. This is normally considered to be the “needle” and might contain special things like regular expressions, etc. depending on the values contained in the `mask`.
- `mask`: Which elements of P1 and P2 to compare. This isn’t really a true object but rather a chunk of memory the same size as the objects that is filled with bytes that say which members of P1 and P2 to compare and how to compare them.

Description: This is an internal version of *ss_pers_cmp* and does only a deep comparison of the two objects.

Return Value: On success returns -1, 0, or 1 depending on whether P1 is less than, equal, or greater than P2 by some arbitrary but consistent comparison algorithm. Returns -2 on failure.

Parallel Notes: Independent

See Also:

- *ss_pers_cmp: 7.15: Compares two persistent objects*
- *Persistent Objects: Introduction for current chapter*

Compute a checksum for a persistent object

`ss_pers_cksum` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_cksum(ss_persobj_t *persobj, ss_val_cksum_t *cksum)
```

Formal Arguments:

- `persobj`: Persistent object whose checksum will be computed.
- `cksum` [OUT]: The computed checksum.

Description: Computes a checksum for the persistent part of a persistent object in memory.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- [Persistent Objects](#): Introduction for current chapter

Find objects in a scope

`ss_pers_find` is a function defined in `sspers.c`.

Synopsis:

```
ss_pers_t * ss_pers_find(ss_scope_t *scope, ss_pers_t *key, ss_persobj_t *mask, size_t nskip,  
size_t *nfound, ss_pers_t *buffer, ss_prop_t *props)
```

Formal Arguments:

- `scope`: Scope to be searched
- `key`: Value for which to search. This is required even if `mask` is null because the `key` determines the type of objects for which to search.
- `mask`: Which elements of `key` to consider when searching. It is an error if no bits of `mask` are set, but if `mask` is the null pointer then `key` is assumed to match every object. If non-null then `mask` and `key` must be of the same type. The reason `mask` is an object pointer rather than an object link is that the memory is really only used to store one-byte flags that control how the matching is performed. In other words, `mask` isn't truly an object—it just has to be the same size as an object.
- `nskip`: Number of initial matched results that should be skipped.
- `nfound` [INOUT]: The input value limits the matching to the specified number of
- `buffer`: Optional buffer to fill in with handles to items that were found. If this is the constant `SS_PERS_TEST` then this function behaves exactly as if the caller had supplied a buffer but does not attempt to return links to the matching objects.
- `props`: Optional properties (See **Persistent Object Properties**)

Description: This function will find all objects in a particular `scope` that match certain fields of a specified `key` object. The `key` and `mask` must refer to persistent objects of the same type where `key` contains the values to compare against and `mask` specifies which part of `key` to consider and how to compare. However, the `mask` is not a true object in that it doesn't need to be created in some table with `ss_pers_new`; it can just be allocated on the stack. Any atomic element of `mask` that has at least one bit set indicates that the corresponding element of `key` is to be considered during the comparison. If no bits of `mask` are set then an error is raised, but if `mask` is the null pointer then we treat the `key` as matching every object in the scope.

If `nfound` is non-null then its incoming value will be used to limit the search to the specified number of returned matches. If more items match than what was specified then the additional items are simply ignored as if they didn't even exist (unless the “`detect_overflow`” property is true, in which case an error is raised). The caller can pass in `SS_NOSIZE` if no limit is desired. If `nfound` is the null pointer (it can only be so if `buffer` is also null) then it is treated as if it had pointed to `SS_NOSIZE`.

The caller can supply a `buffer` for the result or, by passing a null pointer, request that the library allocate the buffer. If `buffer` is supplied then it must contain at least `nfound` (as set on entry to this function) elements to hold the result. But if `buffer` is the special constant `SS_PERS_TEST` then the function behaves as if a valid `buffer` was supplied except that it does not attempt to initialize that buffer in any way. This can be used to count how many matches would be found and even limit the counting by supplying an initial value for `nfound`.

A positive value for an `nskip` argument causes this function to act as if the first `nskip` matched objects didn't, in fact, match.

Return Value: On success this function returns an array of matching persistent object links into the specified `scope` (the caller supplied `buffer` or one allocated by the library) or the constant `SS_PERS_TEST` and `nfound` (if supplied) will point to the number of matches found limited by the incoming value of `nfound` (or `SS_NOSIZE`). If space permits, the last element of the return value will be followed by a null persistent link, which makes it possible to loop over the return value even if `nfound` was the null pointer.

In order to distinguish the case where no item is found from the case where an error occurs, the former results in a non-null return value (the library will allocate an array of size one if the caller didn't supply a `buffer` and initialize it to `SS_PERS_NULL`). The `nfound` returned value is zero in either case.

If no objects match in the specified scope and the object type is not `ss_scope_t` or `ss_file_t` and the *noregistry*'s property is false or not set then each registry scope associated with the file containing “`scope`” will be searched until matches are found in some scope or all registries are processed.

This function returns the null pointer for failure. It is not considered a failure when the `key` simply doesn't match any of the available objects.

Parallel Notes: Independent

Example: Example 1: Find all fields with an association ratio of 1 in the *main* scope:

```

1 // Obtain key from a transient scope; allocate the mask on the stack
2 ss_field_t *key = SS_PERS_NEW(transient, ss_field_t, SS_ALLSAME);
3 ss_fieldobj_t mask;
4 // Set key value for which to search and indicate such in the mask
5 SS_FIELD(key)->assoc_ratio = 1;
6 memset(&mask, 0, sizeof mask);
7 mask.assoc_ratio = SS_VAL_CMP_DFLT; //default comparison
8 // Search for matches
9 size_t nfound = SS_NOSIZE;
10 ss_field_t *found = ss_pers_find(main, key, mask, 0, &nfound, NULL, NULL);
11 // Print names of all matches
12 for (i=0; i<nfound; i++)
13     printf("match %d name=\"%s\"\n", i, ss_string_ptr(SS_FIELD_P(found+i, name)));

```

Example 2: Find first 10 fields with a name consisting of the word “field” in any combination of upper and lower case letters, followed by one or more digits:

```

1 // Obtain key from a transient scope; allocate the mask on the stack
2 ss_field_t *key = SS_PERS_NEW(transient, ss_field_t, SS_ALLSAME);
3 ss_fieldobj_t mask;
4 // Set key value for which to search and indicate such in the mask
5 ss_string_set(SS_FIELD_P(key, name), "^field[0-9]+$");
6 memset(&mask, 0, sizeof mask);

```

(continues on next page)

(continued from previous page)

```

7  memset(&(mask.name), SS_VAL_CMP_RE_ICASE, 1);
8  // Search for matches
9  size_t nfound = 10;
10 ss_field_t found[10];
11 ss_pers_find(main, key, &mask, 0, &nfound, found, NULL);

```

Example 3: Count how many fields are in the scope *procl*:

```

1  ss_field_t *key = ....; // any field object
2  size_t nfound = SS_NOSIZE; // do not limit the search
3  ss_pers_find(procl, key, NULL, 0, &nfound, SS_PERS_TEST, NULL);
4  printf("found %lu item(s)\n", (unsigned long)nfound);

```

See Also:

- *ss_pers_new*: 7.1: Create a new persistent object
- *Persistent Objects*: Introduction for current chapter

Mark object as modified

`ss_pers_modified` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_modified(ss_pers_t *pers, unsigned flags)
```

Formal Arguments:

- `pers`: Persistent object to mark as modified
- `flags`: Bitflags such as `SS_ALLSAME`

Description: If a persistent object is modified then it should also be marked as such by invoking this function. If all tasks modify the persistent object in the same manner then the second argument can be `SS_ALLSAME`, otherwise it should be zero. The `SS_PERS_MODIFIED` macro is a convenience for this function since the client is often passing a subclass of `ss_pers_t` and may get compiler warnings.

The client can call this function either before or after making a modification to the object, but it's generally safer to make this call first so that the object is marked as modified even if the modification is interrupted by an error. It doesn't hurt to mark an object as modified and then not actually modify it – it just causes the synchronization algorithm to take longer to discover that there weren't any changes.

The 'dirty' flag is always set to true to indicate that the object's new value differs (or is about to differ) from what is stored in the file.

If `flags` has the `SS_ALLSAME` bit set then the client is indicating that all tasks belonging to the scope have (or will) make identical modifications. In this case, if the object's `sync'd` flag is set we promote it to :ref:'SS_ALLSAME <SS>' to indicate that the object is synchronized but its `sync_cksum` and `sync_serial` values are outdated. Otherwise the object's 'sync'd' flag is set to false.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

Issues: We should really check whether the scope owning the object is read-only, otherwise we won't get any indication of an error until we try to synchronize.

See Also:

- `SS_PERS_MODIFIED`: 7.29: Mark object as modified

- *Persistent Objects*: Introduction for current chapter

Make a new object unique

`ss_pers_unique` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_unique (ss_pers_t *pers)
```

Formal Arguments:

- `pers`: Persistent object to make unique

Description: If N MPI tasks each create an object that is identical across all the tasks (such as happens when `saf_declare_set` is called with all arguments the same) then SSlib will merge those N new objects into a single permanent object. A similar thing happens when a single task creates multiple new identical objects. The merging happens during a synchronization operation and only for objects that are new (i.e., objects that were not created with the `SS_ALLSAME` bit flag).

By calling this function on a persistent object, the persistent object is modified in a unique manner which causes it to be different than any other object on this MPI task or any other MPI task. The uniqueness should only be set for new objects.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Debugging aid

`ss_pers_debug` is a function defined in `sspers.c`.

Synopsis:

```
herr_t ss_pers_debug (ss_pers_t *pers)
```

Description: Prints all known information about an object to the standard output stream.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Decode persistent object links

`ss_pers_decode_cb` is a function defined in `sspers.c`.

Synopsis:

```
size_t ss_pers_decode_cb (void *buffer, const char *serbuf, size_t size, size_t nelmts)
```

Formal Arguments:

- `buffer`: Array of objects into which to decode `serbuf`.

- `serbuf`: Encoded information to be decoded.
- `size`: Size of each element in buffer array.
- `nelmts`: Number of elements in buffer array.

Description: Decodes the stuff encoded by `ss_pers_encode_cb`.

Return Value: Returns total number of bytes consumed from `serbuf` on success; `SS_NOSIZE` on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Destructor

`SS_PERS_DEST` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_DEST` (`_pers_`)

Description: This is simply a convenience function for *`ss_pers_dest`* so that the caller doesn't have to cast the argument and return value. The underlying object is not destroyed—only the link to that object.

Return Value: Always returns null so it can be easily assigned to the object link being destroyed.

Example: .. `_SC_SS_PERS_DEST`:

```
1 field = SS_OBJ_DEST(field);
```

See Also:

- *`ss_pers_dest`*: 7.4: *Destructor*
- *Persistent Objects*: Introduction for current chapter

Constructor

`SS_PERS_NEW` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_NEW` (`_scope_`, `_type_`, `_flags_`)

Description: This is simply a convenience function for *`ss_pers_new`* so that the caller doesn't have to cast the argument and return value. The `_type_` should be one of the persistent object link datatypes like `ss_field_t`, and not one of the persistent object types like `ss_fieldobj_t`.

Return Value: Returns a pointer to a link to the new object on success; null on failure.

Example: .. `_SC_SS_PERS_NEW`:

```
1 To create a new Field object:  
2 field = SS_PERS_NEW(scope, ss_field_t, SS_ALLSAME);
```

See Also:

- *`ss_pers_new`*: 7.1: *Create a new persistent object*
- *Persistent Objects*: Introduction for current chapter

Copy constructor

`SS_PERS_COPY` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_COPY` (`_old_`, `_scope_`, `_flags_`)

Description: This is simply a convenience function for *`ss_pers_copy`* so that the caller doesn't have to cast the arguments and return value.

Return Value: Returns a pointer to a link to the new object on success; null on failure.

See Also:

- *`ss_pers_copy`*: 7.2: *Copy an object*
- *Persistent Objects*: Introduction for current chapter

Find objects in a scope

`SS_PERS_FIND` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_FIND` (`scope`, `key`, `mask`, `limit`, `nfound`)

Description: This is simply a convenience function for *`ss_pers_find`* so that the caller doesn't have to cast the arguments to `ss_pers_t` pointers (they'll still be run-time type checked).

Return Value: See *`ss_pers_find`*

Parallel Notes: See *`ss_pers_find`*

See Also:

- *`ss_pers_find`*: 7.18: *Find objects in a scope*
- *Persistent Objects*: Introduction for current chapter

Determine link equality

`SS_PERS_EQ` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_EQ` (`link1`, `link2`)

Description: This macro returns true if two links point to the same object. Arguments are cast appropriately.

Return Value: True (positive) if same object, false if different, negative on error.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Determine object equality

`SS_PERS_EQUAL` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_EQUAL` (link1, link2)

Description: This macro returns true if two links point to equivalent objects. Arguments are cast appropriately.

Return Value: True (positive) if objects are equal, false if not equal, negative on error.

Parallel Notes: Independent.

See Also:

- *Persistent Objects*: Introduction for current chapter

Mark object as modified

`SS_PERS_MODIFIED` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_MODIFIED` (_pers_, _flags_)

Description: This macro is simply a wrapper around *`ss_pers_modified`* so the caller doesn't have to cast arguments.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *`ss_pers_modified`*: 7.19: *Mark object as modified*
- *Persistent Objects*: Introduction for current chapter

Check if link is null

`SS_PERS_ISNULL` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_ISNULL` (_pers_)

Description: A null persistent object link is indicated by the link being in the `SS_PERS_LINK_NULL` state.

Return Value: Returns true if the specified link a null pointer or is in the null state; false otherwise.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Make an object unique

`SS_PERS_UNIQUE` is a macro defined in `sspers.h`.

Synopsis:

`SS_PERS_UNIQUE` (_pers_)

Description: Makes an object unique by giving it a unique serial number. The number is unique across all the MPI tasks so that when N tasks create N identical new objects that only differ in serial number, SSlib will convert them to N identical permanent objects instead of merging them all into a single permanent object.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Persistent Objects*: Introduction for current chapter

Persistent Object Tables

No description available.

Members

Find indirect indices for an object

`ss_table_indirect` is a function defined in `sstable.c`.

Synopsis:

`size_t ss_table_indirect (ss_table_t *table, size_t idx, size_t beyond)`

Formal Arguments:

- `table`: Table in which `idx` exists
- `idx`: Index for the object for which we are searching for an indirect index. This is usually a direct index but doesn't necessarily have to be such.
- `beyond`: Return an indirect index greater than this value. A value of zero means to return the first indirect index. This argument can be used to scan through a table looking for all matching indirect indices for a particular direct index. If this is a direct index (such as zero) then the first matching indirect index is returned.

Description: Given a direct index for a table object, return the first indirect index which also points to the object and which is larger than `beyond`.

Return Value: Returns a matching indirect index on success; `SS_NOSIZE` on failure.

Parallel Notes: Independent

See Also:

- *Persistent Object Tables*: Introduction for current chapter

Strings

Variable length strings as stored in persistent objects are manipulated through the SSlib `string` interface and use a datatype `ss_string_t` which is opaque to the client. This allows the implementation of persistent object strings to be changed as necessary to keep pace with functionality and performance improvements in the HDF5 string datatype.

As it turns out, HDF5 is unable to output variable length strings in parallel (1.7.3 2003-09-12). Therefore it has become necessary to change the implementation in SSlib already: all character strings for all objects of a particular scope will be stored in an extendible "Strings" dataset of type `H5T_NATIVE_UCHAR` in the same scope. Any object that contains a variable length string will contain an index into the "Strings" dataset, and when the object is in memory it will also contain a pointer directly to the string value. We employ an opaque HDF5 datatype to represent the string

in memory and register a conversion function to allocate/find the string in the “strings” dataset during I/O. The only problem with this approach is that HDF5-level tools don’t understand that the offset is an index into the Strings dataset for a character string.

When a new task is opened all strings will initially have the same contents for the variable length string buffer, which is read by `ss_string_boot`. As execution progresses different tasks will add different strings to the buffer in different orders and the tasks will become out of sync. When objects of a scope are synchronized we will be guaranteed that all tasks contain a valid Strings buffer, although the order of the new values in the buffer may differ between tasks. The `ss_string_flush` function is responsible for choosing one of the scope tasks to write the string data back to the file.

SSlib variable length strings support uses length rather than NUL characters to mark the end of a string and are therefore capable of storing strings of bytes that might have embedded NUL characters.

Members

Get a C string from a persistent string

`ss_string_get` is a function defined in `ssstring.c`.

Synopsis:

```
char * ss_string_get (const ss_string_t *str, size_t bufsize, char *buf)
```

Formal Arguments:

- `bufsize`: Size of `buf` (only referenced if `buf` is non-null).
- `buf`: Optional buffer in which to store the C string. This buffer is assumed to be an array of at least `bufsize` characters.

Description: Given information about a persistent string, return a pointer to a C string, i.e., an array of NUL-terminated characters. If the caller supplies a buffer then the string will be copied into that buffer and NUL terminated, otherwise this function mallocs a new buffer to hold the result.

Return Value: On success, returns `buf` if non-null or else allocates a result buffer. On failure returns the null pointer. It is a failure to supply a `bufsize` which is not large enough to hold the entire string value with its NUL terminator. The caller is responsible for freeing any return value allocated by this function.

Parallel Notes: Independent

Issues: SSlib stores strings with a byte count, so if the string was stored without a terminating NUL character then it will also be returned as such.

See Also:

- [Strings](#): Introduction for current chapter

Obtain pointer into string object

`ss_string_ptr` is a function defined in `ssstring.c`.

Synopsis:

```
const char * ss_string_ptr (const ss_string_t *str)
```

Description: This function is similar to [ss_string_get](#) except rather than copying the string to some other memory it returns a pointer directly into the `ss_string_t` object. The caller should expect that the pointer is valid only until some other operation on that object.

Return Value: On success, returns a temporary pointer directly into the `str` object. If the value is zero bytes long then a pointer to a NUL character is returned instead of null.

Parallel Notes: Independent

Issues: The returned value is ‘const’ because if it points into the Strings array then it might be the case that multiple strings currently having the same value are sharing the same storage.

See Also:

- [*ss_string_get*](#): 9.1: *Get a C string from a persistent string*
- [*Strings*](#): Introduction for current chapter

Store a C string in a persistent string

`ss_string_set` is a function defined in `ssstring.c`.

Synopsis:

```
herr_t ss_string_set (ss_string_t *str, const char *s)
```

Formal Arguments:

- `str`: The destination persistent string.
- `s`: The source C string to copy.

Description: Given a C-style NUL-terminated character string stored in `s`, make the persistent string object `str` have that same value. The [*ss_string_memset*](#) function can be used to store arbitrary data that might have embedded NUL characters or that might not be NUL-terminated.

Return Value: Returns non-negative on success and negative on failure. The success side effect is that `str` has the same value as `s`.

Parallel Notes: Independent

See Also:

- [*ss_string_memset*](#): 9.4: *Store a byte array in a string*
- [*Strings*](#): Introduction for current chapter

Store a byte array in a string

`ss_string_memset` is a function defined in `ssstring.c`.

Synopsis:

```
herr_t ss_string_memset (ss_string_t *str, const void *value, size_t nbytes)
```

Formal Arguments:

- `str`: The destination variable length string.
- `value`: The optional value to assign to `str`. If `NULL` then a value of all zero bytes is used.
- `nbytes`: The number of bytes in `value`.

Description: This function is similar to [*ss_string_set*](#) except the number of bytes in `value` is explicitly passed instead of looking for the first NUL character.

Return Value: Returns non-negative on success and negative on failure.

Parallel Notes: Independent

See Also:

- *ss_string_set*: 9.3: *Store a C string in a persistent string*
- *Strings*: Introduction for current chapter

Free memory associated with the string

`ss_string_reset` is a function defined in `ssstring.c`.

Synopsis:

```
herr_t ss_string_reset (ss_string_t *str)
```

Description: Frees the character array value stored in `str` but does not free `str` itself.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Weakly reset a string

`ss_string_realloc` is a function defined in `ssstring.c`.

Synopsis:

```
herr_t ss_string_realloc (ss_string_t *str)
```

Description: Sometimes we want a variable length string to keep the same value it had but to be reallocated in the string backing store. For instance, when a variable length string is copied from one scope to another we want to keep the same value but it cannot continue to have the same dataset offset since the new scope has a completely different dataset for string storage.

Return Value: Returns non-negative on success; negative on failure

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Compares two variable length strings

`ss_string_cmp` is a function defined in `ssstring.c`.

Synopsis:

```
int ss_string_cmp (const ss_string_t *s1, const ss_string_t *s2)
```

Description: This function is similar to `memcmp` but its arguments are variable length strings instead.

Return Value: Returns -1 if the value of `S1` is less than `S2`, 1 if `S1` is greater than `S2`, and zero if they are equal in value. Returns -2 on failure (beware that this is a refinement of the more general negative returns on failure used throughout SSlib). It is an error if `S1` or `S2` is a null pointer, but not if either or both have no associated value. Lack of a value is less than any other value and if `S1` and `S2` both lack a value they are considered equal.

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Compare persistent string with C string

`ss_string_cmp_s` is a function defined in `ssstring.c`.

Synopsis:

```
int ss_string_cmp_s (const ss_string_t *str, const char *s)
```

Description: This function is similar to `strcmp` but its first argument is a persistent string instead. It compares the value of the persistent string with the C string `s`.

Return Value: Returns -1 if the value of `str` is less than `s`, 1 if `str` is greater than `s`, and zero if they are equal in value. Returns -2 on failure (beware that this is a refinement of the more general negative returns on failure used throughout SSlib).

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Append one string to another

`ss_string_cat` is a function defined in `ssstring.c`.

Synopsis:

```
herr_t ss_string_cat (ss_string_t *str, const char *s)
```

Description: Changes the value of the persistent string by appending another string.

Return Value: Returns non-negative on success; negative on failure. Successful side effect is that the value of `str` is modified by appending the string `s`, which should be a C NUL-terminated string. If the original value of `str` is NUL-terminated then the additional `s` value will replace that NUL, otherwise the additional value will be added after the existing value.

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Substring modification

`ss_string_splice` is a function defined in `ssstring.c`.

Synopsis:

```
herr_t ss_string_splice (ss_string_t *str, const char *value, size_t start, size_t nbytes, size_t nreplace)
```

Formal Arguments:

- `str`: String object to be modified by this operation.

- **value:** Optional new data for part of the string value. If this argument is the null pointer and **nbytes** is positive then the new data will be all NUL characters (this allows for an easy way to extend the length of a string).
- **start:** Byte offset at which to place the new data in the string.
- **nbytes:** Length of the new data in bytes.
- **nreplace:** Number of bytes replaced by the new data. If **SS_NOSIZE** is passed then all bytes from **start** to the end of the original value will be replaced by the new value.

Description: This function is able to modify a string value by modifying, inserting, or deleting bytes. It does not assume that **value** is a C-style NUL-terminated string.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Query the length of a persistent string

`ss_string_len` is a function defined in `ssstring.c`.

Synopsis:

`size_t ss_string_len (const ss_string_t *str)`

Description: This function returns the number of initial non-NUL characters in a string's value. If **str**'s value is a typical NUL-terminated C-style string then this function's return value is identical to `strlen`. However, if **str**'s value has no NUL characters then this function's return value is identical to *`ss_string_memlen`*.

Return Value: Returns the number of initial non-NUL characters in a string on success; returns **SS_NOSIZE** on failure.

Parallel Notes: Independent

See Also:

- *`ss_string_memlen`*: 9.12: *Query the length of a persistent string*
- *Strings*: Introduction for current chapter

Query the length of a persistent string

`ss_string_memlen` is a function defined in `ssstring.c`.

Synopsis:

`size_t ss_string_memlen (const ss_string_t *str)`

Description: Given a persistent string, return the number of characters contained in that string, including the terminating NUL character if any. Note that for NUL terminated strings this is one more than what `strlen` would have returned, but this behavior is necessary since SSlib byte-counts all string data in order to allow strings that have embedded NUL characters and that might lack a NUL terminator. In other words, SSlib strings can store any type of data.

Return Value: On success, returns the actual number of bytes stored for the string including a NUL terminator if any. Returns **SS_NOSIZE** on error.

Parallel Notes: Independent

See Also:

- *Strings*: Introduction for current chapter

Variable Length Arrays

SSlib has support for small, variable-length, arrays with independent operations. The array elements can be either persistent object links or a user-defined datatype without embedded SSlib special types. The data for all variable length arrays in a scope is aggregated (together with the variable length strings) into a single dataset which is read in its entirety when a scope is opened.

The reason for the distinction between whether the array stores SSlib datatypes or not is because conversion between memory and file representations, calculation of checksums, and interprocess communication require facilities provided by SSlib for those types and aren't available (or are handled entirely differently) for the non-SSlib datatypes. In practice this doesn't turn out to be a problem because variable length arrays are generally only used to store persistent object links (as in a Set object pointing to Collections) or native integers (as in a Field's permutation vector).

An array is born with zero elements of of `ss_pers_t` type. If the array is intended to store something other than object links then its datatype must be changed with `ss_array_target`. The number of elements in an array is changed with `ss_array_resize`. The `ss_array_get` and `ss_array_put` functions query or modify elements of an array and `ss_array_reset` sets the array back to an initial state.

Members

Change array element datatype

`ss_array_target` is a function defined in `ssarray.c`.

Synopsis:

`herr_t ss_array_target (ss_array_t *array, hid_t ftype)`

Formal Arguments:

- `array`: Array whose datatype is to be modified.
- `ftype`: Datatype of the array elements as stored in the file.

Description: Every array has two datatypes associated with it: a datatype for the elements as they are stored in the file, and a datatype of the elements as they exist in memory with the `ss_array_put` and `ss_array_get` functions. When a new array is created the memory datatype is the HDF5 equivalent of `ss_pers_t` (a persistent object link) and the file datatype is its counterpart, `ss_pers_tf`. This function sets the file datatype to the specified value and clears the memory buffer and associated memory datatype.

Return Value: Returns non-negative on success; negative on failure. It is an error to modify the file datatype if the array size is positive.

Parallel Notes: Independent

See Also:

- `ss_array_get`: 10.4: *Obtain array value*
- `ss_array_put`: 10.5: *Modify part of an array*
- *Variable Length Arrays*: Introduction for current chapter

Inquire about array file datatype

`ss_array_targeted` is a function defined in `ssarray.c`.

Synopsis:

```
hid_t ss_array_targeted (ss_array_t *array)
```

Description: Every array has a file datatype that determines how values are stored in a file. This function returns a copy of that datatype.

Return Value: On success, a positive object ID for a copy of the file datatype. If an array stores links to other objects then the returned datatype is a copy of `ss_pers_tf`. Returns negative on failure.

Parallel Notes: Independent

See Also:

- *Variable Length Arrays*: Introduction for current chapter

Change the size of a variable length array

`ss_array_resize` is a function defined in `ssarray.c`.

Synopsis:

```
herr_t ss_array_resize (ss_array_t *array, size_t nelmts)
```

Formal Arguments:

- `array`: Array whose size is to be changed.
- `nelmts`: Number of total elements to be contained in the array.

Description: Elements can be added or removed from the end of an array. If items are added then they are also initialized to zero.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent. If more than one task changes the size of an array then they must all make identical changes to the size.

See Also:

- *Variable Length Arrays*: Introduction for current chapter

Obtain array value

`ss_array_get` is a function defined in `ssarray.c`.

Synopsis:

```
void * ss_array_get (ss_array_t *array, hid_t mtype, size_t offset, size_t nelmts, void *buffer)
```

Formal Arguments:

- `array`: Array from which to retrieve data.
- `mtype`: Datatype for memory. Pass `ss_pers_tm` (or preferably negative) for an array of persistent object links.
- `offset`: First element to be returned. It is an error to specify a starting element that is outside the valid range of values defined for the array.

- `nelmts`: Number of elements to return. The `offset` and `nelmts` define a range of elements to be returned. If the range extends beyond the end of the defined range of elements for `array` then an error is raised; but if `nelmts` is the constant `SS_NOSIZE` then all elements up to and including the last element are returned.
- `buffer`: The optional caller-supplied buffer to be filled in by the request. If the caller didn't supply a buffer then one will be created.

Description: This function extracts the array value or subpart thereof. The value is copied into the optional caller-supplied `buffer` (or a buffer is allocated). If the value consists of more than one element then desired elements to be returned can be specified with an offset and length.

Return Value: Returns a pointer (`buffer` if non-null) on success; null on failure. If the caller doesn't supply `buffer` then this function will allocate one.

Parallel Notes: Independent

See Also:

- [Variable Length Arrays](#): Introduction for current chapter

Modify part of an array

`ss_array_put` is a function defined in `ssarray.c`.

Synopsis:

```
herr_t ss_array_put (ss_array_t *array, hid_t mtype, size_t offset, size_t nelmts, const void *value)
```

Formal Arguments:

- `array`: The array whose value will be modified.
- `mtype`: The datatype of the values pointed to by `value`. If the array contains persistent object links then pass `ss_pers_tm` (or preferably negative).
- `offset`: The array element number at which to put `value`.
- `nelmts`: The number of array elements in `value`. If this is the constant `SS_NOSIZE` then we assume that `value` contains enough data to fill up the current size of the array beginning at the specified `offset`.
- `value`: The value to be written into the array.

Description: `nelmts` values beginning at array element `offset` are modified by setting them to `value`.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

Issues: there really isn't any point in actually converting the existing values to memory format and initializing the array's mbuf if we're about to overwrite the whole thing anyway.

See Also:

- [Variable Length Arrays](#): Introduction for current chapter

Free memory associated with the array

`ss_array_reset` is a function defined in `ssarray.c`.

Synopsis:

```
herr_t ss_array_reset (ss_array_t *array)
```

Description: Frees the array value stored in `array` but does not free `array` itself.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- [Variable Length Arrays](#): Introduction for current chapter

Query the number of elements

`ss_array_nelmts` is a function defined in `ssarray.c`.

Synopsis:

`size_t ss_array_nelmts (const ss_array_t *array)`

Description: This function returns the number of elements defined in `array`.

Return Value: Number of elements on success; `SS_NOSIZE` on failure.

Parallel Notes: Independent

See Also:

- [Variable Length Arrays](#): Introduction for current chapter

Files

An SSlib file is an HDF5 file with a certain minimum internal structure: all SSlib files contain a group named “/SAF” which serves as the top-level scope for the file (there may be additional groups that also implement other scopes). The top-level scope is always opened with the file’s communicator and has a *Scope* table in addition to the usual tables. The *Scope* table is a list of all scopes contained in the file and the first element of that table is the top-level scope itself.

Every scope also has a *File* table that lists all files referred to by objects stored in that scope. The first element of every *File* table is understood to be the file containing that table.

Files are always opened with the [ss_file_open](#) function (or the convenience [ss_file_create](#)) and closed with [ss_file_close](#). The file-related functions operate on or return a link of type `ss_file_t`, which points to some entry in a top-scope’s *File* table.

In addition to the collection of *File* tables in all scopes of all files that are currently open, the library maintains a per-task list of files and a mapping from the *File* table members to this global list. The mapping from a *File* table member to the `gfile` array is accomplished with the [ss_file_open](#) function. This allows the following:

- Members from multiple *File* tables can point to a common underlying HDF5 file,
- The file whose name was originally recorded in the *File* table can be temporarily renamed,
- Newly discovered file objects can be implicitly opened if they match a previous open file.

The implicit opening needs more discussion: whenever SSlib opens a file (call it the *master* file) it looks at the *File* tables of the master file to discover the names of all subordinate files that might be referenced by the master file (i.e., all `ss_file_t` objects except the first one in each table, which refers to the master file itself). If any of the subordinate names are relative, it temporarily converts them to absolute names using the master file’s directory as the current working directory and uses these converted names when searching for matching entries in the `GFile` array. If the subordinate file under consideration matches a name in the `GFile` array then the subordinate file will point to that entry, thus sharing the entry with some other file; otherwise the subordinate file will point to a brand new entry. In any case, the subordinate file’s `explicit_open` flag will be set to false and the `GFile` entry’s `cur_opens` counter will not be incremented. If the `GFile` entry is marked as currently open then the subordinate file is also implicitly

open, sharing the same underlying HDF5 file and MPI communicator. Any implicitly opened file can be reopened at any time, either with the same flags as it already shares or with a brand new name, and once that happens the file is considered to be explicitly opened. Care should be taken to ensure that object links weren't already dereferenced through the implicitly opened file, or two or more links that look identical might not be so (this could actually be checked by SSlib with an appropriate counter). Only files that are explicitly opened can be explicitly closed. Any implicitly opened files will implicitly close once all explicitly opened files in common are closed.

Members

Open or create a file

`ss_file_open` is a function defined in `ssfile.c`.

Synopsis:

```
ss_file_t * ss_file_open (ss_file_t *file, const char *name, unsigned flags, ss_prop_t *props)
```

Formal Arguments:

- `file`: Optional handle to a persistent file object from a *File* table.
- `name`: Optional name of file to be opened.
- `flags`: HDF5-style file access flags.
- `props`: Optional file property list (see **File Properties**).

Description: Explicitly opens an SSlib file and returns a link to the `ss_file_t` object for that file. Either `file` or `name` or both may be specified. If one or the other (but not both) is specified then the file is simply opened with the name contained in the file object or the specified name. If both are specified then a mapping from the `file` object to the specified name is established, which is necessary if the name originally recorded in the *Files* table is no longer valid due to the file being moved in the file system.

Depending on `flags`, the file might be created if it doesn't exist (`H5_ACC_CREATE`) or truncated if it does exist (`H5_ACC_TRUNC`). If the file is truncated, files which were mentioned in its *File* table are not automatically deleted or truncated and other files which link to this truncated file will subsequently contain dangling or invalid links.

SSlib supports transient objects by placing them in transient files. A transient file is simply a special SSlib file that doesn't correspond to any underlying storage (not even an HDF5 file using the `core` virtual file driver). Such files support more or less the same SSlib operations as real files although some operations may be tuned for this special case (e.g., `ss_file_flush`). Transient files are always created for read and write access as are the scopes they contain, and are denoted as such by the bit `H5F_ACC_TRANSIENT` in the `flags` argument. They share the same name space as their permanent cousins, and thus it is not possible to have a transient and permanent file both named "foo.saf" although creating a transient file doesn't affect any file that might already exist by that name.

All *File* tables of the file that is newly opened are scanned and all of their members are added to the *GFile* array. If that array already had the name marked as open then the corresponding entry of the *File* table will be implicitly opened. Files opened implicitly need not be closed and can be opened explicitly at any time (although if one is going to open it explicitly it's best to do so early on).

The `ss_file_create` function is a convenience for creating a new file.

Return Value: Returns a link to the `ss_file_t` object for the newly opened file on success; null on failure. If the `file` argument is supplied then this will be the successful return value.

Parallel Notes: Collective across the file's communicator as specified by `props.comm`, defaulting to the same thing as the library's communicator.

Issues: HDF5 doesn't yet (1.6.0) support the `H5F_ACC_TRANSIENT` bit, which would essentially make all operations on the file no-ops. Therefore this functionality must be supported in SSlib.

See Also:

- *ss_file_create*: 11.4: *Create a new file*
- *ss_file_flush*: 11.11: *Write pending data to file*
- *Files*: Introduction for current chapter

Obtain information about referenced files

`ss_file_references` is a function defined in `ssfile.c`.

Synopsis:

```
ss_file_ref_t * ss_file_references (ss_file_t *master, size_t *nfiles, ss_file_ref_t *fileref, ss_prop_t UN-  
USED *props)
```

Formal Arguments:

- `master`: The file in question
- `nfiles` [INOUT]: Upon return this argument will point to the number of valid entries
- `fileref`: Optional pointer to an array of file reference information that will be initialized by this function and returned (if non-null) as the successful return value of this function.
- `props`: File properties (none defined yet)

Description: A SAF database can consist of many files which reference each other. This function will return information about all such files that might be referenced by the `master` file. The caller is expected to fill in certain members of the returned array and then use that array to call *ss_file_openall*.

Return Value: On success, returns either `fileref` (if non-null) or an allocated array and the `nfiles` argument indicates how many elements of the return value have been initialized. Returns the null pointer on failure.

Parallel Notes: Independent.

See Also:

- *ss_file_openall*: 11.3: *Open many files*
- *Files*: Introduction for current chapter

Open many files

`ss_file_openall` is a function defined in `ssfile.c`.

Synopsis:

```
herr_t ss_file_openall (size_t nfiles, ss_file_ref_t *fileref, unsigned flags, ss_prop_t *props)
```

Formal Arguments:

- `nfiles`: Number of entries in the `fileref` array.
- `fileref`: Array of information for files to be opened.
- `flags`: Flags to control how files are opened.
- `props`: Additional file opening properties (see **File Properties**).

Description: This function opens all files specified in the arguments.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the union of communicators specified in the `fileref` array.

See Also:

- [Files](#): Introduction for current chapter

Create a new file

`ss_file_create` is a function defined in `ssfile.c`.

Synopsis:

`ss_file_t * ss_file_create (const char *name, unsigned flags, ss_prop_t *props)`

Formal Arguments:

- `name`: Name of file to be created.
- `flags`: HDF5-style file access flags (see [ss_file_open](#)).
- `props`: Optional file property list (see [*File Properties*](#)).

Description: Creates and initializes an SSlib file. This is actually just a convenience function for calling [ss_file_open](#) with a `flags` argument of `H5F_ACC_RDWR | H5F_ACC_TRUNC | H5F_ACC_CREAT` in addition to those bit flags passed into this function.

Return Value: Same as [ss_file_open](#).

Parallel Notes: Same as [ss_file_open](#).

See Also:

- [ss_file_open](#): 11.1: *Open or create a file*
- [Files](#): Introduction for current chapter

Test file open status

`ss_file_isopen` is a function defined in `ssfile.c`.

Synopsis:

`hid_t ss_file_isopen (ss_file_t *file, const char *name)`

Formal Arguments:

- `file`: Optional handle to a persistent File object.
- `name`: Optional real name of file to test for open status.

Description: Determines whether a file named `name` (names normalized with `ss_file_fixname` before being compared) is open, or whether the persistent file object `file` corresponds to an open file, or whether `file` is currently mapped to `name` depending on whether only `name` is specified, only `file` is specified, or both `name` and `file` are specified, respectively.

Return Value: Returns true (a positive HDF5 file handle) if the file is currently open (explicitly or implicitly); false if the file is not currently open; negative otherwise. The HDF5 file handle is not duplicated and the client should not invoke `H5Fclose` on the return value.

Parallel Notes: Independent. However, since the underlying HDF5 file was opened collectively, many operations on that file must necessarily be collective and therefore if the return value is to be used as a file (instead of a logic value) then this function will most likely be called collectively across the file's communicator.

Issues: The returned HDF5 file handle is not duplicated before being returned for three reasons: (1) the `H5Freopen` function returns a handle which does not participate in the same file mount structure as the original and thus we cannot guarantee that SSlib's file view would be consistent with that of the returned handle, (2) the `H5Freopen` function is collective which would preclude this function from being usable as an independent test of file availability, and (3) requiring the caller to `H5Fclose` the return value gets in the way of using this function as a predicate.

Since transient files are not supported by HDF5 there can be no HDF5 file handle for a file created with the `H5F_ACC_TRANSIENT` bit set. This function returns the integer 1 for such files, which is a positive true value but which is not a valid HDF5 file handle (or any valid handle for that matter).

See Also:

- [Files](#): Introduction for current chapter

Tests transient state of a file

`ss_file_istransient` is a function defined in `ssfile.c`.

Synopsis:

`htri_t ss_file_istransient (ss_file_t *file)`

Formal Arguments:

- `file`: A link to some File object

Description: This function tests whether `file` is a transient file. Transient files don't correspond to any underlying permanent storage (not even to an HDF5 file with a `core` driver).

Return Value: Returns true (positive) if `file` is a transient file; false if `file` is a permanent file; negative on error. It is an error to query a file which isn't in memory yet and therefore doesn't correspond to an open file.

Parallel Notes: Independent

See Also:

- [Files](#): Introduction for current chapter

Test file writability

`ss_file_iswritable` is a function defined in `ssfile.c`.

Synopsis:

`htri_t ss_file_iswritable (ss_file_t *file)`

Formal Arguments:

- `file`: A link to some File object

Description: Files can be open for read-only access or for read and write access. This function tests the writing capability of the file in question.

Return Value: Returns true (positive) if `file` was opened with the `H5F_ACC_RDWR` flag and false otherwise; returns negative on failure, including when `file` is not open.

Parallel Notes: Independent

See Also:

- [Files](#): Introduction for current chapter

Mark file as read-only

`ss_file_readonly` is a function defined in `ssfile.c`.

Synopsis:

```
herr_t ss_file_readonly (ss_file_t *file)
```

Description: A file can be marked as read-only even after it is opened for read-write. This is often useful when a file is created since a read-only file results in certain optimizations (such as knowing that such a file is always in a synchronized state). Marking a file as read-only can be substantially faster than closing the file and then reopening it since no I/O needs to happen.

The file should be in a synchronized state before this function is called.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the file's communicator.

Issues: It would be nice if HDF5 had a similar function, which would add an extra level of error checking to prevent SSlib from accidentally writing to the HDF5 file after it was marked as read-only.

See Also:

- [Files](#): Introduction for current chapter

Synchronize all scopes of a file

`ss_file_synchronize` is a function defined in `ssfile.c`.

Synopsis:

```
herr_t ss_file_synchronize (ss_file_t *file, ss_prop_t *props)
```

Formal Arguments:

- `file`: The file to synchronize.
- `props`: Optional synchronization properties.

Description: As mentioned in [ss_scope_synchronize](#), persistent object tables may become unsynchronized across the MPI tasks that own them. This function is simply a convenience function to synchronize all tables of all scopes that belong to the specified open file. See **Synchronization Algorithm** for more details.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the file's communicator.

Issues: If the caller supplies a property list then the ``err_newptrs'` integer property should be a member of that list since it is used internally by this function.

See Also:

- [ss_scope_synchronize](#): 13.7: *Synchronize a scope*
- [Files](#): Introduction for current chapter

Query file synchronization state

`ss_file_synchronized` is a function defined in `ssfile.c`.

Synopsis:

```
htri_t ss_file_synchronized (ss_file_t *file)
```

Formal Arguments:

- `file`: An open file

Description: As detailed in [ss_scope_synchronize](#), scopes may become out of sync across the MPI tasks that own them. This function queries all open scopes of the specified file to determine if any of them need to be synchronized, and may be faster than calling [ss_file_synchronize](#) even when all scopes are in a synchronized state.

Return Value: Returns true (positive) if all scopes of `file` are currently synchronized; false if any scope needs to be synchronized; negative on failure.

Parallel Notes: Collective across the file's communicator.

See Also:

- [ss_file_synchronize](#): 11.9: *Synchronize all scopes of a file*
- [ss_scope_synchronize](#): 13.7: *Synchronize a scope*
- [Files](#): Introduction for current chapter

Write pending data to file

`ss_file_flush` is a function defined in `ssfile.c`.

Synopsis:

```
herr_t ss_file_flush (ss_file_t *file, ss_prop_t *props)
```

Formal Arguments:

- `file`: The file to be flushed.
- `props`: Flushing properties. See [ss_scope_flush](#).

Description: As objects are created or modified the library caches changes in memory to prevent repeatedly writing to disk. This function writes that data to disk. However, to reduce the amount of communication necessary in cases where the caller knows the file is synchronized, the various data flushing functions do not synchronize first, so the caller should invoke [ss_file_synchronize](#). The flushing functions also do not generally guarantee that the data is flushed from HDF5 to the underlying file.

Flushing a transient file is a no-op.

The [ss_file_close](#) function both synchronizes and flushes.

Return Value: Returns non-negative on success, negative on failure. It is an error to attempt to flush a file which is not open.

Parallel Notes: Collective across the file's communicator (see [ss_file_open](#)).

Example: The following code flushes data to HDF5 and then tells HDF5 to flush its data to the file:

```
1  ss_file_t file = ss_file_open(...);  
2  ....  
3  ss_file_synchronize(file);
```

(continues on next page)

(continued from previous page)

```

4  ss_file_flush(file, NULL);
5  H5Fflush(ss_file_isopen(file), H5F_SCOPE_GLOBAL);

```

See Also:

- *ss_file_close*: 11.12: *Close a file*
- *ss_file_open*: 11.1: *Open or create a file*
- *ss_file_synchronize*: 11.9: *Synchronize all scopes of a file*
- *ss_scope_flush*: 13.9: *Write pending data to file*
- *Files*: Introduction for current chapter

Close a file

`ss_file_close` is a function defined in `ssfile.c`.

Synopsis:

```
herr_t ss_file_close (ss_file_t *file)
```

Formal Arguments:

- `file`: The file to be closed

Description: Closes a file and all scopes belonging to that file. All scopes belonging to the file are synchronized and flushed to the file first, and then all such scopes are closed. If the file contains scopes that were serving as registries for other files, those scopes will be removed from those files' registry stacks.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the file's communicator (see *ss_file_open*).

Issues: Closing a file simply causes the `cur_open` reference counter to be decremented in the GFile array. When this counter reaches zero the file is considered to be closed and we call `H5Fclose`, but we don't entirely reset the GFile array entry to zero just in case there are still things pointing into this file.

See Also:

- *ss_file_flush*: 11.11: *Write pending data to file*
- *ss_file_open*: 11.1: *Open or create a file*
- *ss_file_synchronize*: 11.9: *Synchronize all scopes of a file*
- *Files*: Introduction for current chapter

Attach an object registry scope

`ss_file_registry` is a function defined in *ssfile.c*.

Synopsis:

```
herr_t ss_file_registry (ss_file_t *file, ss_scope_t *registry)
```

Formal Arguments:

- `file`: The file that is getting the new registry.

- `registry`: The open scope to serve as the registry. This need not be a top-level scope though it usually is. It could even be some scope within `file` in an extreme case.

Description: A *find* operation searches a specific scope for objects that match some partially initialized key value. However, sometimes object definitions are in a separate object registry instead and should be “sucked into” the main file as necessary. An object registry is simply a stack of additional scopes to search when a *find* operation for the specified scope fails to locate any matching objects.

If a scope of `file` is searched during a *find* operation and results in no matches, then the `registry` scope is searched (registries are searched in the order defined with this function) and the object handle that gets returned is marked as coming from a registry. The current version of the library simply makes links to the registry scope, but a future version might copy the object from the registry into the specified `file` along with all prerequisites.

Registry lists are associated with the shared file information in the GFile array. That is, if two `ss_file_t` objects are opened and refer to the same underlying HDF5 file, then adding a registry to one of those `ss_file_t` links will cause the other link to also see the registry. This allows files that are opened implicitly to automatically use the same registry as their explicitly opened counterpart.

Note: The *File*, *Scope*, and *Blob* tables, which describe infrastructure, do not use object registries during *find* operations. If the `registry` scope is closed then it is automatically removed from all the files for which it’s serving as a registry.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent, although it’s typically used in such a way that all tasks of the `file` communicator make identical calls to this function to define a common set of object registries.

Example: Here’s how this function might be used:

```
1  ss_file_t file = ss_file_open("registry.saf", H5F_ACC_RDONLY, NULL);
2  ss_scope_t registry = ss_file_topscope(file);
3  ss_file_t myfile = ss_file_create("myfile.saf", H5F_ACC_RDWR, NULL);
4  ss_file_registry(myfile, registry);
```

See Also:

- [Files](#): Introduction for current chapter

Obtain top scope

`ss_file_topscope` is a function defined in `ssfile.c`.

Synopsis:

```
ss_scope_t * ss_file_topscope (ss_file_t *file, ss_scope_t *buf)
```

Formal Arguments:

- `file`: File for which to obtain a link to a top scope.
- `buf`: Optional buffer in which to store the resulting link.

Description: Given a file, return a link to the top scope of that file. This is really just a convenience function for [ss_pers_topscope](#) that exists mostly for compile-time type checking since this is an exceedingly common operation.

Return Value: Returns a pointer to a top-scope link on success (`buf` if that was non-null); null on failure.

Parallel Notes: Independent

See Also:

- [ss_pers_topscope](#): 7.11: Obtain top scope for an object

- *Files*: Introduction for current chapter

Global File Information

No description available.

Members

Print global file table

`ss_gfile_debug_all` is a function defined in `ssgfile.c`.

Synopsis:

`herr_t ss_gfile_debug_all (FILE *out)`

Description: Displays information about all global files

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Global File Information*: Introduction for current chapter

Print information about a global file entry

`ss_gfile_debug_one` is a function defined in `ssgfile.c`.

Synopsis:

`herr_t ss_gfile_debug_one (size_t idx, FILE *out, const char *prefix)`

Description: Prints information about a single entry in the global file array.

Return Value: Returns non-negative on success; negative on failure

Parallel Notes: Independent

See Also:

- *Global File Information*: Introduction for current chapter

Scopes

A scope is a collection of persistent object tables that belong to a single file. A file always has one top-level scope called “SAF” which is returned when the file is opened (see *ss_file_open*), but may have any number of additional scopes. Each scope is associated with a communicator which is a subset of the communicator for file containing the scope, and that communicator defines what tasks “own” the scope. Operations that open, close, or create scopes are generally collective over the file communicator; operations that modify the contents of a scope are generally collective over the scope communicator; operations that simply access the scope are generally independent.

Scopes satisfy a number of design goals:

- Scopes minimize communication by isolating certain objects to a subset of `MPI_COMM_WORLD`.
- Scopes provide a mechanism for controlled partial reads of the SAF metadata.

- Scopes provide a framework for transient objects.
- Scopes will allow for a crude form of object deletion but at a finer granularity than entire files.
- Scopes turn SAF's auxiliary files into standalone SAF databases.

A scope is a type of persistent object pointed to by a variable of type `ss_scope_t` (see **Persistent Objects**). As such, a scope is simply an entry in a table that gets written to a file. Each file has only one scope table, called `/SAF/Scopes`. That is, only the top-level scope contains a *Scopes* table and the first entry in that table is always the top-level scope itself, `/SAF`.

Since a scope is a persistent object, scopes are created, modified, destroyed, and queried just like any other persistent object. However, since a scope is also part of the file infrastructure, additional operations are defined and documented in this chapter.

Members

Opens a scope

`ss_scope_open` is a function defined in `ssscope.c`.

Synopsis:

```
herr_t ss_scope_open (ss_scope_t *scope, unsigned flags, ss_prop_t *props)
```

Formal Arguments:

- `scope`: A link to a scope object, probably the result of a *find* operation.
- `flags`: Various bit flags to control common scope open switches.
- `props`: Scope opening properties (see **Scope Properties**).

Description: Given a link to a scope (i.e., a link to an entry in the top-level *Scopes* table of some file that is currently open), open the scope. The `flags` argument determines the mode for opening the scope. The following bits are supported at this time:

`H5F_ACC_RDONLY`: The scope is opened for read-only access.

`H5F_ACC_RDWR`: The scope is opened for both read and write access.

`H5F_ACC_DEBUG`: Turn on scope debugging.

The `H5F_ACC_EXCL`, `H5F_ACC_TRUNC`, and `H5F_ACC_CREAT` bits are not supported by this function because they require participation of all tasks in the file's communicator, and therefore SSlib separates scope creation from scope opening.

The `H5F_ACC_RDWR` flag can only be used if the containing file is also `H5F_ACC_RDWR`.

The scope will be a transient scope if and only if the file was opened with *H5F_ACC_TRANSIENT*. Therefore this function simply ignores that bit in the `flags` vector.

Return Value: Returns non-negative on success; negative on failure.

It is an error to open a scope that is already open. However, since the original open might have been performed on a disjoint subset of tasks, the current operation might not be able to detect a duplicate open. If disjoint sets of tasks open the same scope for read-only access and no task has the scope open for writing then things will most likely work properly.

Parallel Notes: Collective across the scope's communicator, `props.comm`, defaulting to the same communicator as the file in which the scope exists.

See Also:

- *Scopes*: Introduction for current chapter

Closes a scope

`ss_scope_close` is a function defined in `ssscope.c`.

Synopsis:

```
herr_t ss_scope_close (ss_scope_t *scope)
```

Description: Closes the specified scope without totally destroying the memory representation. Specifically, the top-scope's Scope table is left intact as are the indirect map arrays in all scopes that have them. This is required for the following common scenario:

The application has two files called `File-A` and `File-B`. The application creates a new object (e.g., a quantity) in `File-A` without using the `SS_ALLSAME` flag and then creates another object (e.g., a unit) in `File-B` that refers to the object in `File-A`. The application closes `File-A` which closes all the scopes in that file. It then attempts to close `File-B`, which includes a synchronization and a flush. However, when flushing, SSlib will need to convert a persistent object link from the Memory state to the Closed state and convert its indirect object index to a direct object index. The only way this can be done is by having the indirect mappings for the table that contained the object in `File-A`.

The scope is assumed to already be synchronized and flushed. In fact, it would not even be possible to flush the scope from this function because doing so may require a call to `H5Dextend`, which is file collective.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the scope's communicator. This is the same set of tasks that originally opened the scope.

See Also:

- *Scopes*: Introduction for current chapter

Query scope open status

`ss_scope_isopen` is a function defined in `ssscope.c`.

Synopsis:

```
hid_t ss_scope_isopen (ss_scope_t *scope)
```

Formal Arguments:

- `scope`: A link to a scope object.

Description: A scope is either opened or closed at any given time on any given task. This function returns that status.

Return Value: Returns true (a positive HDF5 group handle) if `scope` is currently opened by the calling task; false if `scope` is currently closed; negative on error. The group handle is not duplicated and the caller should not invoke `H5Gclose` on the return value. This function returns an error (instead of false) if the scope is not even booted. To be "booted" means the scope object corresponds to some HDF5 group which is open.

Parallel Notes: Independent

Issues: The group handle return value is not duplicated by this function because (1) not doing so is consistent with `ss_file_isopen` and (2) doing so would require this function to be collective.

Since transient files are not supported by HDF5 there can be no HDF5 file handle for a scope created in a transient file. This function returns the integer 1 for such files, which is a positive true value but which is not a valid HDF5 group handle (or any valid handle for that matter).

See Also:

- *ss_file_isopen*: 11.5: *Test file open status*
- *Scopes*: Introduction for current chapter

Determines if scope is an open top-scope

`ss_scope_isopentop` is a function defined in `ssscope.c`.

Synopsis:

`htri_t ss_scope_isopentop (ss_scope_t *scope)`

Description: Determines if `scope` is open and a top-scope. A scope can be in an opened or closed state. Every file has exactly one top-level scope which is the first entry in that scope's Scope table.

Return Value: Returns true (positive) if `scope` is open and a top-scope or false if not. Returns negative on failure.

Parallel Notes: Independent

Issues: It might be better to just look to see if the specified scope has a communicator other than `MPI_COMM_NULL` and has a non-null pointer for a Scope table since only top-level scopes have a Scope table.

See Also:

- *Scopes*: Introduction for current chapter

Tests transient state of a scope

`ss_scope_istransient` is a function defined in `ssscope.c`.

Synopsis:

`htri_t ss_scope_istransient (ss_scope_t *scope)`

Formal Arguments:

- `scope`: Any open scope.

Description: This function tests whether `scope` is a transient scope. A scope is transient if it belongs to a transient file.

Return Value: Returns true (positive) if `scope` is a transient scope; false if `scope` is a permanent scope; negative on error. It is an error to query a scope which isn't open.

Parallel Notes: Independent

See Also:

- *Scopes*: Introduction for current chapter

Tests whether scope can be modified

`ss_scope_iswritable` is a function defined in `ssscope.c`.

Synopsis:

`htri_t ss_scope_iswritable (ss_scope_t *scope)`

Formal Arguments:

- `scope`: Any open scope.

Description: If a scope is opened for read-only access then the objects in that scope cannot be modified. This function tests for that condition.

Return Value: Returns true (positive) if `scope` is open with write access (that is, the `ss_scope_open` call was passed the `H5F_ACC_RDWR` flag); returns false if the scope is opened for read-only access; returns negative on failures. It is considered a failure if `scope` is not open on the calling task.

Parallel Notes: Independent

See Also:

- `ss_scope_open`: 13.1: *Opens a scope*
- *Scopes*: Introduction for current chapter

Synchronize a scope

`ss_scope_synchronize` is a function defined in `ssscope.c`.

Synopsis:

```
herr_t ss_scope_synchronize (ss_scope_t *scope, unsigned tableidx, ss_prop_t *props)
```

Formal Arguments:

- `scope`: A link to an open scope that should be synchronized.
- `tableidx`: Magic number to define which table to synchronize. If `tableidx` is `SS_TABLE_ALL` then all tables of the scope will be synchronized.
- `props`: See **Synchronization Properties**

Description: Various scope modifying operations that would normally be collective across the scope's communicator can be carried out locally. When this happens the various tasks of the scope's communicator may store differing information for the scope. This function is intended to synchronize the various tables of a particular scope across all the MPI tasks that "own" that scope. See **Synchronization Algorithm** for more details.

Return Value: Returns non-negative on success; negative on failure. It is an error to attempt to synchronize a scope that is not open.

Parallel Notes: Collective across the scope's communicator. Substantial communication may be required.

See Also:

- *Scopes*: Introduction for current chapter

Query scope synchronization state

`ss_scope_synchronized` is a function defined in `ssscope.c`.

Synopsis:

```
htri_t ss_scope_synchronized (ss_scope_t *scope, unsigned tableidx)
```

Formal Arguments:

- `scope`: A link to the scope whose synchronization state is to be queried.
- `tableidx`: Magic number to define which table to query. If `tableidx` is greater than or equal to `SS_NPERSL_CLASSES` then all tables of the specified scope must be in a synchronized state before this function returns true.

Description: As detailed in *ss_scope_synchronize*, scopes may become out of sync when tasks independently modify table entries. This function will query whether a scope (or some table of the scope) is out of sync without synchronizing the scope. In fact, even when the scope is in a synchronized state, calling this function may be faster than calling *ss_scope_synchronize*.

Return Value: Returns true (positive) if the scope is in a synchronized state; false if synchronization is necessary; negative on error. It is an error to make this query about a scope which is not open.

Parallel Notes: Collective across a superset of the scope's communicator. Communication is required within the scope communicator and other tasks will return true (positive) because the scope is not open there.

See Also:

- *ss_scope_synchronize*: 13.7: *Synchronize a scope*
- *Scopes*: Introduction for current chapter

Write pending data to file

`ss_scope_flush` is a function defined in *ssscope.c*.

Synopsis:

`herr_t ss_scope_flush` (`ss_scope_t *scope`, unsigned `tableidx`, `ss_prop_t` `UNUSED *props`)

Formal Arguments:

- `scope`: A link to the open scope to be flushed.
- `tableidx`: Magic number to define which table to flush, or `SS_TABLE_ALL`
- `props`: Scope flushing properties (none defined yet)

Description: Flushing a scope causes all of its tables to be written to HDF5. It does not include synchronizing the tables or telling HDF5 to flush its cached data to the file or telling the operating system to flush dirty blocks to disk. That can be done with code similar to the following:

```
1  ss_scope_flush(scope, SS_MAGIC(ss_field_t), properties);
2  H5Fflush(ss_scope_isopen(scope), H5F_SCOPE_GLOBAL); // flushes the whole hdf5 file,
↳and all mounts
```

Return Value: Returns non-negative on success, negative on failure.

Parallel Notes: Conceptually this function is collective across the scope's communicator, however because `ss_table_write` and `ss_string_flush` are file-collective due to HDF5 API restrictions this function must also be file collective. Fortunately the `scope` argument is available on all tasks of the file which makes this restriction easy to program around.

Issues: When flushing a specific table the variable length string values are not written to the file.

See Also:

- *Scopes*: Introduction for current chapter

Query the scope communicator

`ss_scope_comm` is a function defined in *ssscope.c*.

Synopsis:

`herr_t ss_scope_comm` (`ss_scope_t *scope`, `MPI_Comm *comm`, int `*self`, int `*ntasks`)

Formal Arguments:

- `comm`: Optional returned communicator, not duplicated.
- `self`: Optional returned calling task's rank within communicator
- `ntasks`: Optional returned size of communicator

Description: Given a scope, return the scope's communicator without dup'ing it. This is either the scope's communicator or the communicator of the file to which the scope belongs.

Return Value: Returns non-negative on success, negative on failure. The communicator is returned through the `comm` argument when the function is successful.

Parallel Notes: Independent

See Also:

- *Scopes*: Introduction for current chapter

Object Attributes

Attributes are small pieces of data which fall outside the scope of the sharable data model and thus cannot be represented by sets, fields, etc. The meaning of a particular attribute is determined by convention, requiring additional a priori agreement between the writer and the reader, often in the form of documentation or word of mouth.

Any persistent object may have zero or more attributes and each attribute has a name, datatype, element count (as if it were a one dimensional array), and a value. Operations on attributes are largely independent and since attributes are implemented as a scope table, those operations are similar to operations that can be performed on other persistent objects. One restriction on the attribute table, however, is that the attribute can only belong to an object stored in the same scope. This is necessary in order for an object to be able to efficiently find its attributes.

Members**Add a new attribute to an object**

`ss_attr_new` is a function defined in `ssattr.c`.

Synopsis:

```
ss_attr_t * ss_attr_new (ss_pers_t *owner, const char *name, hid_t type, size_t count, const void *value,
                        unsigned flags, ss_attr_t *buf, ss_prop_t *props)
```

Formal Arguments:

- `owner`: The object with which the new attribute is associated.
- `name`: The name of the new attribute.
- `type`: The datatype of the attribute.
- `count`: Number of values in the attribute.
- `value`: Optional array of `count` values each of type `type`. If this array is not supplied then the attribute's value will be initialized to all zero bytes.
- `flags`: Bit flags, such as *SS_ALLSAME*.
- `buf`: The optional buffer for the returned attribute link.
- `props`: Attribute properties (none defined yet).

Description: This function adds a new attribute to the `owner` object (which must not be an attribute itself). An attribute can be thought of as a one dimensional array of values all having the same datatype.

Return Value: Returns a non-null attribute link on success; null on failure. If `buf` is supplied then it will be the success return value.

Parallel Notes: Independent. However if all tasks are collectively creating a single attribute and all are passing the same `type`, `count`, and `value` then they may pass the `SS_ALLSAME` bit in the `flags` argument, thereby allowing a synchronization to do less work later. When the `SS_ALLSAME` bit is passed then the call should be collective across the communicator of the scope containing the `owner` object.

See Also:

- *Object Attributes*: Introduction for current chapter

Find attributes for an object

`ss_attr_find` is a function defined in `ssattr.c`.

Synopsis:

```
ss_attr_t * ss_attr_find (ss_pers_t *owner, const char *name, size_t nskip, size_t maxret, size_t *nret,
                          ss_attr_t *result)
```

Formal Arguments:

- `owner`: The object for which we're searching for attributes.
- `name`: An optional attribute name on which to restrict the search.
- `nskip`: Skip the first `SKIP` matching attributes.
- `maxret`: Return at most `maxret` matching attributes. If the caller passes `SS_NOSIZE` then all matching attributes are returned. If more than `maxret` attributes could be returned the remainder are simply discarded. If `result` is non-null then this argument should reflect the size of that array.
- `nret` [OUT]: The number of attributes stored in the returned array of links.
- `result`: An optional buffer in which to store links to the matching attributes. If supplied, this will be the successful return value. The constant `SS_PERS_TEST` can be supplied in order to prevent the library from allocating a return value (this is useful if the caller simply wants to count the matches).

Description: This function finds all attributes for the persistent object `owner` and returns handles to those attributes. The returned array of handles can be restricted by supplying an attribute name which must match all returned attributes. The `SKIP` and `maxret` arguments can select a contiguous subset of the available attributes.

Return Value: On success, returns an array of links to matching attributes; returns a null pointer on failure. If no matching attributes are found then a non-null malloc'd pointer is returned and `nret` points to zero. That is, the case where no attributes match the search criteria is not considered an error.

Parallel Notes: Independent

See Also:

- *Object Attributes*: Introduction for current chapter

Count matching attributes

`ss_attr_count` is a function defined in `ssattr.c`.

Synopsis:

size_t **ss_attr_count** (ss_pers_t *owner, const char *name)

Description: Counts the number of attributes that belong to `owner` and have the optional string `name` as their name. If no `name` is supplied then all attributes belonging to `owner` are counted.

Return Value: On success, the number of attributes found to match the `owner` and `name` pair; `SS_NOSIZE` on failure.

Parallel Notes: Independent

See Also:

- *Object Attributes*: Introduction for current chapter

Obtain attribute value

`ss_attr_get` is a function defined in `ssattr.c`.

Synopsis:

void * **ss_attr_get** (ss_attr_t *attr, hid_t type, size_t offset, size_t nelmts, void *buffer)

Formal Arguments:

- `attr`: The attribute in question.
- `type`: The desired datatype of the returned value.
- `offset`: The first element of the value to return (an element index, not byte index)
- `nelmts`: The total number of elements to return. If the `offset` and `nelmts` arguments describe a range of elements that is outside that which is known to the attribute then an error is raised. If `nelmts` is `SS_NOSIZE` then the number of returned values is not limited (except perhaps by the non-zero `offset`).
- `buffer`: The optional buffer in which to store the returned values.

Description: This function extracts the attribute value, or subpart thereof. The value is converted to the desired `type`, which must be conversion compatible with the datatype used to store the attribute value. The converted value is copied into the optional caller-supplied `buffer` (or a buffer is allocated). If the value consists of more than one element then desired elements to be returned can be specified with an offset and length.

Return Value: Returns a pointer (`buffer` if non-null) on success; null on failure. If the caller doesn't supply `buffer` then this function allocates one.

Parallel Notes: Independent

See Also:

- *Object Attributes*: Introduction for current chapter

Change an attribute value

`ss_attr_put` is a function defined in `ssattr.c`.

Synopsis:

herr_t **ss_attr_put** (ss_attr_t *attr, hid_t type, size_t offset, size_t nelmts, const void *value, unsigned flags)

Formal Arguments:

- `attr`: The attribute in question.
- `type`: The datatype of `value`.

- `offset`: The element number at which to put `value`.
- `nelmts`: The number of elements in `value`.
- `value`: The value to be written to the attribute.
- `flags`: Flags such as [*SS_ALLSAME*](#).

Description: An attribute's value can be changed by calling this function. If the attribute stores more than one value then the supplied new `value` can be for either the whole attribute or for just part of the attribute as determined by the `offset` and `nelmts` arguments. The datatype of each element of `value` is specified by `type` and must be conversion compatible with the datatype already registered with the attribute.

Return Value: Returns non-negative on success; negative on failure. It is an error if `offset` and `nelmts` specify a range of elements outside that which the attribute already knows about.

Parallel Notes: Independent. However, if the [*SS_ALLSAME*](#) bit is passed in the `flags` argument the call should be collective across all tasks of the communicator for the scope that owns the attribute and all such tasks must pass identical values for `offset` and `nelmts` and a `type` and `value` such that the converted value is identical on all tasks.

See Also:

- [*Object Attributes*](#): Introduction for current chapter

Modify attribute type and size

`ss_attr_modify` is a function defined in `ssattr.c`.

Synopsis:

`herr_t` **ss_attr_modify** (`ss_attr_t` **attr*, `hid_t` *type*, `size_t` *nelmts*, unsigned *flags*)

Formal Arguments:

- `attr`: The attribute whose size will be changed.
- `type`: The new datatype for the attribute, or `H5I_INVALID_HID` if the type is not to be changed. If the datatype is changed but is conversion compatible with the previous type then the attribute's data will be converted to the new datatype. Otherwise the attribute's data will be initialized to all zero.
- `nelmts`: The new number of elements in the attribute value, or `SS_NOSIZE` if the number of elements is not to be changed. If the number of elements decreases then the extra elements are discarded. If the number of elements increases then the new elements will be initialized to all zero bytes.
- `flags`: Bit flags such as [*SS_ALLSAME*](#).

Description: This function modifies the storage datatype and/or number of elements of an attribute.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent. However if the [*SS_ALLSAME*](#) bit is set in the `flags` argument then this function is collective across the communicator of the scope that owns the attribute and all tasks must pass identical values for the `type` and `nelmts`.

See Also:

- [*Object Attributes*](#): Introduction for current chapter

Query attribute metadata

`ss_attr_describe` is a function defined in `ssattr.c`.

Synopsis:

```
const char * ss_attr_describe (ss_attr_t *attr, ss_pers_t *owner, hid_t *type, size_t *nelmts)
```

Formal Arguments:

- `attr`: The attribute to be described.
- `owner` [OUT]: Optional object to which attribute belongs.
- `type` [OUT]: Optional datatype of attribute data. The caller should invoke
- `nelmts` [OUT]: Optional number of elements stored by the attribute.

Description: This is really just a convenience function so if the caller wants the attribute datatype they don't need to do the work of calling `H5decode` on the attribute `type` field.

Return Value: Returns the attribute name on success; null on failure. The name is simply the *ss_string_ptr* value of the `name` field of the attribute.

Parallel Notes: Independent

See Also:

- *ss_string_ptr*: 9.2: Obtain pointer into string object
- *Object Attributes*: Introduction for current chapter

Values

These functions operate on values of pretty much any HDF5 datatype.

Members

Print an arbitrary datum

`ss_val_dump` is a function defined in `ssval.c`.

Synopsis:

```
herr_t ss_val_dump (void *val, hid_t type, void *_parent, FILE *out, const char *html_tag)
```

Formal Arguments:

- `val`: Value to be printed
- `type`: Datatype of `val`
- `_parent`: Optional persistent object into which `val` points
- `out`: Stream to which output should be sent
- `html_tag`: Optional HTML tag to use in output; `NULL` means output plain text

Description: Given memory of a certain type, print it to the specified stream.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Values*: Introduction for current chapter

Value comparison flags

`ss_val_cmp_t` is a collection of related C preprocessor symbols defined in `ssval.h`.

Synopsis:

`SS_VAL_CMP_DFLT`:

`SS_VAL_CMP_SUBSTR`: Strings: NUL terminated substring

`SS_VAL_CMP_SUBMEM`: Strings: binary comparison of substring

`SS_VAL_CMP_RE`: Strings: NUL terminated regular expression

`SS_VAL_CMP_RE_EXTENDED`:

`SS_VAL_CMP_RE_ICASE`:

`SS_VAL_CMP_RE_NEWLINE`:

`SS_VAL_CMP_EQ`: Link: compare with *ss_pers_eq*

`SS_VAL_CMP_EQUAL`: Link: compare with *ss_pers_equal*

Description: The various comparison functions (such as `ss_val_cmp`) can be influenced to use different algorithms during their operation by passing a bit vector describing the mode of operation. For `ss_val_cmp` the bit vector is contained in the first non-zero byte of the mask for each value being compared. Generally, if the flags bits are unrecognized or unsupported by a particular comparator a default algorithm is used.

The Default Algorithm

All comparison functions support a notion of a default comparison algorithm. This is indicated by setting all eight low-order bits in the flag, which can be done with the constant `SS_VAL_CMP_DFLT`.

String Comparisons

`SS_VAL_CMP_SUBSTR`: If the *key* value is a substring of the *buffer* being tested then the comparator reports that they are equal. Otherwise the *key* is always considered to be less than the *buffer*. Both the *key* and *buffer* must be C-style NUL-terminated strings in order for a match to be detected.

`SS_VAL_CMP_SUBMEM`: This is similar to `SS_VAL_CMP_SUBSTRING` except the *key* and *mask* are considered to be bytes of memory and all bytes of the *key* (including NUL bytes if any) must match at some location in *buffer*. If there is no match then the *key* is considered to be less than the *buffer*.

`SS_VAL_CMP_RE`: The *key* value is interpreted as a POSIX regular expression and if that regular expression matches the contents of *buffer* then the comparator reports that *key* and *buffer* are equal, otherwise the *key* is considered to be less than the *buffer*.

If the `SS_VAL_CMP_RE` bit is set then the following bits are also supported (they actually each contain the `SS_VAL_CMP_RE` bit as well):

`SS_VAL_CMP_RE_EXTENDED`: The *key* value is treated as an extended regular expression rather than a basic regular expression. Refer to the documentation for the `REG_EXTENDED` flag of `regcomp` for details.

`SS_VAL_CMP_RE_ICASE`: Ignore case when matching letters.

`SS_VAL_CMP_RE_NEWLINE`: Treat a newline in *key* as dividing *buffer* into multiple lines, so that a dollar sign can match before the newline and a carat can match after. Also, don't permit a dot to match a newline, and don't permit a complemented character class (square brackets with a leading carat) to match a newline. Otherwise, newline acts like any other ordinary character.

Object Link Comparisons

SS_VAL_CMP_EQ: When comparing two persistent object links use *ss_pers_eq*. This is the default.

SS_VAL_CMP_EQUAL: When comparing two persistent object links use *ss_pers_equal*.

Issues: Since these bits must be passed as the bytes of a mask they must be only eight bits wide. The type, however, is defined as `unsigned` because of argument promotion rules.

See Also:

- *ss_pers_eq*: 7.12: *Determine link equality*
- *ss_pers_equal*: 7.13: *Determine object equality*
- *Values*: Introduction for current chapter

HDF5

These functions provide features that are missing from HDF5.

Members

Compares two datatypes

`H5Tcmp` is a function defined in `sshdf5.c`.

Synopsis:

```
int H5Tcmp (hid_t t1, hid_t t2)
```

Description: This is essentially a public version of `H5T_cmp`

Return Value: 1, 0, or -1 depending on whether T1 is less than, equal to, or greater than T2. Returns -2 on failure.

Parallel Notes: Independent

Issues: This function calls internal HDF5 functions for which we have no prototypes.

See Also:

- *HDF5*: Introduction for current chapter

Extra native datatypes

`H5T_NATIVE` is a collection of related C preprocessor symbols defined in `sshdf5.h`.

Synopsis:

```
H5T_NATIVE_SIZE:
```

```
H5T_NATIVE_HID:
```

```
H5T_NATIVE_VOIDP:
```

```
H5T_NATIVE_MPI_COMM:
```

```
H5T_NATIVE_MPI_INFO:
```

Description: These are useful native datatypes that are missing from HDF5.

See Also:

- [HDF5](#): Introduction for current chapter

Declare a file to be transient

H5F_ACC_TRANSIENT is a symbol defined in sshdf5.h.

Synopsis:

H5F_ACC_TRANSIENT

Description: When this bit is passed as the `*flags*` argument to `ss_file_open` or `ss_file_create` then a no-op HDF5 file is created. Since HDF5 doesn't support this functionality, SSlib simply notes that there is no underlying HDF5 file.

Issues: We commandeer a high-order bit for our purposes, knowing that the H5F API uses the low-order bits. This bit will never make it into HDF5, but we need to insure that it doesn't conflict with the other H5F_ACC bits.

See Also:

- `ss_file_create`: 11.4: *Create a new file*
- `ss_file_open`: 11.1: *Open or create a file*
- [HDF5](#): Introduction for current chapter

Datatypes

No description available.

Members

Maximum IndexSpec dimensionality

SS_MAX_INDEXDIMS is a symbol defined in ssopers.h.

Synopsis:

SS_MAX_INDEXDIMS

Description: This constant represents the maximum number of dimensions that can be described by an index specification.

See Also:

- [Datatypes](#): Introduction for current chapter

Number of base quantities

SS_MAX_BASEQS is a symbol defined in ssopers.h.

Synopsis:

SS_MAX_BASEQS

Description: All quantities can be defined in terms of seven basic quantities.

See Also:

- [Datatypes](#): Introduction for current chapter

Wildcard SIL role

SAF_SROLE_ANY is a symbol defined in `sspers.h`.

Synopsis:

SAF_SROLE_ANY

Description: This constant can be used as a wildcard when searching based on SIL role.

See Also:

- *Datatypes*: Introduction for current chapter

SIL roles

`ss_silrole_t` is an enumerated type defined in `sspers.h`.

Synopsis:

“““.

SAF_SROLE_SPACE: sil is for space

SAF_SROLE_STATE: for state space

SAF_SROLE_PARAM: sil is generic param space

SAF_SROLE_CTYPE: sil is for a cell

SAF_SROLE_ATYPE: sil is for an algebraic type

SAF_SROLE_USERD: sil has a user defined role

SAF_SROLE_SUITE: for a suite

Description: These are the roles that can appear in a subset inclusion lattice. One additional constant `SAF__SROLE_ANY`, although not part of this enumeration due to it never appearing in a file, can be used as a wildcard for searching.

See Also:

- *Datatypes*: Introduction for current chapter

Miscellaneous

No description available.

Members

Global bit flags

SS is a collection of related C preprocessor symbols defined in `sslib.h`.

Synopsis:

SS_ALLSAME: All applicable tasks are supplying identical data or performing the same operation. For instance, in a call to *ss_pers_new* this indicates that the new object can be “born synchronized” because all tasks will have the same value for the object without needing to communicate. Absence of this bit simply means that different tasks might be

supplying different data but doesn't guarantee that the data is different. (Note: this value must be distinct from `TRUE`, but don't worry, it's checked at runtime.)

`SS_STRICT`: Flag that causes certain functions to "try harder" to do something

Description: Various bit flags that are useful to different interfaces. These global flags use the high-order bits of a word while the interface-specific flags use the low-order bits.

See Also:

- *Miscellaneous*: Introduction for current chapter

Notes

Miscellaneous notes.

Members

Debugging

SSLlib has a fair amount of debugging support built in, most of which is runtime selectable from the `SSLIB_DEBUG` environment variable and documented in the `ss_debug_env` function.

Members

Enter an interactive debugging loop

`ss_debug` is a function defined in `ssdebug.c`.

Synopsis:

`herr_t ss_debug` (void)

Description: Read and execute debugging commands if the *commands*' word was present in the `:ref:'SSLIB_DEBUG <SSLIB>` environment variable. The file containing the commands is also specified in `SSLIB_DEBUG`. See `ss_debug_env` for complete documentation for that variable. If *commands*' is not specified in `:ref:'SSLIB_DEBUG <SSLIB>` for the calling task or if the command input file is empty then this function is a no-op.

The commands accepted by this function are defined in `ss_debug.s`. In addition, the command *detach*' causes this function to immediately return. The `'ss_debug'` function may be called more than once in any given executable.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- `ss_debug_env`: 22.2: Parse debug setup statements
- `ss_debug_s`: 22.3: Evaluate a debug command
- *Debugging*: Introduction for current chapter

Parse debug setup statements

`ss_debug_env` is a function defined in `ssdebug.c`.

Synopsis:

`herr_t ss_debug_env (MPI_Comm UNUSED_SERIAL comm, const char *s_const)`

Formal Arguments:

- `comm`: The library communicator. Pass any integer value when using a version of SSlib compiled without MPI support.
- `s_const`: Optional string to use instead of looking at the `SSLIB_DEBUG` environment variable. Pass null to use `SSLIB_DEBUG` instead. Passing an empty string (or all white space) accomplishes nothing. Task zero broadcasts this string to all the other tasks.

Description: This function looks at the contents of the `SSLIB_DEBUG` environment variable. It is a semicolon separated list of terms which control various things. Valid terms are:

task=*n**: Controls which tasks will be affected by subsequent debugging terms. A value of *n* with an initial plus sign will add task *n* to the list of selected tasks; a leading minus sign removes the task from the list; lack of a plus or minus makes task *n* the only selected task. The value `all` or `none` can also be supplied which selects all tasks or no tasks, respectively. The value can also be a comma-separated list of task ranks which acts the same as if multiple

`task` terms had been specified (the plus or minus sign should be at the beginning of the list and applies to all values of the list).

error=*n**: When errors are pushed onto the error stack they are each given a unique (within a task)

identification number. When this `error` debugging term is specified then the debugger of choice is invoked when error number *n* is pushed onto the stack. If the equal sign and error number are omitted then the debugger is never started for an error, but the error stack will display the error identification numbers. Only one error number can be specified per task—if more flexibility is needed then the application can be run under a debugger with a breakpoint set in `ss_error`.

file=*name**: Selects the output file to use for subsequent debugging terms for the selected tasks. The `file` will be created if it doesn't exist or truncated if it does exist. The name is actually a `printf` format string and the first format specifier (if present) should be for an integer task number. If *name* is the word `none` then output is disabled; if *name* is a positive integer then the specified file descriptor is used without attempting to open it (this is useful if the descriptor was opened with the shell). If a number and name are both specified separated by a comma then the name is opened and dup'd to the desired file descriptor. If the name begins with a '<' character then the file is opened for read-only.

stop: The specified MPI task(s) will print their MPI rank and process ID and then suspend themselves, giving an opportunity for a debugger to attach.

pause="N": The specified MPI task(s) will immediately pause for N seconds. This is useful when a task needs to give a debugger (such as `strace`) to automatically attach to child processes.

debugger=*name**: Specifies which debugger should be used. The default is `'ddd'`.

debug: The specified debugger (or `ddd`) is started for the affected task or tasks. This probably only works on systems that have a `/proc/self/exe` link to the executable and the `DISPLAY` environment variable set properly for the affected task. If the non-default debugger is desired then the `'debugger'` keyword must appear before this `'debug'` keyword.

signal: Start the debugger when a task is about to die from certain signals (those that signify a program error). The task is suspended (although other signal handlers might still be executed) and must be explicitly killed. The `'debug'` keyword takes precedence over `'signal'`.

stack: Turn automatic error reporting on or off for selected tasks depending on the current setting for the file descriptor. When off, errors are reported by return values as usual and the error stack contains information about the error, but the stack is not automatically printed. The default is that errors are printed to stderr.

pid: Print the process ID for all selected tasks. This is useful when various tools (such as `valgrind`) print PIDs but have no way of knowing the MPI task number.

mpi: Do not register an MPI error handler in the `ss_init` call.

banner=“STR“: Display the specified string value on stderr when `ss_init` is about to return. This is normally used in conjunction with the config file to notify users that they should recompile their application with a newer version of sslib.

commands: Enables the `ss_debug` calls that might appear in applications. The ‘file’ term should be used before this term in order to specify from where the debug commands should be read (don’t forget to use the ‘<’ in front of the file name in order to open it for read-only). If no file is specified then SSLib attempts to read the commands from the stderr stream, which may cause the commands to be read from the controlling terminal in certain situations (but it’s usually better to be explicit by providing the ‘file=<*dev*tty’ term). Specifying an empty file such as `*dev*null` has essentially the same effect as if the ‘commands’ term was not given.

warnings: For the selected MPI tasks, send all miscellaneous SSLib warning messages to the selected file.

check=*what*: Turns on or off various categories of internal consistency checking, some of which incur considerable runtime expense. The *what* is a comma-separated list of category names where that category of checking is turned off if introduced with a minus sign and on otherwise. Only selected tasks are affected. See the table below for a list of categories.

The following internal consistency checking categories are defined. Some categories can take a comma-separated list of attributes separated from the category name by an equal sign. When a category is followed by an equal sign then it must be the last category listed for that `check` term, but additional categories can be specified with additional `check` terms.

sync: When turned on, SSLib will check for many situations where a call to `ss_pers_modified` (or the macro `SS_PERS_MODIFIED`) was accidentally omitted by computing and caching checksums. If the `error` attribute is specified then such situations will be considered errors instead of just generating debugging information on the warning stream. If the `bcast` attribute is specified then information about which objects are transmitted will be displayed to the warning stream.

2pio: SSLib will display certain information about 2-phase I/O if this is turned on. For instance, when aggregation tasks are chosen for a blob the mapping from dataset addresses to aggregators is displayed. The `task` setting doesn’t affect this flag since it’s always task zero that displays this collective information.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Collective across the library communicator. We do this because environment variables are sometimes only available at certain tasks (task zero of the library communicator must have the environment variable).

Example: Example 1: To start the DDD debugger on task 17:

```
1 SSLIB_DEBUG='task=17;debug' ...
```

Example 2: To stop all tasks but task 17:

```
1 SSLIB_DEBUG='task=-17;stop' ...
```

Example 3: To cause task 17 to report errors to a file named “task17.err” and no other task to report errors:

```
1 SSLIB_DEBUG='file=none;stack;task=17;file=task17.err;stack' ...
```

Example 4: Cause HDF5 to emit tracing information to files like `task001.trace`, `task002.trace`, etc. The thing to watch out for here is that HDF5 gets initialized before SSLib and if file descriptor 99 is not open then tracing

is disabled. So we rely on the shell to supply an initial file for descriptor 99 which SSLib will swap out from under HDF5. Until the swap occurs, all tasks will emit tracing to the shell-supplied file:

```
SSLIB_DEBUG="file=99,task%03d.trace" HDF5_DEBUG=99,trace 99>tasks.trace ...
```

Example 5: Invoke a debugger on any task that fails an assertion or receives certain other normally fatal signals. Use gdb instead of the default ddd.

```
SSLIB_DEBUG='debugger=gdb;signal' ...
```

Example 6: To cause each task to redirect its standard error output to its own file:

```
SSLIB_DEBUG='file=2,stderr.%04d' ...
```

Example 7: To type commands interactively to SSLib one makes a call to *ss_debug* in the application and then uses *SSLIB_DEBUG* as follows:

```
SSLIB_DEBUG='task=0;file=<commands.txt;commands' ...
```

Example 8: To turn off the warning/debug messages that are normally emitted from SSLib on the stderr stream one would do the following:

```
SSLIB_DEBUG='file=/dev/null;warnings' ...
```

See Also:

- *SS_PERS_MODIFIED*: 7.29: *Mark object as modified*
- *ss_debug*: 22.1: *Enter an interactive debugging loop*
- *ss_error*: 2.5: *Start debugger for error*
- *ss_init*: 2.8: *Initialize the library*
- *ss_pers_modified*: 7.19: *Mark object as modified*
- *Debugging*: Introduction for current chapter

Evaluate a debug command

ss_debug_s is a function defined in *ssdebug.c*.

Synopsis:

```
herr_t ss_debug_s (const char *cmd)
```

Description: This function parses a debugging command in *cmd* and executes it. The first word in the string is the name of the command and the rest of the string contains the arguments for that command.

- *files*:
Display information about all known files.
- *classes*:
Show names of all object classes.
- *class spec*:
Display information about the specified persistent object. *Class* is one of the class words such as ‘set’ or ‘field’, etc. Use the command ‘classes’ for a complete list.

Return Value: Returns non-negative on success; negative on failure.

Parallel Notes: Independent

See Also:

- *Debugging*: Introduction for current chapter

Overloaded Definitions

These objects have multiple definitions.

Members

SS_MAGIC_ss

This object has overloaded definitions.

Members

Permuted Index

Table 1.3: Permuted Index

Concept	Key	Reference
Miscellaneous	(class 0x5af01000)	<i>SS_MAGIC_ss</i>
Persistent object links	(class 0x5af02000)	<i>SS_MAGIC_ss</i>
Persistent objects	(class 0x5af03000)	<i>SS_MAGIC_ss</i>
Miscellaneous (class	0x5af01000)	<i>SS_MAGIC_ss</i>
Persistent object links (class	0x5af02000)	<i>SS_MAGIC_ss</i>
Persistent objects (class	0x5af03000)	<i>SS_MAGIC_ss</i>
Initiate	2-phase I/O	<i>ss_blob_synchronize</i>
Print information	about a global file entry	<i>ss_gfile_debug_one</i>
Inquire	about array file datatype	<i>ss_array_targeted</i>
Obtain information	about referenced files	<i>ss_file_references</i>
	Add a new attribute to an object	<i>ss_attr_new</i>
	Add new property to a list	<i>ss_prop_add</i>
Debugging	aid	<i>ss_pers_debug</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Synchronize	all scopes of a file	<i>ss_file_synchronize</i>
Append one string to	another	<i>ss_string_cat</i>
	Append one string to another	<i>ss_string_cat</i>
Queries/sets property list	appendability	<i>ss_prop_appendable</i>
Print an	arbitrary datum	<i>ss_val_dump</i>
Change the size of a variable length	array	<i>ss_array_resize</i>
Free memory associated with the	array	<i>ss_array_reset</i>
Modify part of an	array	<i>ss_array_put</i>
Change	array element datatype	<i>ss_array_target</i>
Inquire about	array file datatype	<i>ss_array_targeted</i>
Store a byte	array in a string	<i>ss_string_memset</i>
Obtain	array value	<i>ss_array_get</i>
	Asserts object runtime class	<i>SS_ASSERT_CLASS</i>
	Asserts object runtime type	<i>SS_ASSERT_TYPE</i>
	Asserts object runtime type and existence	<i>SS_ASSERT_MEM</i>
Free memory	associated with the array	<i>ss_array_reset</i>
Free memory	associated with the string	<i>ss_string_reset</i>
HDF5	async callback	<i>ss_aio_hdf5_cb</i>
Terminate the	asynchronous I/O subsystem	<i>ss_aio_finalize</i>
Initialize the	asynchronous I/O subsystem	<i>ss_aio_init</i>
	Attach an object registry scope	<i>ss_file_registry</i>
Query	attribute metadata	<i>ss_attr_describe</i>
Add a new	attribute to an object	<i>ss_attr_new</i>
Modify	attribute type and size	<i>ss_attr_modify</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Change an	attribute value	<i>ss_attr_put</i>
Obtain	attribute value	<i>ss_attr_get</i>
Count matching	attributes	<i>ss_attr_count</i>
Find	attributes for an object	<i>ss_attr_find</i>
Number of	base quantities	<i>SS_MAX_BASEQS</i>
Test whether an object can	be modified	<i>ss_pers_iswritable</i>
Tests whether scope can	be modified	<i>ss_scope_iswritable</i>
Declare a file to	be transient	<i>H5F_ACC_TRANSIENT</i>
	Begin a functionality test	<i>SS_CHECKING</i>
	Bind a blob to a dataset	<i>ss_blob_bind_f</i>
	Bind a blob to a dataset	<i>ss_blob_bind_f1</i>
	Bind a blob to memory	<i>ss_blob_bind_m</i>
	Bind a blob to memory	<i>ss_blob_bind_m1</i>
Global	bit flags	<i>SS</i>
Create a new	blob	<i>ss_blob_new</i>
Create storage for a	blob	<i>ss_blob_mkstorage</i>
Extend a	blob	<i>ss_blob_extend</i>
Extend a	blob	<i>ss_blob_extend1</i>
Query dataset bound to a	blob	<i>ss_blob_bound_f</i>
Query dataset bound to a	blob	<i>ss_blob_bound_f1</i>
Query memory bound to a	blob	<i>ss_blob_bound_m</i>
Query memory bound to a	blob	<i>ss_blob_bound_m1</i>
Write data to a	blob	<i>ss_blob_write</i>
Write data to a	blob	<i>ss_blob_write1</i>
Query	blob extent	<i>ss_blob_space</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Bind a	blob to a dataset	<code>ss_blob_bind_f</code>
Bind a	blob to a dataset	<code>ss_blob_bind_f1</code>
Bind a	blob to memory	<code>ss_blob_bind_m</code>
Bind a	blob to memory	<code>ss_blob_bind_m1</code>
	Block until requests complete	<code>ss_aio_suspend</code>
Query dataset	bound to a blob	<code>ss_blob_bound_f</code>
Query dataset	bound to a blob	<code>ss_blob_bound_f1</code>
Query memory	bound to a blob	<code>ss_blob_bound_m</code>
Query memory	bound to a blob	<code>ss_blob_bound_m1</code>
Store a	byte array in a string	<code>ss_string_memset</code>
Compare persistent string with	C string	<code>ss_string_cmp_s</code>
Get a	C string from a persistent string	<code>ss_string_get</code>
Store a	C string in a persistent string	<code>ss_string_set</code>
HDF5 async	callback	<code>ss_aio_hdf5_cb</code>
Test whether an object	can be modified	<code>ss_pers_iswritable</code>
Tests whether scope	can be modified	<code>ss_scope_iswritable</code>
	Change a floating-point property value	<code>ss_prop_set_f</code>
	Change a property value	<code>ss_prop_set</code>
	Change a signed integer property value	<code>ss_prop_set_i</code>
	Change an attribute value	<code>ss_attr_put</code>
	Change an unsigned integer property value	<code>ss_prop_set_u</code>
	Change array element datatype	<code>ss_array_target</code>
	Change link state	<code>ss_pers_state</code>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Change the size of a variable length array	<i>ss_array_resize</i>
	Check if link is null	<i>SS_PERS_ISNULL</i>
Compute a	checksum for a persistent object	<i>ss_pers_cksum</i>
Asserts object runtime	class	<i>SS_ASSERT_CLASS</i>
Obtain magic number	class	<i>SS_MAGIC_CLASS</i>
	Close a file	<i>ss_file_close</i>
	Closes a scope	<i>ss_scope_close</i>
Evaluate a debug	command	<i>ss_debug_s</i>
Insert	commas into an integer	<i>ss_insert_commas</i>
Query the scope	communicator	<i>ss_scope_comm</i>
	Compare persistent string with C string	<i>ss_string_cmp_s</i>
	Compares two datatypes	<i>H5Tcmp</i>
	Compares two persistent objects	<i>ss_pers_cmp</i>
	Compares two persistent objects	<i>ss_pers_cmp_</i>
	Compares two variable length strings	<i>ss_string_cmp</i>
Value	comparison flags	<i>ss_val_cmp_t</i>
Block until requests	complete	<i>ss_aio_suspend</i>
	Compute a checksum for a persistent object	<i>ss_pers_cksum</i>
	Construct a magic number	<i>SS_MAGIC_CONS</i>
	Constructor	<i>SS_PERS_NEW</i>
Copy	constructor	<i>SS_PERS_COPY</i>
Property	constructor	<i>ss_prop_cons</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Copy an object	<i>ss_pers_copy</i>
	Copy constructor	<i>SS_PERS_COPY</i>
	Count matching attributes	<i>ss_attr_count</i>
Open or	create a file	<i>ss_file_open</i>
	Create a new blob	<i>ss_blob_new</i>
	Create a new file	<i>ss_file_create</i>
	Create a new persistent object	<i>ss_pers_new</i>
	Create a new property list from an existing list	<i>ss_prop_dup</i>
	Create a new property list from scratch	<i>ss_prop_new</i>
	Create an object link	<i>ss_pers_refer</i>
	Create storage for a blob	<i>ss_blob_mkstorage</i>
Returns	current status of a request	<i>ss_aio_error</i>
Read	data from a file	<i>ss_blob_read</i>
Read	data from a file	<i>ss_blob_read1</i>
Write	data to a blob	<i>ss_blob_write</i>
Write	data to a blob	<i>ss_blob_write1</i>
Write pending	data to file	<i>ss_scope_flush</i>
Write pending	data to file	<i>ss_file_flush</i>
Flush pending	data to HDF5	<i>ss_blob_flush</i>
Bind a blob to a	dataset	<i>ss_blob_bind_f</i>
Bind a blob to a	dataset	<i>ss_blob_bind_f1</i>
Query	dataset bound to a blob	<i>ss_blob_bound_f</i>
Query	dataset bound to a blob	<i>ss_blob_bound_f1</i>
Change array element	datatype	<i>ss_array_target</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Inquire about array file	datatype	<i>ss_array_targeted</i>
Query the	datatype of a property or property list	<i>ss_prop_type</i>
Compares two	datatypes	<i>H5Tcmp</i>
Extra native	datatypes	<i>H5T_NATIVE</i>
Print an arbitrary	datum	<i>ss_val_dump</i>
Evaluate a	debug command	<i>ss_debug_s</i>
Parse	debug setup statements	<i>ss_debug_env</i>
Start	debugger for error	<i>ss_error</i>
	Debugging aid	<i>ss_pers_debug</i>
Enter an interactive	debugging loop	<i>ss_debug</i>
	Declare a file to be transient	<i>H5F_ACC_TRANSIENT</i>
	Decode persistent object links	<i>ss_pers_decode_cb</i>
	Dereference an object link	<i>ss_pers_deref</i>
	Destroy a property list	<i>ss_prop_dest</i>
	Destructor	<i>SS_PERS_DEST</i>
	Destructor	<i>ss_pers_dest</i>
	Determine if property exists	<i>ss_prop_has</i>
	Determine link equality	<i>SS_PERS_EQ</i>
	Determine link equality	<i>ss_pers_eq</i>
	Determine magicness	<i>SS_MAGIC_OK</i>
	Determine object equality	<i>SS_PERS_EQUAL</i>
	Determine object equality	<i>ss_pers_equal</i>
	Determines if scope is an open top-scope	<i>ss_scope_isopentop</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Maximum IndexSpec	dimensionality	<i>SS_MAX_INDEXDIMS</i>
Obtain pointer	direct to value	<i>ss_prop_buffer</i>
Change array	element datatype	<i>ss_array_target</i>
Query the number of	elements	<i>ss_array_nelmts</i>
	End functionality test	<i>SS_END_CHECKING</i>
	End functionality test	<i>SS_END_CHECKING_WITH</i>
	Enter an interactive debugging loop	<i>ss_debug</i>
Print information about a global file	entry	<i>ss_gfile_debug_one</i>
	Environment Variables	<i>SSLIB</i>
Determine link	equality	<i>SS_PERS_EQ</i>
Determine link	equality	<i>ss_pers_eq</i>
Determine object	equality	<i>SS_PERS_EQUAL</i>
Determine object	equality	<i>ss_pers_equal</i>
Start debugger for	error	<i>ss_error</i>
Minor	error numbers	<i>SS_MINOR</i>
	Evaluate a debug command	<i>ss_debug_s</i>
Asserts object runtime type and	existence	<i>SS_ASSERT_MEM</i>
Create a new property list from an	existing list	<i>ss_prop_dup</i>
Determine if property	exists	<i>ss_prop_has</i>
	Extend a blob	<i>ss_blob_extend</i>
	Extend a blob	<i>ss_blob_extend1</i>
Query blob	extent	<i>ss_blob_space</i>
	Extra native datatypes	<i>H5T_NATIVE</i>
Indicate functionality test	failure	<i>SS_FAILED</i>
Indicate functionality test	failure	<i>SS_FAILED_WHEN</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Close a	file	<i>ss_file_close</i>
Create a new	file	<i>ss_file_create</i>
Open or create a	file	<i>ss_file_open</i>
Read data from a	file	<i>ss_blob_read</i>
Read data from a	file	<i>ss_blob_read1</i>
Synchronize all scopes of a	file	<i>ss_file_synchronize</i>
Tests transient state of a	file	<i>ss_file_istransient</i>
Write pending data to	file	<i>ss_scope_flush</i>
Write pending data to	file	<i>ss_file_flush</i>
Mark	file as read-only	<i>ss_file_readonly</i>
Inquire about array	file datatype	<i>ss_array_targeted</i>
Print information about a global	file entry	<i>ss_gfile_debug_one</i>
Obtain	file for an object	<i>ss_pers_file</i>
Test	file open status	<i>ss_file_isopen</i>
Query	file synchronization state	<i>ss_file_synchronized</i>
Print global	file table	<i>ss_gfile_debug_all</i>
Declare a	file to be transient	<i>H5F_ACC_TRANSIENT</i>
Test	file writability	<i>ss_file_iswritable</i>
Obtain information about referenced	files	<i>ss_file_references</i>
Open many	files	<i>ss_file_openall</i>
Mark library as	finalized	<i>ss_zap</i>
	Find attributes for an object	<i>ss_attr_find</i>
	Find indirect indices for an object	<i>ss_table_indirect</i>
	Find objects in a scope	<i>SS_PERS_FIND</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Find objects in a scope	<i>ss_pers_find</i>
Global bit	flags	<i>SS</i>
Value comparison	flags	<i>ss_val_cmp_t</i>
Query a	floating point property	<i>ss_prop_get_f</i>
Change a	floating-point property value	<i>ss_prop_set_f</i>
	Flush pending data to HDF5	<i>ss_blob_flush</i>
	Free memory associated with the array	<i>ss_array_reset</i>
	Free memory associated with the string	<i>ss_string_reset</i>
Begin a	functionality test	<i>SS_CHECKING</i>
End	functionality test	<i>SS_END_CHECKING</i>
End	functionality test	<i>SS_END_CHECKING_WITH</i>
Indicate	functionality test failure	<i>SS_FAILED</i>
Indicate	functionality test failure	<i>SS_FAILED_WHEN</i>
Indicate	functionality test skipped	<i>SS_SKIPPED</i>
Indicate	functionality test skipped	<i>SS_SKIPPED_WHEN</i>
	Get a C string from a persistent string	<i>ss_string_get</i>
	Get two-phase I/O properties	<i>ss_blob_get_2pio</i>
	Global bit flags	<i>SS</i>
Print information about a	global file entry	<i>ss_gfile_debug_one</i>
Print	global file table	<i>ss_gfile_debug_all</i>
Flush pending data to	HDF5	<i>ss_blob_flush</i>
	HDF5 async callback	<i>ss_aio_hdf5_cb</i>
Renders	human readable numbers	<i>ss_bytes</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Initiate 2-phase	I/O	<code>ss_blob_synchronize</code>
Get two-phase	I/O properties	<code>ss_blob_get_2pio</code>
Set two-phase	I/O properties	<code>ss_blob_set_2pio</code>
Initialize the asynchronous	I/O subsystem	<code>ss_aio_init</code>
Terminate the asynchronous	I/O subsystem	<code>ss_aio_finalize</code>
Check	if link is null	<code>SS_PERS_ISNULL</code>
Determine	if property exists	<code>ss_prop_has</code>
Determines	if scope is an open top-scope	<code>ss_scope_isopentop</code>
Make a property list	immutable	<code>ss_prop_immutable</code>
Maximum	IndexSpec dimensionality	<code>SS_MAX_INDEXDIMS</code>
	Indicate functionality test failure	<code>SS_FAILED</code>
	Indicate functionality test failure	<code>SS_FAILED_WHEN</code>
	Indicate functionality test skipped	<code>SS_SKIPPED</code>
	Indicate functionality test skipped	<code>SS_SKIPPED_WHEN</code>
Find indirect	indices for an object	<code>ss_table_indirect</code>
Find	indirect indices for an object	<code>ss_table_indirect</code>
Print	information about a global file entry	<code>ss_gfile_debug_one</code>
Obtain	information about referenced files	<code>ss_file_references</code>
Sets persistent object to	initial state	<code>ss_pers_reset</code>
Test library	initialization state	<code>ss_initialized</code>
	Initialize the asynchronous I/O sub-system	<code>ss_aio_init</code>
	Initialize the library	<code>ss_init_func</code>
	Initialize the library	<code>ss_init</code>
	Initiate 2-phase I/O	<code>ss_blob_synchronize</code>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Initiate a write operation	<i>ss_aio_write</i>
	Inquire about array file datatype	<i>ss_array_targeted</i>
	Insert commas into an integer	<i>ss_insert_commas</i>
Insert commas into an	integer	<i>ss_insert_commas</i>
Query an	integer property	<i>ss_prop_get_i</i>
Query an unsigned	integer property	<i>ss_prop_get_u</i>
Change a signed	integer property value	<i>ss_prop_set_i</i>
Change an unsigned	integer property value	<i>ss_prop_set_u</i>
Enter an	interactive debugging loop	<i>ss_debug</i>
Insert commas	into an integer	<i>ss_insert_commas</i>
Obtain pointer	into string object	<i>ss_string_ptr</i>
Determines if scope	is an open top-scope	<i>ss_scope_isopentop</i>
Check if link	is null	<i>SS_PERS_ISNULL</i>
Change the size of a variable	length array	<i>ss_array_resize</i>
Query the	length of a persistent string	<i>ss_string_len</i>
Query the	length of a persistent string	<i>ss_string_memlen</i>
Compares two variable	length strings	<i>ss_string_cmp</i>
Initialize the	library	<i>ss_init_func</i>
Initialize the	library	<i>ss_init</i>
Terminate the	library	<i>ss_finalize</i>
Mark	library as finalized	<i>ss_zap</i>
Test	library initialization state	<i>ss_initialized</i>
Create an object	link	<i>ss_pers_refer</i>
Dereference an object	link	<i>ss_pers_deref</i>
Updates an object	link	<i>ss_pers_update</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Determine	link equality	<i>SS_PERS_EQ</i>
Determine	link equality	<i>ss_pers_eq</i>
Check if	link is null	<i>SS_PERS_ISNULL</i>
Change	link state	<i>ss_pers_state</i>
Decode persistent object	links	<i>ss_pers_decode_cb</i>
Persistent object	links (class 0x5af02000)	<i>SS_MAGIC_ss</i>
Add new property to a	list	<i>ss_prop_add</i>
Create a new property list from an existing	list	<i>ss_prop_dup</i>
Destroy a property	list	<i>ss_prop_dest</i>
Query the datatype of a property or property	list	<i>ss_prop_type</i>
Queries/sets property	list appendability	<i>ss_prop_appendable</i>
Create a new property	list from an existing list	<i>ss_prop_dup</i>
Create a new property	list from scratch	<i>ss_prop_new</i>
Make a property	list immutable	<i>ss_prop_immutable</i>
Queries/sets property	list modifiability	<i>ss_prop_modifiable</i>
Enter an interactive debugging	loop	<i>ss_debug</i>
Construct a	magic number	<i>SS_MAGIC_CONS</i>
Obtain	magic number class	<i>SS_MAGIC_CLASS</i>
Obtain	magic number for type	<i>SS_MAGIC</i>
Obtain	magic number from a pointer	<i>SS_MAGIC_OF</i>
Obtain	magic sequence number	<i>SS_MAGIC_SEQUENCE</i>
Determine	magicness	<i>SS_MAGIC_OK</i>
	Make a new object unique	<i>ss_pers_unique</i>
	Make a property list immutable	<i>ss_prop_immutable</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Make an object unique	<i>SS_PERS_UNIQUE</i>
Open	many files	<i>ss_file_openall</i>
	Mark file as read-only	<i>ss_file_readonly</i>
	Mark library as finalized	<i>ss_zap</i>
	Mark object as modified	<i>SS_PERS_MODIFIED</i>
	Mark object as modified	<i>ss_pers_modified</i>
Count	matching attributes	<i>ss_attr_count</i>
	Maximum IndexSpec dimensionality	<i>SS_MAX_INDEXDIMS</i>
Bind a blob to	memory	<i>ss_blob_bind_m</i>
Bind a blob to	memory	<i>ss_blob_bind_m1</i>
Free	memory associated with the array	<i>ss_array_reset</i>
Free	memory associated with the string	<i>ss_string_reset</i>
Query	memory bound to a blob	<i>ss_blob_bound_m</i>
Query	memory bound to a blob	<i>ss_blob_bound_m1</i>
Query attribute	metadata	<i>ss_attr_describe</i>
	Minor error numbers	<i>SS_MINOR</i>
	Miscellaneous (class 0x5af01000)	<i>SS_MAGIC_ss</i>
Queries/sets property list	modifiability	<i>ss_prop_modifiable</i>
Substring	modification	<i>ss_string_splice</i>
Mark object as	modified	<i>SS_PERS_MODIFIED</i>
Mark object as	modified	<i>ss_pers_modified</i>
Test whether an object can be	modified	<i>ss_pers_iswritable</i>
Tests whether scope can be	modified	<i>ss_scope_iswritable</i>
	Modify attribute type and size	<i>ss_attr_modify</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Modify part of an array	<i>ss_array_put</i>
Extra	native datatypes	<i>H5T_NATIVE</i>
Add a	new attribute to an object	<i>ss_attr_new</i>
Create a	new blob	<i>ss_blob_new</i>
Create a	new file	<i>ss_file_create</i>
Make a	new object unique	<i>ss_pers_unique</i>
Create a	new persistent object	<i>ss_pers_new</i>
Create a	new property list from an existing list	<i>ss_prop_dup</i>
Create a	new property list from scratch	<i>ss_prop_new</i>
Add	new property to a list	<i>ss_prop_add</i>
Check if link is	null	<i>SS_PERS_ISNULL</i>
Construct a magic	number	<i>SS_MAGIC_CONS</i>
Obtain magic sequence	number	<i>SS_MAGIC_SEQUENCE</i>
Obtain magic	number class	<i>SS_MAGIC_CLASS</i>
Obtain magic	number for type	<i>SS_MAGIC</i>
Obtain magic	number from a pointer	<i>SS_MAGIC_OF</i>
	Number of base quantities	<i>SS_MAX_BASEQS</i>
Query the	number of elements	<i>ss_array_nelmts</i>
Minor error	numbers	<i>SS_MINOR</i>
Renders human readable	numbers	<i>ss_bytes</i>
Find	objects in a scope	<i>SS_PERS_FIND</i>
Add a new attribute to an	object	<i>ss_attr_new</i>
Compute a checksum for a persistent	object	<i>ss_pers_cksum</i>
Copy an	object	<i>ss_pers_copy</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Create a new persistent	object	<i>ss_pers_new</i>
Find attributes for an	object	<i>ss_attr_find</i>
Find indirect indices for an	object	<i>ss_table_indirect</i>
Obtain file for an	object	<i>ss_pers_file</i>
Obtain pointer into string	object	<i>ss_string_ptr</i>
Obtain scope for an	object	<i>ss_pers_scope</i>
Obtain top scope for an	object	<i>ss_pers_topscope</i>
Mark	object as modified	<i>SS_PERS_MODIFIED</i>
Mark	object as modified	<i>ss_pers_modified</i>
Test whether an	object can be modified	<i>ss_pers_iswritable</i>
Determine	object equality	<i>SS_PERS_EQUAL</i>
Determine	object equality	<i>ss_pers_equal</i>
Create an	object link	<i>ss_pers_refer</i>
Dereference an	object link	<i>ss_pers_deref</i>
Updates an	object link	<i>ss_pers_update</i>
Decode persistent	object links	<i>ss_pers_decode_cb</i>
Persistent	object links (class 0x5af02000)	<i>SS_MAGIC_ss</i>
Attach an	object registry scope	<i>ss_file_registry</i>
Asserts	object runtime class	<i>SS_ASSERT_CLASS</i>
Asserts	object runtime type	<i>SS_ASSERT_TYPE</i>
Asserts	object runtime type and existence	<i>SS_ASSERT_MEM</i>
Sets persistent	object to initial state	<i>ss_pers_reset</i>
Make a new	object unique	<i>ss_pers_unique</i>
Make an	object unique	<i>SS_PERS_UNIQUE</i>
Compares two persistent	objects	<i>ss_pers_cmp</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Compares two persistent	objects	<i>ss_pers_cmp_</i>
Persistent	objects (class 0x5af03000)	<i>SS_MAGIC_ss</i>
Find	objects in a scope	<i>ss_pers_find</i>
	Obtain array value	<i>ss_array_get</i>
	Obtain attribute value	<i>ss_attr_get</i>
	Obtain file for an object	<i>ss_pers_file</i>
	Obtain information about referenced files	<i>ss_file_references</i>
	Obtain magic number class	<i>SS_MAGIC_CLASS</i>
	Obtain magic number for type	<i>SS_MAGIC</i>
	Obtain magic number from a pointer	<i>SS_MAGIC_OF</i>
	Obtain magic sequence number	<i>SS_MAGIC_SEQUENCE</i>
	Obtain pointer direct to value	<i>ss_prop_buffer</i>
	Obtain pointer into string object	<i>ss_string_ptr</i>
	Obtain scope for an object	<i>ss_pers_scope</i>
	Obtain top scope	<i>ss_file_topscope</i>
	Obtain top scope for an object	<i>ss_pers_topscope</i>
Append	one string to another	<i>ss_string_cat</i>
	Open many files	<i>ss_file_openall</i>
	Open or create a file	<i>ss_file_open</i>
Query scope	open status	<i>ss_scope_isopen</i>
Test file	open status	<i>ss_file_isopen</i>
Determines if scope is an	open top-scope	<i>ss_scope_isopentop</i>
	Opens a scope	<i>ss_scope_open</i>
Initiate a write	operation	<i>ss_aio_write</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Parse debug setup statements	<i>ss_debug_env</i>
Modify	part of an array	<i>ss_array_put</i>
Write	pending data to file	<i>ss_scope_flush</i>
Write	pending data to file	<i>ss_file_flush</i>
Flush	pending data to HDF5	<i>ss_blob_flush</i>
Compute a checksum for a	persistent object	<i>ss_pers_cksum</i>
Create a new	persistent object	<i>ss_pers_new</i>
Decode	persistent object links	<i>ss_pers_decode_cb</i>
	Persistent object links (class 0x5af02000)	<i>SS_MAGIC_ss</i>
Sets	persistent object to initial state	<i>ss_pers_reset</i>
Compares two	persistent objects	<i>ss_pers_cmp</i>
Compares two	persistent objects	<i>ss_pers_cmp_</i>
	Persistent objects (class 0x5af03000)	<i>SS_MAGIC_ss</i>
Get a C string from a	persistent string	<i>ss_string_get</i>
Query the length of a	persistent string	<i>ss_string_len</i>
Query the length of a	persistent string	<i>ss_string_memlen</i>
Store a C string in a	persistent string	<i>ss_string_set</i>
Compare	persistent string with C string	<i>ss_string_cmp_s</i>
Query a floating	point property	<i>ss_prop_get_f</i>
Obtain magic number from a	pointer	<i>SS_MAGIC_OF</i>
Obtain	pointer direct to value	<i>ss_prop_buffer</i>
Obtain	pointer into string object	<i>ss_string_ptr</i>
	Print an arbitrary datum	<i>ss_val_dump</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Print global file table	<i>ss_gfile_debug_all</i>
	Print information about a global file entry	<i>ss_gfile_debug_one</i>
Get two-phase I/O	properties	<i>ss_blob_get_2pio</i>
Set two-phase I/O	properties	<i>ss_blob_set_2pio</i>
Query a floating point	property	<i>ss_prop_get_f</i>
Query an integer	property	<i>ss_prop_get_i</i>
Query an unsigned integer	property	<i>ss_prop_get_u</i>
	Property constructor	<i>ss_prop_cons</i>
Determine if	property exists	<i>ss_prop_has</i>
Destroy a	property list	<i>ss_prop_dest</i>
Query the datatype of a property or	property list	<i>ss_prop_type</i>
Queries/sets	property list appendability	<i>ss_prop_appendable</i>
Create a new	property list from an existing list	<i>ss_prop_dup</i>
Create a new	property list from scratch	<i>ss_prop_new</i>
Make a	property list immutable	<i>ss_prop_immutable</i>
Queries/sets	property list modifiability	<i>ss_prop_modifiable</i>
Query the datatype of a	property or property list	<i>ss_prop_type</i>
Add new	property to a list	<i>ss_prop_add</i>
Change a	property value	<i>ss_prop_set</i>
Change a floating-point	property value	<i>ss_prop_set_f</i>
Change a signed integer	property value	<i>ss_prop_set_i</i>
Change an unsigned integer	property value	<i>ss_prop_set_u</i>
Query a	property value	<i>ss_prop_get</i>
Number of base	quantities	<i>SS_MAX_BASEQS</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
	Queries/sets property list appendability	<i>ss_prop_appendable</i>
	Queries/sets property list modifiability	<i>ss_prop_modifiable</i>
	Query a floating point property	<i>ss_prop_get_f</i>
	Query a property value	<i>ss_prop_get</i>
	Query an integer property	<i>ss_prop_get_i</i>
	Query an unsigned integer property	<i>ss_prop_get_u</i>
	Query attribute metadata	<i>ss_attr_describe</i>
	Query blob extent	<i>ss_blob_space</i>
	Query dataset bound to a blob	<i>ss_blob_bound_f</i>
	Query dataset bound to a blob	<i>ss_blob_bound_f1</i>
	Query file synchronization state	<i>ss_file_synchronized</i>
	Query memory bound to a blob	<i>ss_blob_bound_m</i>
	Query memory bound to a blob	<i>ss_blob_bound_m1</i>
	Query scope open status	<i>ss_scope_isopen</i>
	Query scope synchronization state	<i>ss_scope_synchronized</i>
	Query the datatype of a property or property list	<i>ss_prop_type</i>
	Query the length of a persistent string	<i>ss_string_len</i>
	Query the length of a persistent string	<i>ss_string_memlen</i>
	Query the number of elements	<i>ss_array_nelmts</i>
	Query the scope communicator	<i>ss_scope_comm</i>
	Read data from a file	<i>ss_blob_read</i>
	Read data from a file	<i>ss_blob_read1</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Mark file as	read-only	<i>ss_file_readonly</i>
Renders human	readable numbers	<i>ss_bytes</i>
Obtain information about	referenced files	<i>ss_file_references</i>
Attach an object	registry scope	<i>ss_file_registry</i>
	Renders human readable numbers	<i>ss_bytes</i>
Returns current status of a	request	<i>ss_aio_error</i>
Block until	requests complete	<i>ss_aio_suspend</i>
Weakly	reset a string	<i>ss_string_realloc</i>
	Returns current status of a request	<i>ss_aio_error</i>
Wildcard SIL	role	<i>SAF_SROLE_ANY</i>
SIL	roles	<i>ss_silrole_t</i>
Asserts object	runtime class	<i>SS_ASSERT_CLASS</i>
Asserts object	runtime type	<i>SS_ASSERT_TYPE</i>
Asserts object	runtime type and existence	<i>SS_ASSERT_MEM</i>
Attach an object registry	scope	<i>ss_file_registry</i>
Closes a	scope	<i>ss_scope_close</i>
Find objects in a	scope	<i>SS_PERS_FIND</i>
Find objects in a	scope	<i>ss_pers_find</i>
Obtain top	scope	<i>ss_file_topscope</i>
Opens a	scope	<i>ss_scope_open</i>
Synchronize a	scope	<i>ss_scope_synchronize</i>
Tests transient state of a	scope	<i>ss_scope_istransient</i>
Tests whether	scope can be modified	<i>ss_scope_ismodifiable</i>
Query the	scope communicator	<i>ss_scope_comm</i>
Obtain	scope for an object	<i>ss_pers_scope</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Obtain top	scope for an object	<i>ss_pers_topscope</i>
Determines if	scope is an open top-scope	<i>ss_scope_isopentop</i>
Query	scope open status	<i>ss_scope_isopen</i>
Query	scope synchronization state	<i>ss_scope_synchronized</i>
Synchronize all	scopes of a file	<i>ss_file_synchronize</i>
Create a new property list from	scratch	<i>ss_prop_new</i>
Obtain magic	sequence number	<i>SS_MAGIC_SEQUENCE</i>
	Set two-phase I/O properties	<i>ss_blob_set_2pio</i>
	Sets persistent object to initial state	<i>ss_pers_reset</i>
Parse debug	setup statements	<i>ss_debug_env</i>
Change a	signed integer property value	<i>ss_prop_set_i</i>
Wildcard	SIL role	<i>SAF_SROLE_ANY</i>
	SIL roles	<i>ss_silrole_t</i>
Modify attribute type and	size	<i>ss_attr_modify</i>
Change the	size of a variable length array	<i>ss_array_resize</i>
Indicate functionality test	skipped	<i>SS_SKIPPED</i>
Indicate functionality test	skipped	<i>SS_SKIPPED_WHEN</i>
	Start debugger for error	<i>ss_error</i>
Change link	state	<i>ss_pers_state</i>
Query file synchronization	state	<i>ss_file_synchronized</i>
Query scope synchronization	state	<i>ss_scope_synchronized</i>
Sets persistent object to initial	state	<i>ss_pers_reset</i>
Test library initialization	state	<i>ss_initialized</i>
Tests transient	state of a file	<i>ss_file_istransient</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Tests transient	state of a scope	<i>ss_scope_istransient</i>
Parse debug setup	statements	<i>ss_debug_env</i>
Query scope open	status	<i>ss_scope_isopen</i>
Test file open	status	<i>ss_file_isopen</i>
Returns current	status of a request	<i>ss_aio_error</i>
Create	storage for a blob	<i>ss_blob_mkstorage</i>
	Store a byte array in a string	<i>ss_string_memset</i>
	Store a C string in a persistent string	<i>ss_string_set</i>
Compare persistent string with C	string	<i>ss_string_cmp_s</i>
Free memory associated with the	string	<i>ss_string_reset</i>
Get a C string from a persistent	string	<i>ss_string_get</i>
Query the length of a persistent	string	<i>ss_string_len</i>
Query the length of a persistent	string	<i>ss_string_memlen</i>
Store a byte array in a	string	<i>ss_string_memset</i>
Store a C string in a persistent	string	<i>ss_string_set</i>
Weakly reset a	string	<i>ss_string_realloc</i>
Get a C	string from a persistent string	<i>ss_string_get</i>
Store a C	string in a persistent string	<i>ss_string_set</i>
Obtain pointer into	string object	<i>ss_string_ptr</i>
Append one	string to another	<i>ss_string_cat</i>
Compare persistent	string with C string	<i>ss_string_cmp_s</i>
Compares two variable length	strings	<i>ss_string_cmp</i>
	Substring modification	<i>ss_string_splice</i>
Initialize the asynchronous I/O	subsystem	<i>ss_aio_init</i>
Terminate the asynchronous I/O	subsystem	<i>ss_aio_finalize</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Query file	synchronization state	<i>ss_file_synchronized</i>
Query scope	synchronization state	<i>ss_scope_synchronized</i>
	Synchronize a scope	<i>ss_scope_synchronize</i>
	Synchronize all scopes of a file	<i>ss_file_synchronize</i>
Print global file	table	<i>ss_gfile_debug_all</i>
	Terminate the asynchronous I/O subsystem	<i>ss_aio_finalize</i>
	Terminate the library	<i>ss_finalize</i>
Begin a functionality	test	<i>SS_CHECKING</i>
End functionality	test	<i>SS_END_CHECKING</i>
End functionality	test	<i>SS_END_CHECKING_WITH</i>
Indicate functionality	test failure	<i>SS_FAILED</i>
Indicate functionality	test failure	<i>SS_FAILED_WHEN</i>
	Test file open status	<i>ss_file_isopen</i>
	Test file writability	<i>ss_file_iswritable</i>
	Test library initialization state	<i>ss_initialized</i>
Indicate functionality	test skipped	<i>SS_SKIPPED</i>
Indicate functionality	test skipped	<i>SS_SKIPPED_WHEN</i>
	Test whether an object can be modified	<i>ss_pers_iswritable</i>
	Tests transient state of a file	<i>ss_file_istransient</i>
	Tests transient state of a scope	<i>ss_scope_istransient</i>
	Tests whether scope can be modified	<i>ss_scope_iswritable</i>
Obtain	top scope	<i>ss_file_topscope</i>
Obtain	top scope for an object	<i>ss_pers_topscope</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Determines if scope is an open	top-scope	<i>ss_scope_isopentop</i>
Declare a file to be	transient	<i>H5F_ACC_TRANSIENT</i>
Tests	transient state of a file	<i>ss_file_istransient</i>
Tests	transient state of a scope	<i>ss_scope_istransient</i>
Compares	two datatypes	<i>H5Tcmp</i>
Compares	two persistent objects	<i>ss_pers_cmp</i>
Compares	two persistent objects	<i>ss_pers_cmp_</i>
Compares	two variable length strings	<i>ss_string_cmp</i>
Get	two-phase I/O properties	<i>ss_blob_get_2pio</i>
Set	two-phase I/O properties	<i>ss_blob_set_2pio</i>
Asserts object runtime	type	<i>SS_ASSERT_TYPE</i>
Obtain magic number for	type	<i>SS_MAGIC</i>
Asserts object runtime	type and existence	<i>SS_ASSERT_MEM</i>
Modify attribute	type and size	<i>ss_attr_modify</i>
Make a new object	unique	<i>ss_pers_unique</i>
Make an object	unique	<i>SS_PERS_UNIQUE</i>
Query an	unsigned integer property	<i>ss_prop_get_u</i>
Change an	unsigned integer property value	<i>ss_prop_set_u</i>
Block	until requests complete	<i>ss_aio_suspend</i>
	Updates an object link	<i>ss_pers_update</i>
Change a floating-point property	value	<i>ss_prop_set_f</i>
Change a property	value	<i>ss_prop_set</i>
Change a signed integer property	value	<i>ss_prop_set_i</i>
Change an attribute	value	<i>ss_attr_put</i>
Change an unsigned integer property	value	<i>ss_prop_set_u</i>

Continued on next page

Table 1.3 – continued from previous page

Concept	Key	Reference
Obtain array	value	<i>ss_array_get</i>
Obtain attribute	value	<i>ss_attr_get</i>
Obtain pointer direct to	value	<i>ss_prop_buffer</i>
Query a property	value	<i>ss_prop_get</i>
	Value comparison flags	<i>ss_val_cmp_t</i>
Change the size of a	variable length array	<i>ss_array_resize</i>
Compares two	variable length strings	<i>ss_string_cmp</i>
Environment	Variables	<i>SSLIB</i>
	Weakly reset a string	<i>ss_string_realloc</i>
Test	whether an object can be modified	<i>ss_pers_iswritable</i>
Tests	whether scope can be modified	<i>ss_scope_iswritable</i>
	Wildcard SIL role	<i>SAF_SROLE_ANY</i>
Test file	writability	<i>ss_file_iswritable</i>
	Write data to a blob	<i>ss_blob_write</i>
	Write data to a blob	<i>ss_blob_write1</i>
Initiate a	write operation	<i>ss_aio_write</i>
	Write pending data to file	<i>ss_scope_flush</i>
	Write pending data to file	<i>ss_file_flush</i>

Symbols

`_saf_convert` (*C function*), 180
`_saf_find_parent_field` (*C function*), 108
`_saf_gen_stdtypes` (*C function*), 51
`_saf_is_primitive_type` (*C function*), 181
`_saf_strdup` (*C function*), 152

C

`CloseDatabase` (*C function*), 198

G

`GetAddDelSequence` (*C function*), 186

H

`H5F_ACC_TRANSIENT` (*C variable*), 290
`H5Tcmp` (*C function*), 289

M

`main` (*C function*), 188, 190, 197, 202, 203, 206, 207
`make_base_space` (*C function*), 189, 190, 203
`make_coord_field` (*C function*), 190
`make_coord_field_dofs` (*C function*), 191
`make_direct_coord_field` (*C function*), 189
`make_direct_temperature_field` (*C function*), 189
`make_displacement_field` (*C function*), 203
`make_distribution_factors_on_ss2_field` (*C function*), 203
`make_global_coord_field` (*C function*), 204
`make_indirect_coord_field` (*C function*), 189
`make_indirect_temperature_field` (*C function*), 190
`make_init_suite` (*C function*), 204
`make_mesh_connectivity` (*C function*), 191
`make_pressure_on_ss1_field` (*C function*), 204
`make_scalar_field` (*C function*), 192
`make_scalar_field_dofs` (*C function*), 192
`make_stress_field` (*C function*), 192
`make_stress_field_dofs` (*C function*), 193

`make_stress_on_cell_1_field` (*C function*), 204
`make_temperature_on_cell_2_field` (*C function*), 205
`make_temperature_on_ns1_field` (*C function*), 205
`make_time_base_field` (*C function*), 205
`make_time_suite` (*C function*), 205

O

`OpenDatabase` (*C function*), 187, 194, 201

R

`ReadBackElementHistory` (*C function*), 195

S

`SAF_1DC` (*C macro*), 156
`SAF_1DF` (*C macro*), 157
`SAF_2DC` (*C macro*), 157
`SAF_2DF` (*C macro*), 157
`SAF_3DC` (*C macro*), 157
`SAF_3DF` (*C macro*), 158
`saf_allgather_handles` (*C function*), 153
`SAF_ASSERT_DISABLE` (*C variable*), 43
`SAF_BARRIER` (*C macro*), 150
`SAF_BOUNDARY` (*C macro*), 83
`SAF_CATCH` (*C variable*), 48
`SAF_CATCH_ALL` (*C variable*), 48
`SAF_CATCH_ERR` (*C macro*), 49
`saf_close_database` (*C function*), 64
`SAF_COMMON` (*C macro*), 83
`SAF_CONSTANT` (*C macro*), 107
`SAF_CORDER` (*C macro*), 159
`saf_createProps_database` (*C function*), 66
`saf_createProps_lib` (*C function*), 53
`saf_data_has_been_written_to_comp_field` (*C function*), 109
`saf_data_has_been_written_to_field` (*C function*), 109

`saf_declare_algebraic` (*C function*), 169
`saf_declare_alternate_indexspec` (*C function*), 172
`saf_declare_basis` (*C function*), 176
`saf_declare_category` (*C function*), 76
`saf_declare_collection` (*C function*), 79
`saf_declare_coords` (*C function*), 110
`saf_declare_default_coords` (*C function*), 110
`saf_declare_evaluation` (*C function*), 181
`saf_declare_field` (*C function*), 111
`saf_declare_field_tmpl` (*C function*), 101
`saf_declare_quantity` (*C function*), 137
`saf_declare_relrep` (*C function*), 183
`saf_declare_role` (*C function*), 178
`saf_declare_set` (*C function*), 70
`saf_declare_state_group` (*C function*), 127
`saf_declare_state_tmpl` (*C function*), 124
`saf_declare_subset_relation` (*C function*), 84
`saf_declare_suite` (*C function*), 132
`saf_declare_topo_relation` (*C function*), 93
`saf_declare_unit` (*C function*), 142
`SAF_DECOMP` (*C macro*), 107
`SAF_DEFAULT_DBPROPS` (*C variable*), 66
`SAF_DEFAULT_LIBPROPS` (*C variable*), 160
`saf_describe_algebraic` (*C function*), 170
`saf_describe_alternate_indexspec` (*C function*), 173
`saf_describe_basis` (*C function*), 177
`saf_describe_category` (*C function*), 76
`saf_describe_collection` (*C function*), 80
`saf_describe_evaluation` (*C function*), 181
`saf_describe_field` (*C function*), 114
`saf_describe_field_tmpl` (*C function*), 102
`saf_describe_quantity` (*C function*), 138
`saf_describe_relrep` (*C function*), 184
`saf_describe_role` (*C function*), 179
`saf_describe_set` (*C function*), 71
`saf_describe_state_group` (*C function*), 129
`saf_describe_state_tmpl` (*C function*), 125
`saf_describe_subset_relation` (*C function*), 87
`saf_describe_suite` (*C function*), 133
`saf_describe_topo_relation` (*C function*), 95
`saf_describe_unit` (*C function*), 142
`saf_divide_quantity` (*C macro*), 139
`saf_divide_unit` (*C macro*), 143
`SAF_EMBEDBND` (*C macro*), 84
`SAF_EQUIV` (*C macro*), 151
`SAF_ERROR_REPORTING` (*C variable*), 44
`saf_error_str` (*C function*), 50
`saf_extend_collection` (*C function*), 81
`saf_final` (*C function*), 51
`saf_find_algebraics` (*C function*), 170
`saf_find_alternate_indexspecs` (*C function*), 174
`saf_find_bases` (*C function*), 177
`saf_find_categories` (*C function*), 77
`saf_find_collections` (*C function*), 82
`saf_find_coords` (*C function*), 115
`saf_find_default_coords` (*C function*), 116
`saf_find_evaluations` (*C function*), 182
`saf_find_field_tmpls` (*C function*), 103
`saf_find_fields` (*C function*), 116
`saf_find_matching_sets` (*C function*), 72
`saf_find_one_algebraic` (*C function*), 171
`saf_find_one_basis` (*C function*), 178
`saf_find_one_evaluation` (*C function*), 183
`saf_find_one_quantity` (*C function*), 139
`saf_find_one_relrep` (*C function*), 184
`saf_find_one_role` (*C function*), 179
`saf_find_one_unit` (*C function*), 143
`saf_find_quantities` (*C function*), 139
`saf_find_relreps` (*C function*), 184
`saf_find_roles` (*C function*), 179
`saf_find_sets` (*C function*), 73
`saf_find_state_groups` (*C function*), 129
`saf_find_state_tmpl` (*C function*), 125
`saf_find_subset_relations` (*C function*), 88
`saf_find_suites` (*C function*), 134
`saf_find_topo_relations` (*C function*), 95
`saf_find_unit_not_applicable` (*C function*), 144
`saf_find_units` (*C function*), 144
`SAF_FORDER` (*C macro*), 161
`saf_freeInfo_path` (*C function*), 59
`saf_freeProps_database` (*C function*), 66
`saf_freeProps_lib` (*C function*), 53
`SAF_GENERAL` (*C macro*), 84
`saf_get_attribute` (*C function*), 148
`saf_get_cat_att` (*C function*), 78
`saf_get_count_and_type_for_field` (*C function*), 117
`saf_get_count_and_type_for_subset_relation` (*C function*), 89
`saf_get_count_and_type_for_topo_relation` (*C function*), 96
`saf_get_field_att` (*C function*), 118
`saf_get_field_tmpl_att` (*C function*), 104
`saf_get_set_att` (*C function*), 75
`saf_get_state_grp_att` (*C function*), 130
`saf_get_state_tmpl_att` (*C function*), 126
`saf_get_suite_att` (*C function*), 134
`saf_getInfo_errmsg` (*C function*), 59
`saf_getInfo_hdfversion` (*C function*), 59
`saf_getInfo_isHDFfile` (*C function*), 60
`saf_getInfo_isSAFdatabase` (*C function*), 60
`saf_getInfo_libversion` (*C function*), 61

- saf_getInfo_mpiversion (*C function*), 61
 saf_getInfo_permissions (*C function*), 62
 saf_getInfo_staterror (*C function*), 62
 saf_init (*C macro*), 52
 saf_is_self_stored_field (*C function*), 118
 saf_is_self_stored_topo_relation (*C function*), 97
 saf_log_unit (*C function*), 145
 saf_multiply_quantity (*C function*), 140
 saf_multiply_unit (*C function*), 146
 SAF_NA_INDEXSPEC (*C variable*), 162
 SAF_NELMTS (*C macro*), 151
 SAF_NODAL (*C macro*), 107
 SAF_NOT_APPLICABLE_INT (*C variable*), 163
 SAF_NOT_IMPL (*C variable*), 163
 SAF_NOT_SET_DB (*C variable*), 64
 SAF_NULL_FIELD (*C macro*), 108
 SAF_NULL_FTmpl (*C macro*), 100
 SAF_NULL_REL (*C macro*), 100
 SAF_NULL_SET (*C macro*), 69
 SAF_NULL_STATE_GRP (*C macro*), 127
 SAF_NULL_STmpl (*C macro*), 124
 SAF_NULL_SUITE (*C macro*), 132
 saf_offset_unit (*C function*), 146
 saf_open_database (*C function*), 64
 SAF_PARALLEL_VAR (*C variable*), 154
 SAF_POSTCOND_DISABLE (*C variable*), 45
 SAF_PRECOND_DISABLE (*C variable*), 45
 saf_put_attribute (*C function*), 149
 saf_put_cat_att (*C function*), 78
 saf_put_field_att (*C function*), 119
 saf_put_field_tmpl_att (*C function*), 104
 saf_put_set_att (*C function*), 75
 saf_put_state_grp_att (*C function*), 130
 saf_put_state_tmpl_att (*C function*), 126
 saf_put_suite_att (*C function*), 134
 SAF_QAMOUNT (*C variable*), 135
 SAF_QCURRENT (*C variable*), 136
 SAF_QLENGTH (*C variable*), 136
 SAF_QLIGHT (*C variable*), 136
 SAF_QMASS (*C variable*), 136
 SAF_QNAME (*C macro*), 137
 SAF_QTEMP (*C variable*), 137
 SAF_QTIME (*C variable*), 137
 saf_quantify_unit (*C function*), 147
 SAF_RANK (*C macro*), 151
 saf_read_alternate_indexspec (*C function*), 175
 saf_read_field (*C function*), 119
 saf_read_state (*C function*), 130
 saf_read_subset_relation (*C function*), 90
 saf_read_topo_relation (*C function*), 98
 saf_readInfo_path (*C function*), 62
 SAF_REGISTRIES (*C variable*), 46
 SAF_REGISTRY_SAVE (*C variable*), 46
 saf_same_collections (*C function*), 83
 SAF_SELF (*C macro*), 75
 saf_setProps_Clobber (*C function*), 67
 saf_setProps_DbComm (*C function*), 67
 saf_setProps_DontAbort (*C function*), 54
 saf_setProps_ErrFunc (*C function*), 54
 saf_setProps_ErrorLogging (*C function*), 55
 saf_setProps_ErrorMode (*C function*), 55
 saf_setProps_LibComm (*C function*), 56
 saf_setProps_MemoryResident (*C function*), 68
 saf_setProps_ReadOnly (*C function*), 68
 saf_setProps_Registry (*C function*), 56
 saf_setProps_StrMode (*C function*), 57
 saf_setProps_StrPoolSize (*C function*), 58
 SAF_SIZE (*C macro*), 151
 SAF_SROLE_ANY (*C variable*), 291
 saf_target_field (*C function*), 121
 saf_target_subset_relation (*C function*), 91
 saf_target_topo_relation (*C function*), 98
 SAF_TRACING (*C variable*), 47
 SAF_TRY_BEGIN (*C variable*), 49
 SAF_TRY_END (*C variable*), 49
 SAF_UNIVERSE (*C macro*), 69
 saf_update_database (*C function*), 65
 saf_use_written_subset_relation (*C function*), 91
 SAF_VALID (*C macro*), 152
 SAF_VERSION_ANNOT (*C variable*), 154
 SAF_VERSION_MAJOR (*C variable*), 154
 SAF_VERSION_MINOR (*C variable*), 155
 SAF_VERSION_RELEASE (*C variable*), 155
 saf_version_string (*C function*), 156
 SAF_VERSION_VAR (*C variable*), 155
 SAF_WHOLE_FIELD (*C variable*), 108
 saf_write_alternate_indexspec (*C function*), 175
 saf_write_field (*C function*), 122
 saf_write_state (*C function*), 131
 saf_write_subset_relation (*C function*), 92
 saf_write_topo_relation (*C function*), 99
 SAF_XOR (*C macro*), 152
 SAF_ZONAL (*C macro*), 108
 ss_array_get (*C function*), 266
 ss_array_nelmts (*C function*), 268
 ss_array_put (*C function*), 267
 ss_array_reset (*C function*), 267
 ss_array_resize (*C function*), 266
 ss_array_target (*C function*), 265
 ss_array_targeted (*C function*), 266
 SS_ASSERT_CLASS (*C macro*), 228
 SS_ASSERT_MEM (*C macro*), 228
 SS_ASSERT_TYPE (*C macro*), 227
 ss_attr_count (*C function*), 284

`ss_attr_describe` (*C function*), 287
`ss_attr_find` (*C function*), 284
`ss_attr_get` (*C function*), 285
`ss_attr_modify` (*C function*), 286
`ss_attr_new` (*C function*), 283
`ss_attr_put` (*C function*), 285
`ss_bytes` (*C function*), 225
`SS_CHECKING` (*C macro*), 228
`ss_debug` (*C function*), 292
`ss_debug_env` (*C function*), 293
`ss_debug_s` (*C function*), 295
`SS_END_CHECKING` (*C variable*), 230
`SS_END_CHECKING_WITH` (*C macro*), 230
`ss_error` (*C function*), 224
`SS_FAILED` (*C variable*), 229
`SS_FAILED_WHEN` (*C macro*), 229
`ss_file_close` (*C function*), 275
`ss_file_create` (*C function*), 271
`ss_file_flush` (*C function*), 274
`ss_file_isopen` (*C function*), 271
`ss_file_istransient` (*C function*), 272
`ss_file_iswritable` (*C function*), 272
`ss_file_open` (*C function*), 269
`ss_file_openall` (*C function*), 270
`ss_file_readonly` (*C function*), 273
`ss_file_references` (*C function*), 270
`ss_file_registry` (*C function*), 275
`ss_file_synchronize` (*C function*), 273
`ss_file_synchronized` (*C function*), 274
`ss_file_topscope` (*C function*), 276
`ss_finalize` (*C function*), 223
`ss_gfile_debug_all` (*C function*), 277
`ss_gfile_debug_one` (*C function*), 277
`ss_init` (*C macro*), 225
`ss_init_func` (*C function*), 217
`ss_initialized` (*C function*), 223
`ss_insert_commas` (*C function*), 225
`SS_MAGIC` (*C macro*), 233
`SS_MAGIC_CLASS` (*C macro*), 233
`SS_MAGIC_CONS` (*C macro*), 234
`SS_MAGIC_OF` (*C macro*), 234
`SS_MAGIC_OK` (*C macro*), 233
`SS_MAGIC_SEQUENCE` (*C macro*), 234
`SS_MAX_BASEQS` (*C variable*), 290
`SS_MAX_INDEXDIMS` (*C variable*), 290
`ss_pers_cksum` (*C function*), 252
`ss_pers_cmp` (*C function*), 250
`ss_pers_cmp_` (*C function*), 251
`ss_pers_copy` (*C function*), 244
`SS_PERS_COPY` (*C macro*), 257
`ss_pers_debug` (*C function*), 255
`ss_pers_decode_cb` (*C function*), 255
`ss_pers_deref` (*C function*), 246
`ss_pers_dest` (*C function*), 245
`SS_PERS_DEST` (*C macro*), 256
`ss_pers_eq` (*C function*), 248
`SS_PERS_EQ` (*C macro*), 257
`ss_pers_equal` (*C function*), 249
`SS_PERS_EQUAL` (*C macro*), 258
`ss_pers_file` (*C function*), 247
`ss_pers_find` (*C function*), 252
`SS_PERS_FIND` (*C macro*), 257
`SS_PERS_ISNULL` (*C macro*), 258
`ss_pers_iswritable` (*C function*), 248
`ss_pers_modified` (*C function*), 254
`SS_PERS_MODIFIED` (*C macro*), 258
`ss_pers_new` (*C function*), 243
`SS_PERS_NEW` (*C macro*), 256
`ss_pers_refer` (*C function*), 246
`ss_pers_reset` (*C function*), 245
`ss_pers_scope` (*C function*), 247
`ss_pers_state` (*C function*), 249
`ss_pers_topscope` (*C function*), 248
`ss_pers_unique` (*C function*), 255
`SS_PERS_UNIQUE` (*C macro*), 258
`ss_pers_update` (*C function*), 246
`ss_prop_add` (*C function*), 236
`ss_prop_appendable` (*C function*), 242
`ss_prop_buffer` (*C function*), 241
`ss_prop_cons` (*C function*), 236
`ss_prop_dest` (*C function*), 236
`ss_prop_dup` (*C function*), 235
`ss_prop_get` (*C function*), 239
`ss_prop_get_f` (*C function*), 240
`ss_prop_get_i` (*C function*), 240
`ss_prop_get_u` (*C function*), 240
`ss_prop_has` (*C function*), 237
`ss_prop_immutable` (*C function*), 243
`ss_prop_modifiable` (*C function*), 242
`ss_prop_new` (*C function*), 235
`ss_prop_set` (*C function*), 237
`ss_prop_set_f` (*C function*), 239
`ss_prop_set_i` (*C function*), 238
`ss_prop_set_u` (*C function*), 238
`ss_prop_type` (*C function*), 241
`ss_scope_close` (*C function*), 279
`ss_scope_comm` (*C function*), 282
`ss_scope_flush` (*C function*), 282
`ss_scope_isopen` (*C function*), 279
`ss_scope_isopentop` (*C function*), 280
`ss_scope_istransient` (*C function*), 280
`ss_scope_iswritable` (*C function*), 280
`ss_scope_open` (*C function*), 278
`ss_scope_synchronize` (*C function*), 281
`ss_scope_synchronized` (*C function*), 281
`SS_SKIPPED` (*C variable*), 229
`SS_SKIPPED_WHEN` (*C macro*), 230
`ss_string_cat` (*C function*), 263

ss_string_cmp (*C function*), 262
ss_string_cmp_s (*C function*), 263
ss_string_get (*C function*), 260
ss_string_len (*C function*), 264
ss_string_memlen (*C function*), 264
ss_string_memset (*C function*), 261
ss_string_ptr (*C function*), 260
ss_string_realloc (*C function*), 262
ss_string_reset (*C function*), 262
ss_string_set (*C function*), 261
ss_string_splice (*C function*), 263
ss_table_indirect (*C function*), 259
ss_val_dump (*C function*), 287
ss_zap (*C function*), 224

U

UpdateDatabase (*C function*), 198

W

WriteCurrentMesh (*C function*), 187, 196, 202