SSC Reference Manual

Aron P. Dobos

December 31, 2012

Abstract

The SSC (SAM Simulation Core) software library implements the underlying renewable energy system modeling calculation engine utilitized by the popular desktop System Advisor Model (SAM) tool. SSC provides a simple and programmer-friendly application programming interface (API) that allows developers to directly integrate SAM calculations into other tools. The API includes mechanisms to set input variable values, run simulations, and retrieve calculated outputs. The library is provided to users as precompiled binary dynamic libraries for Windows, Mac OSX, and Linux, with the bare minimum system library dependencies. While the native API language is ISO-standard C, language bindings for MATLAB, and Python are included. The API and model implementations are thread-safe and reentrant, allowing the software to be efficiently utilized in a parallel computing environment.

This document describes the software architecture of SSC, introduces the SSC Development Environment tool, explains how to set up simulations from code, query the library for variable information, and provides examples and “case studies" in various programming languages to ease the adoption of SSC into other systems.

Note. The reader is assumed to have familiarity with the C programming language, as well as some level of proficiency using the System Advisor Model (SAM) tool.

Contents

1 Overview

The SSC software library provides a standard interface through which complex engineering and economic models of renewable energy systems can be configured and invoked. The System Advisor Model (SAM) desktop application is essentially a user-friendly front-end for SSC, and uses SSC to perform all system performance and financial calculations. It is assumed that the reader is well versed in SAM, its suite of models, typical inputs and outputs, and general usage.

1.1 Framework Description

SSC is designed as a general simulation framework that is model agnostic. For example, there is no specific `pvwatts(...)` function call in the API that accepts model parameters and returns a calculated result. Rather, a generic mechanism is provided for sending a model data, running the model, and retrieving results from it.

The standard way to invoke a model in SSC involves three steps:

1. A generic data container is populated by the user with named variables and their values. These variables are inputs to the model.
2. A particular model (like `pvwattsv1`) is selected to *process* the data container. The model may run successfully using the input variables defined by the user, or fail with error messages. If the model succeeds, it populates the data container with the calculated output variables.
3. Calculated outputs are retrieved from the data container, or error messages are retrieved.

Essentially, this sequence is no different than calling a function in a programming language. Step 1 can be thought of as passing parameters to the function, step 2 as running the function, and step 3 as getting the calculated result returned. In SSC, the "function" is called a *module*, and the data container is simply *data*. The *module* operates on the *data* and transforms it by calculating new variables and adding them to the dataset. In the API, these are represented by the `ssc_module_t` and `ssc_data_t` types, which will be covered in detail later.

1.2 Modeling Systems

Different modules are available in SSC for each type of simulation possible in SAM. A sample of modules is shown in Table 1.

To perform a complete technoeconomic systems analysis, multiple compute modules can be called in succession. For example, when the PVWatts+Residential configuration is selected in SAM, several compute modules are called in to perform the calculation, as listed below.

1. `pvwattsv1`: The system parameters are used to calculate the hourly PV system performance.
2. `annualoutput`: The availability and degradation factors are applied to the annual energy output for the duration of the specified analysis period.
3. `utilityrate`: The value of the generated energy is calculated given the specifics of the complex utility rate structure, and the first year's hourly energy value is reported, along with the annual total

Module	Description
pvwattsv1	Implements the NREL PVWatts model for PV performance
pvsamv1	Implements the component-based PV models in SAM
wfreader	Reads standard format weather data files (TM2, TM3, EPW, SMW)
irradproc	General purpose irradiance processor for calculating POA
windpower	Implements the NREL wind turbine and simple wind farm model
cashloan	Implements residential and commercial cashflow economic model
ippppa	Commercial PPA and Independent Power Producer (IPP) financial models
annualoutput	Calculates degraded out-years system output for the analysis period
utilityrate	Calculates the value of energy using complex utility rate structures

Table 1: Some commonly used SSC modules

value of energy at each year of the analysis period.

4. cashloan: Given the amount of the energy produced each year and the dollar value of this energy, this module calculates the cost of energy, net present value, and other metrics of the system based on specified system cost, incentives, operation and maintainance costs, taxes, and loan parameters.

In some cases, the output variables of one module have the right units and number of data values to feed directly into the next module in a sequence, but this is not enforced. For example, the pvwattsv1 module has an output variable ac which is the hourly energy in kWh produced by the system. However, the annualoutput module requires an input with the name energy_net_hourly. It is up to the user to ensure that variables are translated appropriately between the outputs of one module and the inputs of another, including any units conversions that may be necessary. Luckily, SSC provides exhaustive documentation of all of the input variables required by a module and all of the outputs that will be calculated. Working with input and output variables will be the topic of a subsequent chapter.

2 A Basic Example: PVWatts

In this section, we will write a simple command-line C program to calculate the monthly energy production of a 1 kW PV system at a particular location. The complete source code for this example program is included with the SSC SDK in the file example1_pvwatts.c.

2.1 Program Skeleton

The SSC API is defined a C header called sscapi.h, and this file must be included in your program before any SSC functions can be called. We will assume that the weather file is specified as a command line argument, and is one of the types that the pvwattsv1 can read (TM2, TM3, EPW, SMW). The skeleton of our program might look like the listing below.

```
#include <stdio.h>
#include "sscapi.h"

int main(int argc, char *argv[])
```

```
{
    if ( argc < 2 )
    {
        printf("usage: pvwatts.exe <weather-file>\n");
        return -1;
    }

    // run PVWatts simulation for the specified weather file

    return 0;
}
```

2.2 Setting up data inputs

Now we will fill in the comment in the middle of the code step-by-step. Referring to the three step process outlined in Section 1, the first task is to create a data container. This is done with the `ssc_data_create()` function call, which returns an `ssc_data_t` type. If for some reason the function call fails due to the system having run out of memory, `NULL` will be returned. It is important to check the result to make sure the data container was created successfully.

```
ssc_data_t data = ssc_data_create();
if ( data == NULL )
{
    printf("error: out of memory.\n");
    return -1;
}
```

Next, we assign the input variables required by the `pvwattsv1` module. In §**??**, we will show how to obtain information about the data types, names, labels, and units of variables. For now, it suffices to say that SSC supports data as numbers, text strings, arrays, matrices, and tables.

```
ssc_data_set_string( data, "file_name", argv[1] ); // set the weather file name
ssc_data_set_number( data, "system_size", 1.0f );  // system size of 1 kW DC
ssc_data_set_number( data, "derate", 0.77f );      // system derate
ssc_data_set_number( data, "track_mode", 0 );      // fixed tilt system
ssc_data_set_number( data, "tilt", 20 );           // 20 degree tilt
ssc_data_set_number( data, "azimuth", 180 );       // south facing (180 degrees)
```

At this point, we have a data container with six variables that are the input parameters to the module that we wish to run.

2.3 Simulating and retrieving outputs

Next, an instance of the module itself must be created, per below. The name of the desired module must be passed to the `ssc_module_create()` function. If the specified module name is not recognized, or the system is out of memory, the function may return `NULL`.

```
ssc_module_t module = ssc_module_create( "pvwattsv1" );
if ( NULL == module )
{
    printf("error: could not create 'pvwattsv1' module.\n");
    ssc_data_free( data );
    return -1;
}
```

Next, the simulation must be run. If all the required input variables have been defined in the data container and have appropriate data types and values, the simulation should finish successfully. Otherwise, an error will be flagged.

```
if ( ssc_module_exec( module, data ) == 0 )
{
    printf("error during simulation.\n");
    ssc_module_free( module );
    ssc_data_free( data );
    return -1;
}
```

By default, the `ssc_module_exec()` function prints any log or error messages to standard output on the console. There is a more complex way to run a simulation that can reroute messages somewhere else, for example if a program wanted to pop up a message box or collect all messages and return them to the sure. This will be discussed in later sections.

Assuming that the simulation succeeded, it is now time to extract the calculated results. Modules may calculate hundreds of outputs, but for this example, we simply are interested in the total AC energy produced by the 1 kW PV system. As it turns out, `pvwattsv1` only provides the hourly data, so it is up to us to sum up the hourly values. The code snippet below shows how to query the data object for the `ac` variable and aggregate it.

```
double ac_total = 0;
int len = 0;
ssc_number_t *ac = ssc_data_get_array( data, "ac", &len );
if ( ac != NULL )
{
    int i;
    for ( i=0; i<len; i++ )
        ac_total += ac[i];
    printf("ac: %lg kWh\n", ac_total*0.001 );
}
else
{
    printf("variable 'ac' not found.\n");
}
```

2.4 Cleaning up

We're now finished simulating the PV system and retrieving the results. To avoid filling up the computer's memory with unneeded data objects, it is important to free the memory when it is no longer needed. To this end, the `ssc_module_free()` and `ssc_data_free()` functions are provided. Note that after calling one of these functions, the `module` and `data` variables are invalidated and cannot be used unless reassigned to another or a new data object or module.

```
ssc_module_free( module );
ssc_data_free( data );
```

2.5 Compiling and running

Using the MinGW compiler toolchain on Windows, it is very straightforward to compile and run this example. Simply issue the command below at the Windows command prompt (`cmd.exe` from Start/Run), assuming that the `sscapi.h` header file and 32-bit `ssc.dll` dynamic library is in the same folder as the source file. The complete source code for this example is included in the SSC SDK. This example was tested with MinGW gcc version 4.6.2.

```
c:\> gcc example1_pvwatts.c ssc.dll -o pvwatts.exe
```

To run the program, specify a weather file on the command-line. Here, we use TMY2 data for Daggett, CA, which is included in the SDK as *daggett.tm2*.

```
c:\> pvwatts.exe daggett.tm2
ac: 1468.54 kWh
```

If all goes well, the total annual AC kWh for the system is printed on the console.

2.6 Some additional comments

The `ssc_data_t` and `ssc_module_t` data types are opaque references to internally defined data structures. The only proper way to interact with variables of these types is using the defined SSC function calls. As listed in the `sscapi.h` header file, both are typedef'd as `void*`.

3 Data Variables

Simulation model inputs and outputs are stored in a data container of type `ssc_data_t`, which is simply a collection of named variables. Internally, the data structure is an unordered map (hash table), permitted very fast lookup of variables by name. Every input and output variable in SSC is designated a name, a *variable type*, and a *data type*, along with other meta data (labels, units, etc).

The variable name is mechanism by which data are identified by SSC, and the user is required to supply input variables with names that SSC has predefined. Names can contain letters, numbers, and underscores. Although names can be defined with upper and lower case letters, SSC does not distinguish between them internally: hence `Beam_Irradiance` is the same variable as `beam_irradiance`.

3.1 Variable Types

The *variable type* identifies each variable as an input, output, or in-out variable. In the SSC API, these values are defined by the constants reproduced below.

```
#define SSC_INPUT 1
#define SSC_OUTPUT 2
#define SSC_INOUT 3
```

Input variables are required to be set by the user before a module is called, while output variables are calculated by the module and assigned to the data container when the module has finished running. In-out variables are supplied by the user before the model runs, and the model transforms the value in some fashion.

Note that from the perspective of the data container, inputs and outputs are stored without distinction in an equivalent manner. Thus, it is impossible to tell simply from the contents of a data container whether a variable was an input or an output - it is just a pile of data. It is only the simulation modules that specify the variable type.

3.2 Data Types

Every variable in SSC is assigned a particular data type, and each simulation module to specifies the data type of each variable required as input, as well as the data type of each output. SSC can work with numbers, text strings, one-dimensional arrays, two-dimensional matrices, and tables. The data type constants are listed below.

```
#define SSC_INVALID 0
#define SSC_STRING 1
#define SSC_NUMBER 2
#define SSC_ARRAY 3
#define SSC_MATRIX 4
#define SSC_TABLE 5
```

All numbers are stored using the `ssc_number_t` data type. By default, this is a typedef of the 32-bit floating point C data type (`float`). The purpose of using `float` instead of the 64-bit `double` is to save memory. While a simulation module may perform all of its calculations internally using 64-bit precision floating point, the inputs and results are transferred into and out of SSC using the smaller data type.

Arrays (1-D) and matrices (2-D) store numbers only - there is no provision for arrays or matrices of text strings, or arrays of tables. Arrays and matrices are optimized for efficient storage and transfer of large amounts of numerical data. For example, a photovoltaic system simulation at 1 second timesteps for a whole year would produce 525,600 data points, and potentially several such data vectors might be reported by a single simulation model. This fact underscores the reasoning to use the 32-bit floating point data type: just 20 vectors of 525,600 values each would require 42 megabytes of computer memory.

SSC does not provision separate storage for integers and floating point values - all numbers are stored internally as floating point. However, a module may specific a numeric input with the *integer* constraint which is checked automatically before the module is run. Constraints will be discussed later.

Text strings (SSC_STRING) are stored as 8-bit characters. SSC does not support multi-byte or wide-character string representations, and all variable names and labels use only the 7-bit ASCII Latin alphabet. Consequently, text is stored as null ('\0') terminated char* C strings. Weather file names are common input variables that use the SSC_STRING data type.

The table (SSC_TABLE) data type is the only hierarchical data type in SSC. Essentially, it allows a simulation module to receive or return outputs in a structured format with named fields. A table is nothing more than a named variable that is itself a data container, which can store any number of named variables of any SSC data type. Currently, most SSC modules do not make heavy use of tables, but they are fully implemented and supported for future complex modules that may be added.

3.3 Variable Documentation

As stated before, each module defines all of the input variables it requires and output variables it produces. Since SSC is a model simulation framework that contains within it numerous calculation modules that each may have hundreds of variables, it is impractical to separate to the documentation of variables from their definitions. Otherwise, the documentation would tend to be consistently out of date as modules are updated and new ones are added.

Field	Description
Variable type	SSC_INPUT, SSC_OUTPUT, SSC_INOUT
Data type	SSC_NUMBER, SSC_STRING, SSC_ARRAY, SSC_MATRIX, SSC_TABLE
Name	Variable name (case insensitive)
Label	A description of the variable's purpose
Units	The units of the numerical value(s)
Meta	Additional information. Could specify encoding of values, see below.
Group	General category of variable, e.g. "Weather", "System Input"
Required	Specifies whether the variable must be assigned a value. See §3.4
Constraints	Constraints on the values or number of values. See §3.4
UI Hints	Suggestions on how to display this variable. Currently unused.

Table 2: Variable information provided by SSC

Consequently, SSC provides functions in the API to return information about all of the variables defined by a module. Each variable specifies the information shown in Table 2.

To obtain information for a variable, a module is queried using the ssc_module_var_info() function, which returns a ssc_info_t reference. The ssc_module_var_info() function is called repeatedly with an index variable that is incremented by the user to cycle through all of the variables defined by the module. An example is shown below, assuming that the variable module has been successfully created for a particular simulation module (e.g., see §2.3). Querying SSC for available modules is discussed in §4.1.

```
int i=0;
ssc_info_t p_inf = NULL;
while ( p_inf = ssc_module_var_info( module, i++ ) )
{
    // var_type: SSC_INPUT, SSC_OUTPUT, SSC_INOUT
```

```
    int var_type = ssc_info_var_type( p_inf );

    // data_type: SSC_STRING, SSC_NUMBER, SSC_ARRAY, SSC_MATRIX, SSC_TABLE
    int data_type = ssc_info_data_type( p_inf );

    const char *name = ssc_info_name( p_inf );
    const char *label = ssc_info_label( p_inf );
    const char *units = ssc_info_units( p_inf );
    const char *meta = ssc_info_meta( p_inf );
    const char *group = ssc_info_group( p_inf );
    const char *required = ssc_info_required( p_inf );
    const char *constraints = ssc_info_constrants( p_inf );

    // here, you can print all of this data to text file
    // or present it in another way to the user

    printf( "%s %s (%s) %s\n", name, label, units, meta );
}
```

The interactive SSCdev tool provided with the SSC library automatically generates all of this documentation for all modules in an interactive GUI spreadsheet-style interface. This utility will be discussed in more detail in §6.1.

3.4 Default Values and Constraints

Sometimes there are so many input variables for a module that SSC assigns reasonable default values for some of the inputs if they are not explicitly specified by the user. In other cases, some inputs may only have relevance for advanced users, and so a standard default value is provided that is recommended for the majority of simulations. If a default value is provided, it is specified in the *required* field of the variable information table.

If the *required* field contains ‘‘`*`’’, this means that there is no default value and the user must specify one in all cases. If the *required* field begins with the character ‘‘`?`’’, it means that the variable is optional, and that SSC will use the value assigned by the user if it is provided. The manner in which an optional variable affects the calculation is specific to each module, and so cannot be expanded upon here.

A default value is provided by SSC if the *required* field appears as ‘‘`?=<value>`’’, where `<value>` may be a number, a text string, or comma-separated array, depending on the data type of the variable. In effect, this means that the variable always be given a value before simulation, but the user need not specify it. For example, the `pvwattsv1` module specifies `rotlim` (tracker rotation limit) variable’s *required* field as ‘‘`?=45.0`’’, meaning that unless the user specifies a different value, a rotation limit of 45 degrees is used.

In addition to providing some default values, SSC performs some data input validation before running a simulation using the *constraints* field. This field may optionally define any number of flags recognized by SSC that may limit the valid range of a numeric input variable, or define the required length of an input array. A selection of the flags used by SSC is listed in Table 3. Note that these flags are defined internally

by the modules, and are not editable by users. Also, in the case that the *constraints* field is empty, that does not necessarily imply that any values are acceptable - the module writer may have inadvertently omitted a constraint flag, and may instead manually check each variable when the module runs.

Flag	Description
MIN=<value>	Specifies the minimum permissible value for a number
MAX=<value>	Specifies the maximum permissible value for a number
BOOLEAN	Requires the number to be equal to 0 (false) or 1 (true)
INTEGER	Requires the number to be an integer value
POSITIVE	Requires the number to have a positive, non-zero value
LOCAL_FILE	Requires the text string to be a local file on disk that is readable and exists

Table 3: Some constraints recognized by SSC

Constraints can be combined to further restrict the value of a number. For example, in pvwattsv1, the track_mode variable's constraints field is ''MIN=0,MAX=3,INTEGER''. This specification limits the possible values of the variable to 0, 1, 2, or 3, which are the numeric values associated with a fixed, 1-axis, 2-axis, or azimuth-axis tracking PV system.

3.5 Manipulating Data Containers

As was shown in the initial PVWatts example, a data container is created and freed using the ssc_data_first() and ssc_data_next() functions. Here, some additional functions for working with data containers are explored.

The ssc_data_unassign() function removes a variable from the container, releasing any memory associated with it. To erase all of the variables in a container and leave it empty (essentially resetting it), use the ssc_data_clear() function. For example:

```
ssc_data_unassign( data, "tilt" ); // remove the 'tilt' variable

ssc_data_clear( data ); // reset the container by erasing all the variables
```

To find out the names of all variables in a container, functions ssc_data_first() and ssc_data_next() are used. These functions return the name of a variable, or NULL if there are no more variables in the container. The ssc_data_query() function returns the data type of the specified variable, which is useful for subsequently retrieving its value. The example below prints out the text of all the string variables in a data container:

```
const char *name = ssc_data_first( data );
while( name != 0 )
{
    // do something, like query the data type
    int type = ssc_data_query( data, name );

    if ( type == SSC_STRING )
    {
        const char *text = ssc_data_get_string( data, name );
```

```
        printf("string variable %s: %s\n", name, text );
    }

    name = ssc_data_next( data );
}
```

See the API reference in §8 for detailed documentation of each function's parameters and return values.

3.6 Assigning and Retrieving Values

This section discusses the particulars of assigning and retrieving variable values. Note that assigning a variable that already exists in the container causes the old value to be erased.

3.6.1 Numbers

Numbers are transferred and stored using the `ssc_number_t` data type. To assign a value:

```
ssc_data_set_number( data, "track_mode", 2 ); // integer value
ssc_data_set_number( data, "tilt", 30.5f ); // floating point value

ssc_number_t azimuth = 182.3f;
ssc_data_set_number( data, "azimuth", azimuth ); // from a variable
```

To retrieve a numeric value, it must be declared first, since the retrieval function returns true or false. False is returned if the specified variable does not exist in the data container.

```
ssc_number_t value;
if( ssc_data_get_number( data, "tilt", &value ) )
    printf( "tilt = %f\n", value );
else
    printf( "not found\n" );
```

3.6.2 Arrays

Arrays are passed to SSC using standard C arrays (pointers), along with the length of the array. Examples:

```
ssc_number_t monthdays[12] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
ssc_data_set_array( data, "nday", monthdays, 12 );

// dynamically allocate an array of size len
ssc_number_t *xx = (ssc_number_t*) malloc( sizeof(ssc_number_t)*len );
for( i=0; i<len; i++ ) xx[i] = i*i;

ssc_data_set_array( data, "isquared", xx, len );
```

```
free( xx ); // SSC does not take ownership, so free any arrays here.
```

> **Note:** When assigning an array, SSC makes an internal copy, and does not take ownership of the array passed in. Thus it is up to the user to deallocate any memory created in the client code as appropriate.

Array data is retrieved from SSC using the `ssc_data_get_array()` function. The length of the retrieved array is returned in the length reference parameter. If the requested variable is not assigned, or is not an `SSC_ARRAY`, `NULL` is returned. Example:

```
int len = 0;
const ssc_number_t *array = ssc_data_get_array( data, "xx", &len );

if( array != NULL && len > 0 )
{
    printf("len: %d, first item=%f\n", len, array[0] );
}
```

> **Note:** The `ssc_data_get_array()` returns a reference to internally managed memory: do not deallocate or otherwise change the array returned by it. This is done to improve efficiency when working with large data vectors.

3.6.3 Matrices

Matrices are two dimensional arrays of numbers. In SSC, matrices are stored as one contiguous array, in row-major order. Like arrays, they consist of `ssc_number_t` data and are passed to SSC using standard C arrays, along with the number of rows and columns.

Row-major ordering means that the two dimensional array is flattened into a single dimensional vector. In a 4x3 matrix, the flattened vector will have 12 values in which indices [0..3] represent the first row, [4..7] the second row, and [8..11] the third row. Example:

```
ssc_number_t mat[12] = { 4,4,5,1,
                         6,7,2,5,
                         1,6,2,3 };

ssc_data_set_matrix( data, "mat1", mat, 4, 3 );
```

> **Note:** SSC creates an internal copy of the matrix passed to `ssc_data_set_matrix()`. If the matrix was allocated dyamically, it is up to the user written client code to deallocate it properly: SSC does not take ownership.

Matrices are returned in the same format. If the requested variable does not exist or is not a matrix, `NULL` is returned. Example:

```
int nrows, ncols;
ssc_number_t *mat = ssc_data_get_matrix( data, "mat1", &nrows, &ncols );

if( mat != NULL && nrows == 4 && ncols == 3 )
```

```
{
    int r, c, index = 0;
    for( r=0; r<nrows; r++ )
        for( c=0; c<ncols; c++ )
            printf( "mat[%d,%d]=%f\n", r, c, mat[index++] );
}
```

> **Note:** The `ssc_data_get_matrix()` returns a reference to internally managed memory: do not deallocate or otherwise change the array returned by it.

3.6.4 Strings

SSC supports null-terminated ASCII text strings as a data type. Setting a string value is straightforward. SSC creates an internal copy of the string data. Examples:

```
ssc_data_set_string( data, "weather_file", "c:/Data/Weather/TX Abilene.tm2" );

const char *location = "Hawaii";
ssc_data_set_string( data, "location", location );
```

Retrieving strings is similarly straightforward:

```
const char *location = ssc_data_get_string( data, "location" );
if( location != NULL )
    printf("location = %s\n", location );
```

> **Note:** Do not free or modify the pointer returned by `ssc_data_get_string()`. A reference to an internally managed character byte array is returned.

3.6.5 Tables

Tables provide a way to communicate data to SSC in a hierarchical structure format. This data type is provisioned for future simulation modules, but current ones do not use this data type. Nonetheless, the SSCdev utility (§6) fully supports tables. Before a table can be assigned, it must be created and filled with variables. A table is simply a normal data container. Example:

```
ssc_data_t table = ssc_data_create(); // create an empty table
ssc_data_set_string( table, "first", "peter" );
ssc_data_set_string( table, "last", "jones" );
ssc_data_set_number( table, "age", 24 );

ssc_data_set_table( data, "person", table ); // assign the table

ssc_data_free( table ); // free the table after it is assigned
```

Since a table is a just an `ssc_data_t` object, it can store any SSC data type, including another table. This structure allows for nested tables, as potentially needed by a future complicated simulation module.

> **Note:** SSC creates an internal copy of the table and all of its variables. As such, the table should be deallocated after it is assigned, per the example above.

Retrieving a table is a straightforward operation:

```
ssc_data_t table = ssc_data_get_table( data, "person" );
if( table != NULL )
{
    printf( "person.first = %s\n", ssc_data_get_string( table, "first" ) );
    printf( "person.last = %s\n", ssc_data_get_string( table, "last" ) );
}
// do not free the 'table' variable: it is internal to SSC
```

> **Note:** The `ssc_data_get_table()` function returns an internal reference to a data container. Do not deallocate or otherwise modify the returned reference.

Note that the `ssc_data_first()` and `ssc_data_next()` functions described in §3.5 can be used to determine all of the fields of a table.

4 Simulation Modules

The SSC framework exposes several different simulation modules that calculate outputs given inputs. Each module can run independently of all the others, provided that all the input variables that are required are provided by the user. The System Advisor Model (SAM) tool often runs several SSC modules in succession to simulate the whole system performance and economic evaluation. In between successive calls to different SSC modules, SAM may change the names or scale the output variables of one module to set it up with appropriate inputs for the next one, as explained in §1.2.

4.1 Querying SSC for Available Modules

SSC provides a programming interface by which the library can be queried about all of the modules contained within it. The example below prints all of the available modules to the console along with their version numbers and descriptions:

```
ssc_entry_t entry;
int index = 0;
while( entry = ssc_module_entry( index++ ) )
{
    const char *module_name = ssc_entry_name( entry );
    const char *description = ssc_entry_desc( entry );
    int version = ssc_entry_version( entry );

    printf( "Module %s, version %d: %s\n", module_name, version, description );
}
```

4.2 Simple Method #1

A simulation module can be invoked in one of several ways. The simplest way to call a module is to use the `ssc_module_exec_simple()` function. This method prints any warnings or errors to the console, and simply returns true or false. The PVWatts example from earlier could be written as:

```
ssc_data_t data = ssc_data_create();
ssc_data_set_string( data, "file_name", "TX Abilene.tm2" );
ssc_data_set_number( data, "system_size", 4 );
ssc_data_set_number( data, "derate", 0.77f );
ssc_data_set_number( data, "track_mode", 0 );
ssc_data_set_number( data, "tilt", 20 );
ssc_data_set_number( data, "azimuth", 180 );

if( ssc_module_exec_simple( "pvwattsv1", data ) )
{
    int i, len;
    ssc_number_t ac_sum = 0;
    ssc_number_t *ac = ssc_data_get_array( data, "ac", &len );
    for( i=0;i<len;i++ ) ac_sum += ac[i];
    printf( "success, ac: %f Wh\n", ac_sum );

}
else
    printf( "an error occured\n" );

ssc_data_free( data );
```

Note that there is no need to explicitly create an `ssc_module_t` object using `ssc_module_create()`; the function will return false if the module name specified could not be found or could not be created.

The `ssc_module_exec_simple()` function is thread-safe, provided that the module called is itself thread-safe. It is intended that all SSC modules are implemented in a thread-safe manner.

4.3 Simple Method #2

A similar function to run a module exists that returns to the caller the first error message encountered. This variation of *exec* function is not thread-safe, as indicated by its name. Example:

```
// ... set up a data container with inputs ...

const char *error = ssc_module_exec_simple_nothread( "pvwattsv1", data );
if( error == NULL )
    printf("success!\n");
else
    printf( "module failed with error: %s\n", error );
```

In this case, a `NULL` return value indicates that no error occurred.

4.4 Running with Progress Updates, Warnings, and Errors

The last (most complex) way to run a module is to provide a special handler that can reroute errors and warning messages, as well as provide progress updates to the controlling software. To accomplish this, the client code must provide a *handler* (aka *callback*) function to SSC. The handler function must have the signature below.

```
ssc_bool_t (*handler)( ssc_module_t module,
        ssc_handler_t handle,
        int action,
        float f0,
        float f1,
        const char *s0,
        const char *s1,
        void *user_data );
```

> **Note:** This documentation is specific to the C language API for SSC handler functions. Not all programming language interfaces included with SSC (e.g. MATLAB, Python, VBA) support the custom handler mechanism described here. However, log messages (notices, warnings, errors) can still be retrieved after a simulation is completed using the `ssc_module_log()` function (see §4.5, although real-time progress updates cannot be issued.

Supposing that such a function named `my_ssc_handler_function()` has been defined, used the `ssc_module_exec_with_handl` function to run the module. You may pass an a generic data pointer to the function also, so that within the context of the handler the messages or progress updates can be routed accordingly.

```
void *my_data = <some user data to send to the callback, or NULL>;

if ( ssc_module_exec_with_handler( module, data, my_ssc_handler_function, my_data ) )
    printf("success!\n");
else
    printf("fail!\n");
```

The code below represents the default handler built into SSC. Note that the handler function must return a boolean (0/1) value to indicate to SSC whether to continue simulating or not. This provides the option to abort the simulation if a user has clicked a cancel button in the meantime, or similar. The various parameters are used to convey SSC status information to the handler, and have different meanings (and data) depending on the `action_type` parameter.

```
static ssc_bool_t default_internal_handler( ssc_module_t p_mod, ssc_handler_t p_handler,
    int action_type, float f0, float f1,
    const char *s0, const char *s1,
    void *user_data )
{
    if (action_type == SSC_LOG)
    {
        // print log messages to console
        std::cout << "Log ";
```

```
        switch( (int)f0 ) // determines type
        {
        case SSC_NOTICE:
            std::cout << "Notice: " << s0 << " time " << f1 << std::endl;
            break;
        case SSC_WARNING:
            std::cout << "Warning: " << s0 << " time " << f1 << std::endl;
            break;
        case SSC_ERROR:
            std::cout << "Error: " << s0 << " time " << f1 << std::endl;
            break;
        default:
            std::cout << "Unknown: " << f0 << " time " << f1 << std::endl;
            break;
        }
        return 1;
    }
    else if (action_type == SSC_UPDATE)
    {
        // print status update to console
        std::cout << "Progress " << f0 << "%:" << s1 << " time " << f1 << std::endl;
        return 1; // return 0 to abort simulation as needed.
    }
    else
        return 0;
}
```

As noted in previous sections, the default handler simply prints warnings, errors, and simulation progress updates to the standard output stream on the console. To implement a custom handler, it is recommended to copy the structure of the handler above, but to replace the output to the console with specialized handling for the messages and progress. For example, in a graphical user interface program, the `user_data` pointer could be used to pass in a pointer to a dialog class containing a progress bar and a text box. The messages could be sent to the text box, and the progress bar updated appropriately. The custom SSC handler function implemented in the graphical SSCdev tool (based on the excellent wxWidgets GUI library) is reproduced below, showing how messages and progess can be reported to the user.

```
static ssc_bool_t my_handler( ssc_module_t p_mod, ssc_handler_t p_handler, int action,
    float f0, float f1, const char *s0, const char *s1, void *user_data )
{
    wxProgressDialog *dlg = (wxProgressDialog*) user_data;
    if (action == SSC_LOG)
    {
        wxString msg;

        switch( (int)f0 )
        {
        case SSC_NOTICE: msg << "Notice: " << s0 << " time " << f1; break;
```

```
        case SSC_WARNING: msg << "Warning: " << s0 << " time " << f1; break;
        case SSC_ERROR: msg << "Error: " << s0 << " time " << f1; break;
        default: msg << "Unknown: " << f0 << " time " << f1; break;
        }

        // post the SSC message on the application-wide log window
        app_frame->Log(msg);

        return 1;
    }
    else if (action == SSC_UPDATE)
    {
        // update progress dialog with percentage complete
        dlg->Update( (int) f0, s0 );
        wxGetApp().Yield(true);

        return 1; // return 0 to abort simulation as needed.
    }
    else
        return 0;
}
```

A third action type called `SSC_EXECUTE` can be found in the SSC header file. This is an advanced capability that is not described in this user guide, and is intended to be removed in the future from the SSC API.

4.5 Retrieving Messages

During simulation, a module may generate any number of (hopefully) informative messages about warnings or errors encountered. The custom handler approach allows these messages to be reported to the user as the simulation progresses, but they are also stored in the module for retrieval later. The `ssc_module_log()` function allows the client code to retrieve all the messages, regardless of which execute function was used to run the module. Example:

```
const char *text;
int type;
float time;
int index = 0;
while( (text = ssc_module_log( module, index++, &type, &time )) )
{
    switch( type ) // determines type
    {
    case SSC_NOTICE:
        std::cout << "Notice: " << text << " time " << time << std::endl;
        break;
    case SSC_WARNING:
        std::cout << "Warning: " << text << " time " << time << std::endl;
```

```
        break;
    case SSC_ERROR:
        std::cout << "Error: " << text << " time " << time << std::endl;
        break;
    default:
        std::cout << "Unknown: " << text << " time " << time << std::endl;
        break;
    }
}
```

In essence, the `ssc_module_log()` is called repeatedly with an increasing index number until the function returns `NULL`, indicating that there are no more messages. Along with the text of the message, information about the type (notice, warning, error) and the time that it was issued by the module are reported.

> **Note:** Do not free the C string returned by `ssc_module_log()`. It is a reference to internally managed memory.

5 Version Information

Functions are included in SSC to return version information and details about the particular build. For example:

```
int version = ssc_version();
const char *build = ssc_build_info();

printf( "ssc version %d: %s\n", version, build );
```

On Windows, this code may print:

```
ssc version 22: Windows 32 bit Visual C++ Nov 13 2012 18:44:14
```

On Linux, a newer version of SSC may report something like:

```
ssc version 24: Unix 64 bit GNU/C++ Dec  5 2012 11:42:51
```

6 Using the SSCdev Tool

6.1 Looking up Variables

7 Language Interfaces

7.1 Native C/C++

7.2 MATLAB

7.3 Visual Basic for Applications (VBA)

7.4 Java via Native Interface

7.5 C# and .NET

7.6 PHP

7.7 Python

7.8 Ruby-on-Rails

8 C API Reference

8.1 C:/Users/adobos/Projects/ssc/ssc/sscapi.h File Reference

SSC: SAM Simulation Core.

Typedefs

- typedef void * ssc_data_t
- typedef float ssc_number_t
- typedef int ssc_bool_t
- typedef void * ssc_entry_t
- typedef void * ssc_module_t
- typedef void * ssc_info_t
- typedef void * ssc_handler_t

Functions

- SSCEXPORT int ssc_version ()
- SSCEXPORT const char * ssc_build_info ()
- SSCEXPORT ssc_data_t ssc_data_create ()
- SSCEXPORT void ssc_data_free (ssc_data_t p_data)
- SSCEXPORT void ssc_data_clear (ssc_data_t p_data)
- SSCEXPORT void ssc_data_unassign (ssc_data_t p_data, const char *name)
- SSCEXPORT int ssc_data_query (ssc_data_t p_data, const char *name)
- SSCEXPORT const char * ssc_data_first (ssc_data_t p_data)
- SSCEXPORT const char * ssc_data_next (ssc_data_t p_data)
- SSCEXPORT void ssc_data_set_string (ssc_data_t p_data, const char *name, const char *value)
- SSCEXPORT void ssc_data_set_number (ssc_data_t p_data, const char *name, ssc_number_t value)
- SSCEXPORT void ssc_data_set_array (ssc_data_t p_data, const char *name, ssc_number_t *pvalues, int length)
- SSCEXPORT void ssc_data_set_matrix (ssc_data_t p_data, const char *name, ssc_number_t *pvalues, int nrows, int ncols)
- SSCEXPORT void ssc_data_set_table (ssc_data_t p_data, const char *name, ssc_data_t table)
- SSCEXPORT const char * ssc_data_get_string (ssc_data_t p_data, const char *name)
- SSCEXPORT ssc_bool_t ssc_data_get_number (ssc_data_t p_data, const char *name, ssc_number_t *value)
- SSCEXPORT ssc_number_t * ssc_data_get_array (ssc_data_t p_data, const char *name, int *length)
- SSCEXPORT ssc_number_t * ssc_data_get_matrix (ssc_data_t p_data, const char *name, int *nrows, int *ncols)
- SSCEXPORT ssc_data_t ssc_data_get_table (ssc_data_t p_data, const char *name)

- SSCEXPORT ssc_entry_t ssc_module_entry (int index)
- SSCEXPORT const char * ssc_entry_name (ssc_entry_t p_entry)
- SSCEXPORT const char * ssc_entry_description (ssc_entry_t p_entry)
- SSCEXPORT int ssc_entry_version (ssc_entry_t p_entry)
- SSCEXPORT ssc_module_t ssc_module_create (const char *name)
- SSCEXPORT void ssc_module_free (ssc_module_t p_mod)
- SSCEXPORT const ssc_info_t ssc_module_var_info (ssc_module_t p_mod, int index)
- SSCEXPORT int ssc_info_var_type (ssc_info_t p_inf)
- SSCEXPORT int ssc_info_data_type (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_name (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_label (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_units (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_meta (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_group (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_required (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_constraints (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_uihint (ssc_info_t p_inf)
- SSCEXPORT ssc_bool_t ssc_module_exec_simple (const char *name, ssc_data_t p_data)
- SSCEXPORT const char * ssc_module_exec_simple_nothread (const char *name, ssc_data_t p_data)
- SSCEXPORT ssc_bool_t ssc_module_exec (ssc_module_t p_mod, ssc_data_t p_data)
- SSCEXPORT ssc_bool_t ssc_module_exec_with_handler (ssc_module_t p_mod, ssc_data_t p_data, ssc_bool_t(*pf_handler)(ssc_module_t, ssc_handler_t, int action, float f0, float f1, const char *s0, const char *s1, void *user_data), void *pf_user_data)
- SSCEXPORT void ssc_module_extproc_output (ssc_handler_t p_mod, const char *output_line)
- SSCEXPORT const char * ssc_module_log (ssc_module_t p_mod, int index, int *item_type, float *time)
- SSCEXPORT void **__ssc_segfault** ()

8.1.1 Detailed Description

SSC: SAM Simulation Core. A general purpose simulation input/output framework. Cross-platform (Windows/-MacOSX/Unix) and is 32 and 64-bit compatible.

Note

Be sure to use the correct library for your operating platform: ssc32 or ssc64. Opaque pointer types will be 4-byte pointer on 32-bit architectures, and 8-byte pointer on 64-bit architectures.
Shared libraries have the .dll file extension on Windows, .dylib on MacOSX, and .so on Linux/Unix.

Copyright

2012 National Renewable Energy Laboratory

Authors

Aron Dobos, Steven Janzou

8.1.2 Typedef Documentation

8.1.2.1 typedef int ssc_bool_t

The boolean type used internally in SSC.

Zero values represent false; non-zero represents true.

8.1.2.2 typedef void* ssc_data_t

An opaque reference to a structure that holds a collection of variables.

This structure can contain any number of variables referenced by name, and can hold strings, numbers, arrays, and matrices. Matrices are stored in row-major order, where the array size is nrows*ncols, and the array index is calculated by r*ncols+c.

An ssc_data_t object holds all input and output variables for a simulation. It does not distinguish between input, output, and input variables - that is handled at the model context level.

For convenience, a ssc_data_t can be written to a file on disk, and later retrieved.

8.1.2.3 typedef void* ssc_entry_t

Returns information of all computation modules built into ssc. Example:

```
int i=0;
ssc_entry_t p_entry;
while( p_entry = ssc_module_entry(i++) )
{
    printf("Compute Module '%s': \n", ssc_entry_name(p_entry),
      ssc_entry_description(p_entry));
}
```

The opaque data structure that stores information about a compute module.

8.1.2.4 typedef void* ssc_handler_t

An opaque pointer for transferring external executable output back to SSC

8.1.2.5 typedef void* ssc_info_t

An opaque reference to variable information. A compute module defines its input/output variables.

8.1.2.6 typedef void* ssc_module_t

An opaque reference to a computation module.

Note

A computation module performs a transformation on a ssc_data_t. It usually is used to calculate output variables given a set of input variables, but it can also be used to change the values of variables defined as INOUT. Modules types have unique names, and store information about what input variables are required, what outputs can be expected, along with specific data type, unit, label, and meta information about each variable.

8.1.2.7 typedef float ssc_number_t

The numeric type used internally in SSC.

All numeric values are stored in this format. Do not change this without recompiling the library.

8.1.3 Function Documentation

8.1.3.1 SSCEXPORT const char* ssc_build_info ( )

Returns a ASCII text string that lists the compiler, platform, build date/time and other information.

Returns

Information about the build configuration of this particular SSC library binary.

8.1.3.2 SSCEXPORT void ssc_data_clear (ssc_data_t *p_data*)

Clears all of the variables in a data object.

8.1.3.3 SSCEXPORT ssc_data_t ssc_data_create ( )

Creates a new data object in memory. A data object stores a table of named values, where each value can be of any SSC datatype.

8.1.3.4 SSCEXPORT const char* ssc_data_first (ssc_data_t *p_data*)

Returns the name of the first variable in the table, or 0 (NULL) if the data object is empty.

8.1.3.5 SSCEXPORT void ssc_data_free (ssc_data_t *p_data*)

Frees the memory associated with a data object.

8.1.3.6 SSCEXPORT ssc_number_t* ssc_data_get_array (ssc_data_t *p_data,* const char * *name,* int * *length*)

Returns the value of a *SSC_ARRAY* variable with the given name.

8.1.3.7 SSCEXPORT ssc_number_t* ssc_data_get_matrix (ssc_data_t *p_data,* const char * *name,* int * *nrows,* int * *ncols*)

Returns the value of a *SSC_MATRIX* variable with the given name.

Note

Matrices are specified as a continuous array, in row-major order. Example: the matrix [[5,2,3],[9,1,4]] is stored as [5,2,3,9,1,4].

8.1.3.8 SSCEXPORT ssc_bool_t ssc_data_get_number (ssc_data_t *p_data*, const char * *name*, ssc_number_t * *value*)

Returns the value of a *SSC_NUMBER* variable with the given name.

8.1.3.9 SSCEXPORT const char* ssc_data_get_string (ssc_data_t *p_data*, const char * *name*)

Note

Retrieving variable values. These functions return internal references to memory, and the returned string, array, matrix, and tables should not be freed by the user.Returns the value of a *SSC_STRING* variable with the given name.

8.1.3.10 SSCEXPORT ssc_data_t ssc_data_get_table (ssc_data_t *p_data*, const char * *name*)

Returns the value of a *SSC_TABLE* variable with the given name.

8.1.3.11 SSCEXPORT const char* ssc_data_next (ssc_data_t *p_data*)

Returns the name of the next variable in the table, or 0 (NULL) if there are no more variables in the table. ssc_data_first must be called first.

8.1.3.12 SSCEXPORT int ssc_data_query (ssc_data_t *p_data*, const char * *name*)

Querys the data object for the data type of the variable with the specified name

Returns

data type, or SSC_INVALID if that variable was not found.

8.1.3.13 SSCEXPORT void ssc_data_set_array (ssc_data_t *p_data*, const char * *name*, ssc_number_t * *pvalues*, int *length*)

Assigns value of type *SSC_ARRAY*

8.1.3.14 SSCEXPORT void ssc_data_set_matrix (ssc_data_t *p_data*, const char * *name*, ssc_number_t * *pvalues*, int *nrows*, int *ncols*)

Assigns value of type *SSC_MATRIX*

Note

Matrices are specified as a continuous array, in row-major order. Example: the matrix [[5,2,3],[9,1,4]] is stored as [5,2,3,9,1,4].

8.1.3.15 SSCEXPORT void ssc_data_set_number (ssc_data_t *p_data*, const char * *name*, ssc_number_t *value*)

Assigns value of type *SSC_NUMBER*

8.1.3.16 SSCEXPORT void ssc_data_set_string (ssc_data_t *p_data*, const char * *name*, const char * *value*)

```
// Iterate over all variables in a data object
const char *key = ssc_data_first( my_data );
while (key != 0)
{
        // do something, like query the type
        int type = ssc_data_query( my_data, key );

        key = ssc_data_next( my_data );
}
```

Note

Assigning variable values. These functions do not take ownership of the data pointeres for arrays, matrices, and tables. A deep copy is made into the internal SSC engine. You must remember to free the table that you create to pass into ssc_data_set_table( ) for example.Assigns value of type *SSC_STRING*

8.1.3.17 SSCEXPORT void ssc_data_set_table (ssc_data_t *p_data*, const char * *name*, ssc_data_t *table*)

Assigns value of type *SSC_TABLE*.

8.1.3.18 SSCEXPORT void ssc_data_unassign (ssc_data_t *p_data*, const char * *name*)

Unassigns the variable with the specified name.

8.1.3.19 SSCEXPORT const char* ssc_entry_description (ssc_entry_t *p_entry*)

Returns a short text description of a compute module.

8.1.3.20 SSCEXPORT const char* ssc_entry_name (ssc_entry_t *p_entry*)

Returns the name of a compute module. This is the name that is used to create a new compute module.

8.1.3.21 SSCEXPORT int ssc_entry_version (ssc_entry_t *p_entry*)

Returns version information about a compute module.

8.1.3.22 SSCEXPORT const char* ssc_info_constraints (ssc_info_t *p_inf*)

Returns constraints on the values accepted. For example, MIN, MAX, BOOLEAN, INTEGER, POSITIVE are possible constraints.

8.1.3.23 SSCEXPORT int ssc_info_data_type (ssc_info_t *p_inf*)

Returns the data type of a variable: SSC_STRING, SSC_NUMBER, SSC_ARRAY, SSC_MATRIX, SSC-_TABLE

8.1.3.24 SSCEXPORT const char* ssc_info_group (ssc_info_t *p_inf*)

Returns any grouping information. Variables can be assigned to groups for presentation to the user, for example

8.1.3.25 SSCEXPORT const char* ssc_info_label (ssc_info_t *p_inf*)

Returns the short label description of the variable

8.1.3.26 SSCEXPORT const char* ssc_info_meta (ssc_info_t *p_inf*)

Returns any extra information about a variable

8.1.3.27 SSCEXPORT const char* ssc_info_name (ssc_info_t *p_inf*)

Returns the name of a variable

8.1.3.28 SSCEXPORT const char* ssc_info_required (ssc_info_t *p_inf*)

Returns information about whether a variable is required to be assigned for a compute module to run. It may alternatively be given a default value, specified as '?=

'.

8.1.3.29 SSCEXPORT const char* ssc_info_uihint (ssc_info_t *p_inf*)

Returns additional information for use in a target application about how to show the variable to the user. Not used currently.

8.1.3.30 SSCEXPORT const char* ssc_info_units (ssc_info_t *p_inf*)

Returns the units of the values for the variable

8.1.3.31 SSCEXPORT int ssc_info_var_type (ssc_info_t *p_inf*)

Returns variable type information: SSC_INPUT, SSC_OUTPUT, or SSC_INOUT

8.1.3.32 SSCEXPORT ssc_module_t ssc_module_create (const char * *name*)

Creates an instance of a compute module with the given name.

Returns

0 (NULL) if invalid name given and the module could not be created

8.1.3.33 SSCEXPORT ssc_entry_t ssc_module_entry (int *index*)

Returns compute module information for the i-th module in the SSC library.

Returns

0 (NULL) for an invalid index.

8.1.3.34 SSCEXPORT ssc_bool_t ssc_module_exec (ssc_module_t *p_mod*, ssc_data_t *p_data*)

Runs a configured computation module over the specified data set. Returns 1 or 0. Detailed notices, warnings, and errors can be retrieved either via a request_handler callback function, or using the ssc_module_-message function.

8.1.3.35 SSCEXPORT ssc_bool_t ssc_module_exec_simple (const char * *name*, ssc_data_t *p_data*)

The simplest way to run a computation module over a data set. Simply specify the name of the module, and a data set. If the whole process succeeded, the function returns 1, otherwise 0. No error messages are available. This function can be thread-safe, depending on the computation module used. If the computation module requires the execution of external binary executables, it is not thread-safe. However, simpler implementations that do all calculations internally are probably thread-safe. Unfortunately there is no standard way to report the thread-safety of a particular computation module.

8.1.3.36 SSCEXPORT const char* ssc_module_exec_simple_nothread (const char * *name*, ssc_data_t *p_data*)

Another very simple way to run a computation module over a data set. The function returns NULL on success. If something went wrong, the first error message is returned. Because the returned string references a common internal data container, this function is never thread-safe.

8.1.3.37 SSCEXPORT ssc_bool_t ssc_module_exec_with_handler (ssc_module_t *p_mod*, ssc_data_t *p_data*, ssc_bool_t(*)(ssc_module_t, ssc_handler_t, int action, float f0, float f1, const char *s0, const char *s1, void *user_data) *pf_handler*, void * *pf_user_data*)

A full-featured way to run a compute module with a callback function to handle custom logging, progress updates, and external binary execute requests

8.1.3.38 SSCEXPORT void ssc_module_extproc_output (ssc_handler_t *p_mod*, const char * *output_line*)

Helper function to be called from within a scc_exec_with_handler callback function to log binary output messages

8.1.3.39 SSCEXPORT void ssc_module_free (ssc_module_t *p_mod*)

Releases an instance of a compute module created with ssc_module_create

8.1.3.40 SSCEXPORT const char* ssc_module_log (ssc_module_t *p_mod*, int *index*, int * *item_type*, float * *time*)

Retrive notices, warnings, and error messages from the simulation

8.1.3.41 SSCEXPORT const ssc_info_t ssc_module_var_info (ssc_module_t *p_mod*, int *index*)

Returns references to variable info objects. Example for a previously created 'p_mod' object:

```
int i=0;
const ssc_info_t p_inf = NULL;
while ( p_inf = ssc_module_var_info( p_mod, i++ ) )
{
  int var_type = ssc_info_var_type( p_inf );   // SSC_INPUT, SSC_OUTPUT, SSC_INOUT
  int data_type = ssc_info_data_type( p_inf ); // SSC_STRING, SSC_NUMBER, SSC_ARRAY,
       SSC_MATRIX

  const char *name = ssc_info_name( p_inf );
  const char *label = ssc_info_label( p_inf );
  const char *units = ssc_info_units( p_inf );
  const char *meta = ssc_info_meta( p_inf );
  const char *group = ssc_info_group( p_inf );
}
```

Returns

Returns NULL for invalid index.

Note

Note that the ssc_info_* functions that return strings may return NULL if the computation module has not specified a value, i.e. no units or no grouping name.

8.1.3.42 SSCEXPORT int ssc_version ( )

Returns the library version. Version numbers start at 1.

Returns

SSC version number.