SSC Reference Manual

Version 36: SDK 2014-1-21

Aron Dobos and Paul Gilman

January 24, 2014

The SSC (SAM Simulation Core) software library is the collection of simulation modules used by the National Renewable Energy Laboratory's System Advisor Model (SAM). The SSC application programming interface (API) allows software developers to integrate SAM's modules into web and desktop applications. The API includes mechanisms to set input variable values, run simulations, and retrieve values of output variables. The SSC modules are available in pre-compiled binary dynamic libraries minimum system library dependencies for Windows, OS X, and Linux operating systems. The native API language is ISO-standard C, and language wrappers are available for C#, Java, MATLAB, and Python. The API and model implementations are thread-safe and reentrant, allowing the software to be used in a parallel computing environment.

This document describes the SSC software architecture, modules and data types, provides code samples, introduces SDKtool, and describes the language wrappers.

Contents

1 Overview

The SAM simulation core (SSC) software development kit (SDK) provides tools for creating models of renewable energy systems using the SSC library. The System Advisor Model (SAM) desktop application is a user-friendly front-end for the SSC library that uses SSC modules to perform all system performance and financial calculations. This manual assumes that the reader is familiar with SAM's suite of performance and financial models, input and output variables, and other capabilities. The SDK consists of the application programming interface (API), library, programming language wrappers, integrated development environment, code samples, and this manual.

1.1 Framework Description

The SSC framework provides a generic mechanism for running modules, setting values of module inputs, and reading values of module outputs. The mechanisms are generic because they do not depend on the module. For example, the same mechanism provides access to the PVWatts (`pvwattsv1`) and Utility IPP (`ippppa`) modules.

Using an SSC module is analogous to calling a function in a programming language: you pass parameter values to the function, run the function, and then read values returned by the function. In SSC, you store input variables in a data container, run the module, and retrieve outputs from the same data container:

1. Create a data container of type `ssc_data_t` for model input variables.
2. Call a simulation module such as `pvwattsv1` or `ippppa`, both of type `ssc_module_t` to process variables in the data container. If the module runs successfully, it populates the data container with output variables. If it fails, it generates error messages.
3. Retrieve results from the data container.

1.2 Modeling an Energy System

The SSC simulation modules process weather data files, simulate the performance of energy systems and their components, and calculate financial cash flows and metrics.

Creating a model of a renewable energy system in SSC involves running a set of modules to represent the physical system and a financing structure. For example, a model of a residential photovoltaic system might require four modules:

- `pvwattsv1`: Defines the system's design parameters and calculates the hourly output over a single year.

- `annualoutput`: Calculates the system's annual output over the system's multi-year life.
- `utilityrate`: Calculates the value of energy generated by the system for each hour of the first year, and as an annual total for subsequent years.
- `cashloan`: Calculates financial metrics including the cost of energy, net present value, based on installation and operating costs, incentives, taxes, and loan parameters.

The following pseudocode shows what a program that calculates the net present value of a residential photovoltaic system would look like. It calls the modules in the order shown above, with some outputs of each module serving as inputs for the next. Because the modules do not all use the same names or units for variables representing the same quantities, some code is required to translate values. For example, the `pvwattsv1` module's output variable `ac` which is the hourly energy in Wh produced by the system, while the `annualoutput` module's input variable `energy_net_hourly` is in kWh.

1. Create data container
2. Assign values to pvwattsv1 input variables
3. Run pvwattsv1
4. Retrieve system hourly output from data container
5. Convert hourly output values from Wh to kWh
6. Assign values to annual output input variables
7. Run annualoutput
8. Retrieve yearly output values from data container
9. Assign values to utilityrate input variables
10. Run utilityrate
11. Retrieve energy values from data container
12. Assign values to cashloan input variables
13. Run cashloan
14. Retrieve net present value from data container
15. Display net present value

2 A Basic Example: PVWatts

In this section, we write a short command-line C program to calculate the total annual energy production of a 1 kW PV system at a particular location. The program performs the following tasks:

1. Create Data Container
2. Assign values to the `pvwattsv1` input variables
3. Run `pvwattsv1`
4. Retrieve system hourly output values from the data container
5. Add up the hourly values and convert them to kWh
6. Display the total annual output

The complete source code for this program is included with the SSC SDK in the file `example1_pvwatts.c`.

This example is written in C, SSC's native language. The language interfaces included with the SDK and describe in §7 allow you to use SSC in programs written in other languages.

You can also use the scripting language included with SDKtool to run SSC modules and build models based on SSC as described in §6

2.1 Write the program skeleton

To start, we'll write the program skeleton. It includes the standard input-output and `sscapi.h` header files. The SSC API header file must be included in any program that uses SSC functions.

The main function gives the weather file as a command line argument. It must be in one of the formats that `pvwattsv1` can read, TMY2, TMY3, EPW, or SMW.

```
#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 Create the data container

Next we replace the comment in the program skeleton with code that creates the SSC data container that will store all input and output variables.

The `ssc_data_create()` function returns a value of type `ssc_data_t`, which we assign to the variable `data`.

If the function call fails due to the system having run out of memory, the function returns `NULL`. By checking the function's return value, we can be sure that it succeeded in creating the data container.

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

2.3 Assign values to the module input variables

The `pvwattsv1` module requires six input variables. You can use the approach described in §3.4 to list information about a module's input and output variables. You can also use the SDKtool module browser described in §6 to explore the variables.

The `pvwattsv1` input variables are numbers, except for the weather file name which is a string. SSC also supports array, matrix, and table data types.

```
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)
```

2.4 Run the module

To run the `pvwattsv1` module, we first create an instance of the module by passing its name as a string to the `ssc_module_create()` function, and then run the module.

The `ssc_module_create()` function may return `NULL` if it does not recognize the name or if the system is out of memory.

```
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;
}
```

The `ssc_module_exec` runs the module we created with the data container stored in `data` and returns a value of 1 if the simulation succeeds. If any of the module's required input variables use the wrong data type or a value that does not meet constraints, or if the simulation fails for some other reason, it returns a value of 0.

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

The `ssc_module_exec()` function is one way to run a module. For more details about the function and others that run modules with different options for handling errors, see §4.4.

2.5 Retrieve results from the data container and display them

If the simulation succeeds, we can retrieve the model outputs that we want from the data container. For this example, the only output we need is the hourly AC energy produced by the system, which we will use to calculate the system's total annual AC energy in kWh. For the `pvwattsv1` module, the system's hourly energy is stored in the `ac` variable in Wh.

```
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");
}
```

2.6 Cleaning up

Now that we're done with the simulation, we should free the memory associated with the module and data container. The `ssc_module_free()` function releases the module, and `ssc_data_free()` frees memory associated with the data container.

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

2.7 Compile and run the C program

This description explains how to compile and run the C program in Windows using the MinGW development environment, assuming we saved the code to the file `example1_pvwatts.c`. This example was tested with MinGW gcc version 4.6.2.

To compile and run the program, the following files must be in the same folder:

- The `.c` file containing the program code
- The `sscapi.h` header file
- The dynamic library file, for example `ssc32.dll`
- The `daggett.tm2` weather file

To compile the program, type the following command at the Windows command prompt:

```
c:\> gcc example1_pvwatts.c ssc32.dll -o pvwatts.exe
```

To run the program, specify a weather file on the command line. Here, we use the TMY2 file for Daggett, California:

```
c:\> pvwatts.exe daggett.tm2
ac: 1668.23 kWh
```

> **Note:** You may need to rename the `ssc32.dll` file to `ssc.dll` for the program to compile and run successfully.

2.8 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), permitting 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).

3.1 Variable Names

The variable name is how SSC identifies data, and your program must use the predefined names to refer to the variables. A variable names may contain letters, numbers, and underscores. SSC does not distinguish between uppercase and lowercase letters, so `Beam_Irradiance` is the same variable as `beam_irradiance`.

3.2 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 following constants listed below.

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

Input variables must be defined before calling a module. Output variables are calculated by the module, which adds them to the data container after running. In-out variables must be defined before running the module, which changes its value after running.

From the perspective of the data container, inputs and outputs are equivalent. The data container is just a pile of data, and provides no information about whether its variables are

input, output, or in-out variables. Only is only the simulation modules specify the variable type.

3.3 Data Types

Every variable in SSC is assigned a particular data type. Each simulation module specifies the data type of each variable it requires as input and 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 behind our decision 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 stores all numbers are stored as floating point. A module may limit a numeric variable to an integer, boolean (0 or 1), or positive value using a constraint flag as described in §3.5. The constraints are checked automatically before the module runs.

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. The weather file name is a common input variable that uses the `SSC_STRING` data type.

The table (`SSC_TABLE`) data type is the only hierarchical data type in SSC. It allows a

simulation module to receive or return outputs in a structured format with named fields. A table is 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 modules that may require them.

3.4 Variable Documentation

Each module defines all of the input variables it requires and output variables it produces. The SSC functions described below return information about the variables defined by a module. Each variable specifies the information shown in Table 1.

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 §3.5
Constraints	Constraints on the values or number of values. See §3.5
UI Hints	Suggestions on how to display this variable. Currently unused.

Table 1: Variable information provided by SSC

The `ssc_module_var_info()` function queries a module and returns a `ssc_info_t` reference. The `while` loop below calls `ssc_module_var_info()` until all of the module's variables have been displayed. This example assumes that the variable `module` was successfully created as described in §4.3 or in the example in §2.4. You can use a similar approach to list available modules with their descriptions as described in §4.6.

```
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 );
}
```

Another option for exploring a module's variables is SDKtool. It displays the variables in an interactive GUI spreadsheet-style interface described in §6.2.

3.5 Required Variables, Constraints, and Default Values

The *Required* and *Constraints* fields of a variable's documentation indicate whether the variable is required or optional, and whether it has a default value. Table 2 shows the *Required* flags and their meanings, and Table 3 shows the *Constraints* flags.

Flag	Description
`*`	Variable requires a user-specified value. If a required variable is not assigned a value before running the module, the simulation fails.
`?`	Optional variable. SSC ignores the variable unless a value is specified.
`?=<value>`	Default value, where `<value>` may be a number, a text string, or comma-separated array, depending on the data type of the variable.

Table 2: *Require* field tags

For some modules with a large numbers of input variables, SSC assigns default values to minimize the number of values that must be specified. SSC uses the default value unless a different value is specified. For example, the `pvwattsv1` module uses the default vaue of 45 degrees for `rotlim` unless the value is changed, indicating a default maximum tracker rotation angle of 45 degrees for single-axis trackers. Some other input variables may have a specialized purpose, and the default value is the recommended one for most applications. For example, the `pvwattsv1` input variable `enable_user_poa` variable enables an option

to replace the weather file with data representing the array's plane-of-array irradiance to allow for running the model with measured radiation data, or data from another model. The default value of zero is for the standard mode that uses a weather file as input.

SSC performs some data input validation before running a simulation using the *constraints* field. The field lists flags that SSC uses to limit the 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. These flags are defined internally by the modules, and are not user-editable. An empty *constraints* field does not necessarily indicate that the module does not check the variable's value before running. The module writer may have inadvertently omitted a constraint flag, and may use code within the module to check the values 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.

3.6 Manipulating Data Containers

As we saw in the PVWatts example in §2, a data container is created and freed using the `ssc_data_create()` and `ssc_data_free()` 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, use the functions `ssc_data_first()`

and `ssc_data_next()`. These functions return either 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 );
}
```

3.7 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.

3.7.1 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 first must be declared because 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" );
```

3.7.2 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.

3.7.3 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.

3.7.4 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.

3.7.5 Tables

Tables provide a way to communicate data to SSC in a hierarchical structure format. This data type is not used by any of the current SSC modules. (SDKtool (§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 straightforward:

```
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 §3.6 can be used to determine all of the fields of a table.

4 Simulation Modules

The SSC framework exposes several 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.

Table 4 shows the SSC simulation modules at the time of this writing. You can generate a list of current modules with descriptions using the method described in §4.6. Most of the SSC modules are available in the SAM user interface as a performance or financial model option, or as an option for modeling a component of the system.

SSC provides for ways to invoke a simulation module. Each method involves a different level of interaction during the execution of the model, differences in how errors and warnings are retrieved, and the appropriateness for use in a multithreaded environment.

4.1 Simple Method 1: Return True or False

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 );
```

Module	Description
6parsolve*	CEC module model with user inputs
pv6parmod*	CEC module model
pvsamv1	Flat-plate PV model
pvwattsv1	PVWatts model
pvwattsfunc*	PVWatts model for single timestep
pvwattsfrompoa*	PVWatts with POA radiation as input
pvsandiainv*	Sandia inverter model as a standalone module
wfreader	Weather file reader
irradproc	Irradiance processor for calculating POA irradiance
utilityrate*	Retail electricity rate for Version 1 of the OpenEI URDB
utilityrate2	Retail electricity rate for Version 2 of the OpenEI URDB
annualoutput	Calculate annual output for cash flow calculations
cashloan	Residential and commercial financial model
ippppa	Commercial PPA and Utility IPP financial models
timeseq	Convert start and end times, and time step into time stamp arrays
levpartflip	Leveraged partnership flip financial model
equpartflip	Partnership flip financial models
saleleaseback	Sale lease back financial model
singleowner	Single owner financial model
swh	Solar water heating model
test_irr*	IRR calculator from utility financing models for testing
test_pvshade*	PV self-shading model for testing
geothermal	Geothermal power model
windpower	Wind power model
poacalib*	Calibrates plane-of-array irradiance data
geothermalui	Calculates interim design values for the geothermal power model
wfcsvconv*	Converts a weather file to CSV
wfcsvread*	Reads weather file in CSV format
tcstrough_empirical*	Empirical trough model
tcstrough_physical*	Physical trough model
tcsgeneric_solar*	Generic solar system (CSP) model
tcsmolten_salt*	Molten salt power tower model
tcsdirect_steam*	Direct steam power tower model
tcslinear_fresnel*	Linear Fresnel model
tcsdish*	Dish-Stirling model
tcsiscc*	Integrated solar combined cycle model
tcsmslf*	Molten salt linear Fresnel model
hcpv	High-X concentrating photovoltaic model

Table 4: List of SSC simulation modules. (*) Indicates modules not used in SAM 2014.1.14.

```
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. The primary downside to using this simple invocation method is that the host program receives no progress updates, and the error messages and warnings cannot be programmatically retrieved after the module has finished running.

4.2 Simple Method 2: Return First Error Message

The second module invocation method is very similar to the first, except that the first error message is returned to the caller. Because this function uses a global variable to store the first error messages during the course of module execution, this invocation method should not be used in a multithreaded host environment. This is explicitly indicated by the `_nothread` suffix on the function 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. As with the simplest method 1, there is no way to retrieve additional errors or warnings after the module has finished running. Progress updates are also not provided to the calling environment.

4.3 Explicit Module Instantiation

A third way to run a computation module is to explicitly create a module object, execute it with a data container, and then release it. This is the method we used for the PVWatts example in §2.4.

The SSC function `ssc_module_create()` function is used to create a new instance of a computation module. The function returns an opaque reference of type `ssc_module_t` to the module object, or returns `NULL` if the module could not be created. The single parameter required is the name of the module requested. If a module is created successfully, it must eventually be released when it is no longer needed via the `ssc_module_free()` function. Example:

```
// ... set up a data container with inputs ...
ssc_module_t module = ssc_module_create( "cashloan" );
if ( NULL == module )
    printf("could not create 'cashloan' module!\n");
else
{
    // ... query the module for variable info or run it ...

    // free the module when it is no longer needed
    ssc_module_free( module );
}
```

The advantage of explicitly creating a module is that all errors, warnings, and log messages generated during the course of the module's execution will be saved and can be later retrieved using the `ssc_module_log()` function, which is described subsequently in detail in §4.5. Alternatively, if the goal is simply to obtain information about the input and output variables associated with a module as in §3.4, the module must be explicitly instantiated also.

The next section describes a mechanism for running compute modules and providing real-time progress updates, reporting messages as they occur, and canceling a module's execution before it has finished.

4.4 Running with Progress Updates, Warnings, and Errors

The last (most complex) way to run a module is to provide a special function that can handles error and warning messages as they are generated, as well as provide progress updates to the controlling host 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 module invocation method is only supported in the native C language API for SSC. The other programming language interfaces provided with SSC (MATLAB, Python, C#, Java) do not support callback functions via the foreign-function-interface (FFI) wrappers. 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_handler()` 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 SDKtool (based on the 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; // invalid action
}
```

The return value of the handler function is checked by compute modules to determine whether the calculations should proceed. In the example above, no facility is provided for canceling a simulation as the handler function always returns 1 (true). However, a dialog box with a cancel button could set a flag in the user data structure when pressed.

If the cancel flag is set to true, the handler function could return 0 (false) to abort the simulation. It is left as an exercise to the user to implement such functionality in their host program.

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.

4.6 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 );
}
```

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 SDKtool

SDKtool is the developer interface for SSC. (SSCDev was renamed to SSC SDKtool with the January 24 release of the SDK.) SDKtool allows you to explore variables in the SSC modules, run individual modules, and write scripts to build and test your models. When you run modules in SDKtool, it displays all of the variables in the data container with their values. Use SDKtool for the following tasks:

Operating System	Directory Name	SDKtool	SSC Library
Windows 32-bit	win32	sdktool.exe	ssc.dll
Windows 64-bit	win32	sdktool.exe	ssc.dll
OS X 64-bit	osx64	SDKtool.app	ssc.dylib
Linux 64-bit	linux64	SDKtool	ssc.so

Table 5: SDKtool and Library Versions

- Learn how modules you plan to use in your model work before before you start programming.
- Troubleshoot problems with your implementation of SSC modules.
- Build and test models in LK Script using the SDKtool script editor. LK Script is documented in `lk_guide.pdf`, included in the SDK package.

6.1 Using SDKtool

Using SDKtool involves running the executable, and loading the appropriate SSC library:

1. Run the executable file shown in Table 5 for your operating system. For example, the 32-bit Windows executable. `sdktool.exe` is in the `win32` directory of the SDK package.
2. Click **Choose SSC library** and choose the library for your version of SDKtool. For example, for 32-bit Windows, choose the `ssc.dll` library from the `win32` directory.

   A list of modules should appear in the module browser, and the library path, name, and version number should appear in the status bar at the bottom of the SDKtool window. If the module browser is empty, try choosing the SSC library again.
3. Click a module's name to see its variables. For example, click `pvwattsv1` to see a list of the PVWatts module's 53 variables.

SDKtool displays a list of available modules, and lists each module's input and output variables, but does not provide enough information to completely understand the modules. You should be prepared to use the SAM user interface and documentation together with SDKtool to help you understand how the modules and variables work.

For example, the `annualoutput` module includes a variable named `energy_availability` that uses the data type `SSC_ARRAY`. Why is the data type not `SSC_NUMBER` instead? To find out, we can examine the `annualoutput` input variables in SAM's user interface, which displays them on the Performance Adjustment input page. The `energy_availabilty` variable appears with the label "Percent of annual output," and SAM provides two options

for entering the variable's value: As a single value or as an annual schedule. The array data type is necessary to store the annual values. The `annualoutput` module assumes that if there is a single non-zero value at the top of the array, that value represents a single adjustment factor that applies to all years. Otherwise it applies values from the array to each year.

For some variables, even the SAM user interface may not be helpful. When you need help with an SSC module or variable, post a question about it on the SAM support forum at https://sam.nrel.gov/forums/support-forum. For example, you might wonder what the `annualoutput` module's `system_use_lifetime_output` variable does. That variable is not on any of SAM's input pages, and the information in SDKtool does not explain its function. The variable determines whether the module generates a single set of 8,760 hourly values for the `hourly_e_net_delivered` variable, or whether it generates a set of 8,760 values for each year in the analysis period. All of SAM's performance models use a value of zero except for the geothermal model, which is a special case and requires lifecycle output data to accurately represent changes in resource over time as it is depleted and replenished. For most cases, you should set its value to zero.

Script

The Script menu commands duplicate the buttons in the Script Editor. They are implemented as menu items to make the shortcut keys available.

Open Open a script file (`.lk`).

Save Save the current script.

Find Find text in the current script.

Find next Find the next instance of a text string in the current string.

Run script Run the current script.

6.2 Module Browser

The module browser displays the modules available in the SSC library. The module browser can:

- Display a list of available SSC modules.
- Display a module's variables.
- Copy a tab-delimited list of a module's variables to your computer's clipboard.
- Run a single module using the variable values defined in the data container.

You can use the module browser as a reference while you are developing your model to help ensure that you assign values to all of the required input variables, and use the correct data

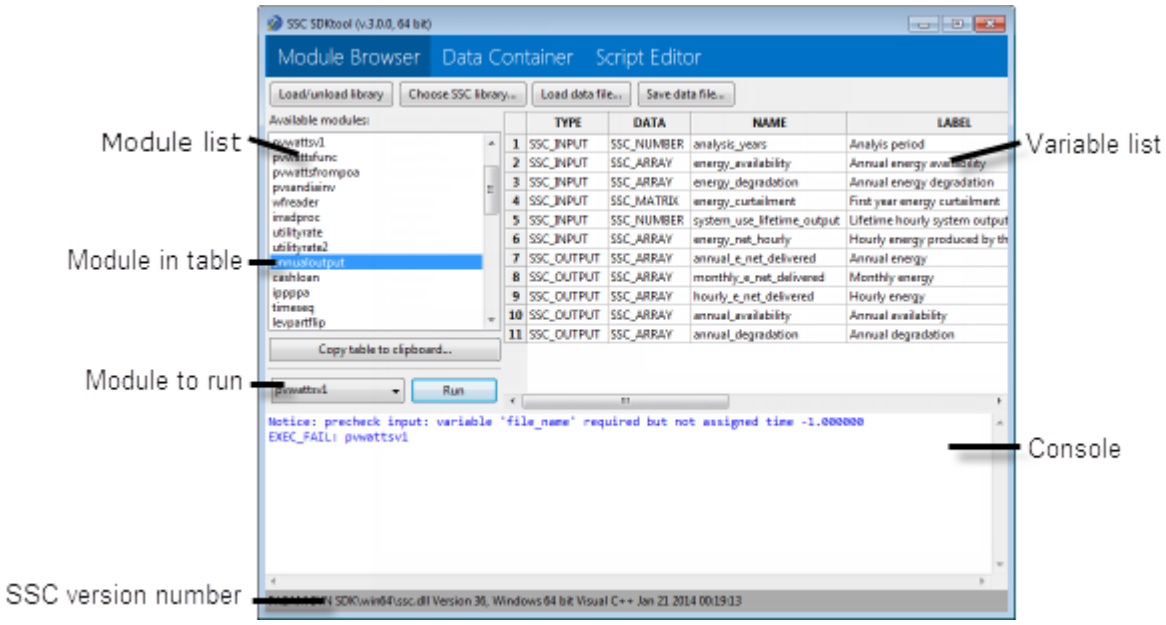

Figure 1: Module browser

types and units.

You can also use the module browser with the data container to manually set values of input variables and run modules individually. This might be useful for learning how a module works, or for troubleshooting your model's implementation of a module.

Module browser commands:

Load/unload library Loads or unloads the current SSC library for troubleshooting. If a module does not run, you can try unloading and reloading the library.

Choose SSC library Displays a file dialog where you can choose and load the SSC library. Be sure to choose the correct library for your version of SDKtool.

Load data file Open a data file. (`.bdat`).

Save data file Save the variables in the data container to a data file. (`.bdat`).

Copy table to clipboard Copies the table of module variables as tab-delimited text to your computer's clipboard.

Run Runs the active module in the module browser. (You must select a module from the dropdown list before you click **Run**.)

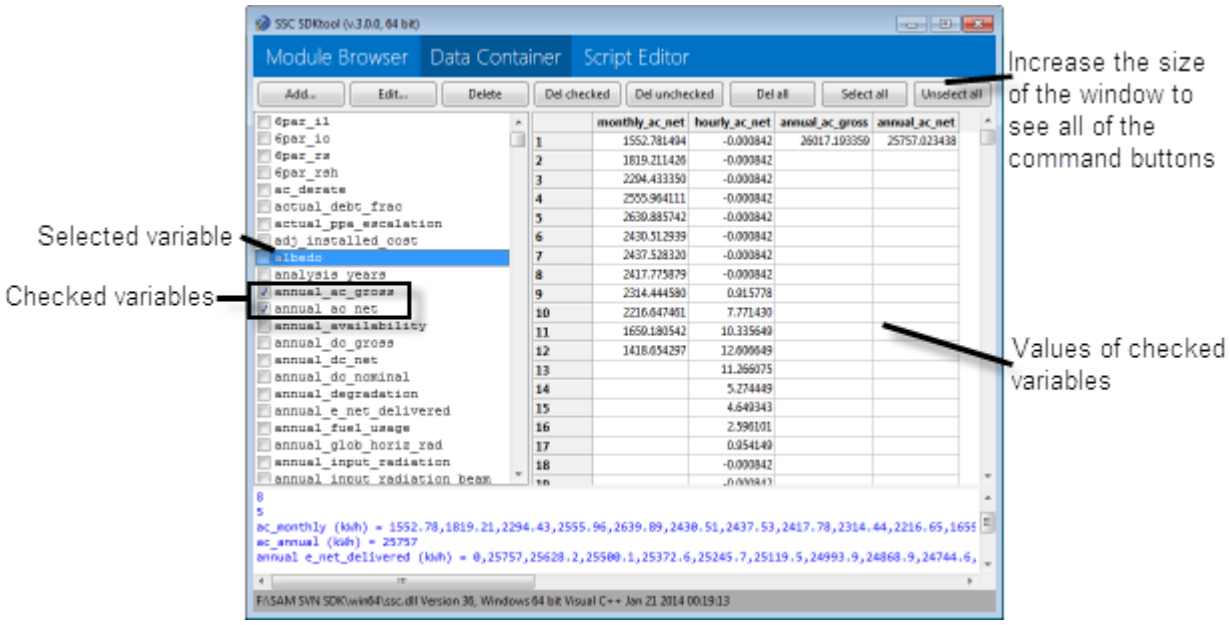

Figure 2: Data container

6.3 Data Container

The data container displays a list of variables for the current set of simulations. When you run a module from the data browser, you must add input variables to the data container by hand using the **Add** and **Edit** buttons.

When you run modules from the script editor, the data container is automatically populated with variables from the script. You can use the data container commands for the following tasks:

- Add, edit, and remove variables from the data container.
- Copy a tab-delimited list of variable names and values to your computer's clipboard.
- Show graphs of time series data.

The data container commands act either on the selected variable indicated with a blue highlight, or on variables with a check mark. You may need to maximize the SDKtool window to see all of the command buttons:

Add Add a variable to the data container. If you add a module's input variable to the data container, be sure to use the variable's exact name and data type as it appears in the module browser.

Edit Change the selected variable's name, data type, or value.

Delete Remove the selected variable from the data container.

Del checked Remove all variables with check marks from the data container.

Del unchecked Remove all variables without check marks from the data container.

Del all Delete all variables from the data container.

Select all Check all variables in the data container. This command may take several seconds to complete.

Unselect all Clear check marks from all variables in the data container.

Copy to clipboard Copy the data container table as a tab-delimited list of selected variable names and their values to the clipboard.

Show stats Display a table of statistics for the selected variable, which must be of type `SSC_ARRAY` with 8,760 values.

Timeseries graph Display time series and statistical graphs of variables with check marks, which must be of type `SSC_ARRAY` with 8,760 values.

> **Note:** If you manually add variables to the data container before running a script that does not use the variables, they will be deleted from the data container.

6.3.1 Data Files

You can save and open the contents of the data container in a data file (`.bdat`) using the **Save data file** and **Load data file** commands on the File menu. The data files use a binary format, and store all of the variables in the data container.

6.3.2 Time Series Graphs and Statistical Tables

When the data container includes hourly arrays with 8,760 values, you can display the hourly data and statistical summaries:

Select an hourly array and click **Show stats** to display a statistical summary of the array data in a table. You can export data from the monthly table by selecting it using your computer's copy-to-clipboard keyboard shortcut (Ctrl-C) in windows.

Check one or more hourly arrays and click **Timeseries graph** time series data and statistical summaries in graphs. You can export graph images and tables of graph data by right-clicking graphs and selecting commands in the shortcut menu.

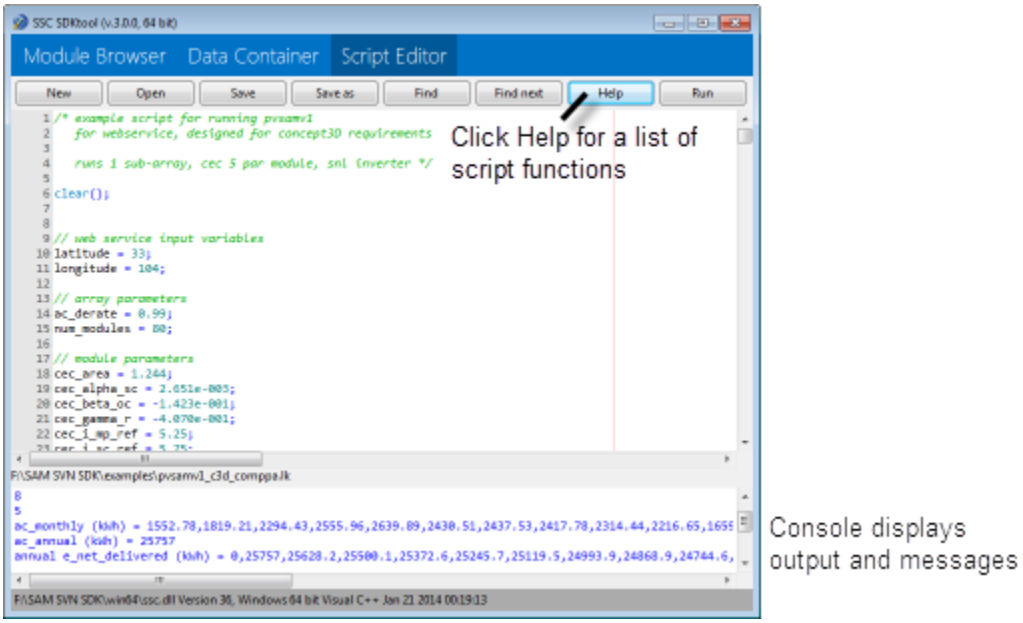

Figure 3: Script editor

See §6.5 for an example.

6.4 Script Editor

The script editor allows you to write, edit, and run scripts in LK Script to build and test models using the SSC modules. Documentation of LK Script is included in the SDK package in `lk_guide.pdf`.

The script editor commands are:

New Clear the current script from the script editor.

Open (**Ctrl-O**) Open a script file (`.lk`).

Save (**Ctrl-S**) Save the current script file (`.lk`).

Save as Save the current script to a different file (`.lk`).

Find Find text in the current script.

Find next Find the next instance of text in the current script.

Help Display a list of script functions.

Run Run the current script.

6.5 SDKtool Example 1: Run a Module from the Module Browser

The following example shows how to use the module browser to expore and run the `pvwattsv1` module. You can use the same steps to explore other simulation modules.

> **Note:** As described in §6.2, the module browser can only run a single simulation module at a time. To combine modules to build a model, you can write a script as described in §6.6.

As we saw in §1.2, the `pvwattsv1` simulation module reads data from a weather file and inputs from the data container that define the system's size, array orientation and type of tracking, and derating factor, and calculates values for the 1 by 8760 array storing the system's hourly electricity output over a single year.

To explore the `pvwattsv1` simulation module in the module browser:

1. Start SDKtool and be sure the correct SSC library is loaded as described in §6.1
2. In the **Module Browser**, click the `pvwattsv1` name to see a list of the module's variables. The inputs are of type `SSC_INPUT`. Of the inputs, only a subset indicated by (`*`) are required and have no default value, as described in §3.5. At a minimum, you must assign values to these variables to run the module.
3. In the drop-down list at the bottom of the window under the list of available modules, choose the `pvwatts1` module and click **Run**.

   A message about the `file_name` variable appears in the console. That is because `file_name` is a required variable, but it does not yet have a value.
4. In the **Data Container**, click **Add**, and type "file_name" for the variable name, assign it the `SSC_STRING` data type, and click **file** to assign it the value of one of the TMY2 files included in the SDK Examples folder (for example, *daggett.tm2*).
5. Next, add the remaining 5 required variables shown in the module browser to the data container, using the data type and units shown in the table. For example, `system_size` should be `SSC_NUMBER` with a value in kilowatts. Be sure to use variable names and data types that exactly match those in the module browser.

   When you are finished, you should see a table similar to Table 6. This is the model's data container, which so far contains the required input for the `pvwattsv1` simulation module.
6. In the **Module Browser**, click **Run** to run the module.

The data container should now show a table with all of the `pvwattsv1` module variables. You can edit and view the variable values:

- Use the check boxes to add or remove variables from the table. Double-click the

file_name	system_size	derate	track_mode	azimuth	tilt
.../daggett.tm2	4	0.77	0	180	20

Table 6: The 6 required input variables for the pvwattsv1 module with their values

variable's name to edit it (or select the variable and click **Edit**).

- Display graphs of any array with 8,760 values. For example, to view the cell temperature and AC output variables, clear all of the check boxes, check the `tcell` and `ac` variables, and click **Time series graph**.
- Display a table of statistical values for any variable with 8,760 values. For example, select the `poa` variable to show the plane-of-array irradiance minimum, maximum, total, mean, and monthly values, and click **Show stats**.

> **Note:** To display the statistical summary of an hourly variable, select it by clicking its name in the list rather than checking it.

6.6 SDKtool Example 2: Build a Model in LK Script

This example uses a script to run the `pvwattsv1` simulation module as we did in Example 1, and combine it with the `annualoutput` module to create a simple model that estiimates a PV system's annual output over a multi-year period.

Writing a script is a useful way to test your model before you implement it in C++ or another programming language. When you develop the model as a script in SDKtool, you can use the module browser and data container to help you troubleshoot your model within an environment designed specifically to run the modules. Once you get your model running as a script, you can easily translate it into another language.

Copy or cut and paste the script examples below into the SDKtool's script editor.

The first part of the script performs three tasks:

1. Assign values to the required input variables for `pvwattsv1`.
2. Run the module.
3. Display the total annual output.

The next part of the script uses the system's AC output calculated by the `pvwattsv1` module to estimate the system's total annual output over multiple years:

1. Assign values to the required input variables for `annualoutput`.
2. Run the module.

3. Display the year-by-year output.

The `pvwattsv1` module's required input variables are shown in Table 6. The module browser shows each variable's data type, units, and any constraints.

```
// Create variables for the six pvwattsv1 module required inputs
weather_file = 'F:/SAM SDK/examples/daggett.tm2';
nameplate_capacity = 4; // DC kW
derating_factor = 0.77;
tracking = 0; // fixed
azimuth = 180; // due south
tilt = 20; // from horizontal

// Assign input values to the module inputs
var('file_name',weather_file);
var('system_size',nameplate_capacity);
var('derate',derating_factor);
var('track_mode',tracking);
var('azimuth',azimuth);
var('tilt',tilt);

// Run module
run('pvwattsv1');

// Get results
year_1_energy_total = var('ac_annual');
year_1_energy_hourly = var('ac');

sum_of_hourly = sum(year_1_energy_hourly);

// Display results
outln("Year 1 output: " + sprintf('%,',to_int(year_1_energy_total)) + " kWh");
outln("  as sum of hourly output: " + sprintf('%,',to_int(sum_of_hourly)) + " Wh");
outln();
```

The next part of the script uses the `annualoutput` module to calculate the system's output over a multi-year period. It has the six required inputs shown in Table 7.

The module browser does not provide information about the size of the `energy_availability` and `energy_degradation` arrays, or the `energy_curtailment` matrix. We can refer to the SAM user interface to explore those variables further. SAM displays them on the Performance Adjustment page as shown in Table 8.

Name	Data Type	Units
analysis_years	`SSC_NUMBER`	years
energy_availability	`SSC_ARRAY`	%
energy_degradation	`SSC_ARRAY`	%
energy_curtailment	`SSC_MATRIX`	none
system_use_lifetime_output	`SSC_NUMBER`	none
energy_net_hourly	`SSC_ARRAY`	kW

Table 7: The six required input variables for the `annualoutput` module

Variable	Name in SAM	Size
energy_availability	Percent of annual output	1x1 to 1x50
energy_degradation	Year-to-year decline in output	1x1 to 1x50
energy_curtailment	Hourly Factors	24x12

Table 8: Information from the SAM user interface about the inputs to `annualoutput`

The SAM user interface allows the **Percent of annual output** (`energy_availability`) and **Year-to-year decline in output** (`energy_degradation`) variables to be specified either as a single value or as an "annual schedule." The input table for the annual schedule has a maximum size of 50 rows, indicating the maximum size of the arrays. If the array has a single value, `annualoutput` applies that value to each year in the analysis period. This behavior is described in SAM's Help system.

The 24 columns and 12 rows of the input table for the **Hourly Factors** variable shows the dimensions of the (`energy_curtailment`) matrix.

```
// Create variables for the annualoutput module required inputs
// system_output already defined above
analysis_period = 25;
availability = [98];
degradation = [0.05];
curtailment = [ [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
```

```
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,],
                [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,] ];
hourly_mode = 0;

// Assign input values to the module inputs
var('analysis_years', analysis_period);
var('energy_availability', availability);
var('energy_degradation', degradation);
var('energy_curtailment', curtailment);
var('system_use_lifetime_output', hourly_mode);
var('energy_net_hourly', year_1_energy_hourly);

// Run module
run('annualoutput');

yearly_energy = var('annual_e_net_delivered');

for ( i = 1; i <= analysis_period; i++)
{
  outln("Year " + i + " output: " + sprintf('%,',to_int(yearly_energy[i])) + ( " kWh"));
}
```

7 Language Interfaces

The SSC software development kit (SDK) includes interfaces (or wrappers) for using SSC directly from programming languages other than its native C language interface. This section describes each language interface and requirements, as well any limitations in functionality compared with the native C interface.

Each language interface comes with a set of readme and example files that you should refer to in addition to the descriptions below to set up your system to run SSC with a language interface.

7.1 Python

The Python language interface has been tested on Windows with the ActivePython distribution, and on the Linux and OSX native Python packages.

The Python API is defined in the `sscapi.py` source file included in the SDK. The file includes example code if it is invoked as the main source program.

> **Note:** In the PySSC class constructor, a *Ctypes* object is created by referencing the `ssc.dll` dynamic library. It is up to the user to modify the file name and/or path of this library to be appropriate for the operating system and platform.

The structure of SSC API calls in Python is slightly different from the C interface because in Python, an instance of the PySSC class must be created first. Example:

```
ssc = PySSC()
dat = ssc.data_create()

ssc.data_set_string(dat, 'file_name', 'daggett.tm2')
ssc.data_set_number(dat, 'system_size', 4)
ssc.data_set_number(dat, 'derate', 0.77)
ssc.data_set_number(dat, 'track_mode', 0)
ssc.data_set_number(dat, 'azimuth', 180)
ssc.data_set_number(dat, 'tilt_eq_lat', 1)

# run PV system simulation
mod = ssc.module_create("pvwattsv1")

if ssc.module_exec(mod, dat) == 0:
    print 'PVWatts V1 simulation error'
    idx = 1
    msg = ssc.module_log(mod, 0)
    while (msg != None):
        print '\t: ' + msg
        msg = ssc.module_log(mod, idx)
        idx = idx + 1
else:
    ann = 0
    ac = ssc.data_get_array(dat, "ac")
    for i in range(len(ac)):
        ac[i] = ac[i]/1000
        ann += ac[i]
    print 'PVWatts V1 Simulation ok, e_net (annual kW)=', ann
    ssc.data_set_array(dat, "e_with_system", ac) # copy over ac

ssc.module_free(mod)
```

```
ssc.data_free( dat )
```

Notice that the structure of creating and freeing data containers and modules is identical to the C language API, except that the functions are members of the PySSC class. For more information, see the `sscapi.py` source file for additional examples and usage cases.

7.2 MATLAB

MATLAB is a numerical computing language that is widely used for data processing, algorithm development, and plotting. It is a proprietary product that is available on OSX, Linux, and Windows.

The built-in MATLAB *foreign function interface (FFI)* library routines that are used to interface to SSC require a C compiler to preprocess the SSC header file. Some versions of MATLAB include a C compiler such as LCC installed by default, while others may require that one to be installed separately. See `http://www.mathworks.com/support/compilers/R2012b/win64.html` for more information.

The MATLAB environment should also be configured to recognize the path containing the SSC MATLAB files.

The MATLAB language interface automatically attempts to detect the platform and load the correct SSC dynamic library. You must ensure that the `sscapi.h` C language header file can be located by MATLAB, as well as the library appropriate for your system.

For additional information, see the MATLAB code examples supplied with the SSC SDK, as well as the content of the `ssccall.m` file itself.

The SSC API is provided in the `ssccall.m` M-file. The interface provides a single MATLAB function called `ssccall(...)` that accepts different parameter arguments to invoke the various SSC library calls. The example below shows how to setup and run the *pvwattsv1* module from MATLAB.

```
ssccall('load');

% create a data container to store all the variables
data = ssccall('data_create');

% setup the system parameters
ssccall('data_set_string', data, 'file_name', 'abilene.tm2');
ssccall('data_set_number', data, 'system_size', 4);
ssccall('data_set_number', data, 'derate', 0.77);
ssccall('data_set_number', data, 'track_mode', 0);
```

```
ssccall('data_set_number', data, 'tilt', 30);
ssccall('data_set_number', data, 'azimuth', 180);

% create the PVWatts module
module = ssccall('module_create', 'pvwattsv1');

% run the module
ok = ssccall('module_exec', module, data);
if ok,
    % if successful, retrieve the hourly AC generation data and print
    % annual kWh on the screen
    ac = ssccall('data_get_array', data, 'ac');
    disp(sprintf('pvwatts: %.2f kWh',sum(ac)/1000.0));
else
    % if it failed, print all the errors
    disp('pvwattsv1 errors:');
    ii=0;
    while 1,
        err = ssccall('module_log', module, ii);
        if strcmp(err,''),
            break;
        end
        disp( err );
        ii=ii+1;
    end
end

% free the PVWatts module that we created
ssccall('module_free', module);

% release the data container and all of its variables
ssccall('data_free', data);

% unload the library
ssccall('unload');
```

7.3 C# and .NET

The C# language interface provides direct access to the SSC library from the Microsoft .NET software environment. C# is an object-oriented garbage-collected language that is

compiled typically to an intermediate bytecode that is executed in a managed environment by the bytecode interpreter engine. Objects and memory are directly managed by the interpreter, relieving programmers of the burdens of detailed memory management. However, since SSC is implemented at its core in a native compiled machine code library, care must be taken by the typical C# programmer to ensure that the interface to the unmanaged SSC library is properly utilized.

The C# SSC language interface is provided as a single source file called `SSC.cs`. This file contains direct invocation mappings to the unmanaged C library API, as well as an easy-to-use wrapper around the low-level function calls. This API has been tested using Visual Studio 2012 .NET on Win32 and x64. .NET implementations on other platforms such as Mono have not been tested.

To call the SSC library from C#, include the `SSC.cs` file in your .NET project in Visual Studio. For each opaque data pointer type in the SSC native C API, there is a similarly named C# class within the SSC namespace. See the example below on how to create a data container and module to run PVWatts. Additional examples of the other SSC API functions using the C# wrapper classes is available in the included test program in the SDK.

```
private void btnPVWatts_Click(object sender, EventArgs e)
{
    txtData.Clear();

    SSC.Data data = new SSC.Data();

    data.SetString("file_name", "abilene.tm2");
    data.SetNumber("system_size", 4.0f);
    data.SetNumber("derate", 0.77f);
    data.SetNumber("track_mode", 0);
    data.SetNumber("tilt", 20);
    data.SetNumber("azimuth", 180);

    SSC.Module mod = new SSC.Module("pvwattsv1");

    if (mod.Exec(data))
    {
        float tot = data.GetNumber("ac_annual");
        float[] ac = data.GetArray("ac_monthly");
        for (int i = 0; i < ac.Count(); i++)
            txtData.AppendText("[" + i + "]: " + ac[i] + " kWh\n");
        txtData.AppendText("AC total: " + tot + "\n");
```

```
        txtData.AppendText("PVWatts test OK\n");
    }
    else
    {
        int idx = 0;
        String msg;
        int type;
        float time;
        while (mod.Log(idx, out msg, out type, out time))
        {
            String stype = "NOTICE";
            if (type == SSC.API.WARNING) stype = "WARNING";
            else if (type == SSC.API.ERROR) stype = "ERROR";

            txtData.AppendText("[ " + stype + " at time:" + time + " ]: " + msg + "\n");
            idx++;
        }

        txtData.AppendText("PVWatts example failed\n");
    }
}
```

> **Note:** The SSC C# class library is designed to properly handle allocation and freeing of underlying native SSC data types. However, due to the nature of the .NET runtime environment's garbage collector, it may appear that SSC objects are not freed right away. While it is possible to invoke the garbage collector manually, it is generally not recommended to do so by the Visual Studio documentation.

For more information, consult the `SSC.cs` source file, as well as the example Visual Studio project provided with the SDK.

7.4 Java

The Java interface to SSC utilizes the low-level Java Native Interface (JNI) specification to directly access the SSC library. The JNI wrapper consists of a C source file `jssc.c` that wraps the SSC native C language API into JNI methods that can be compiled and loaded by the Java Virtual Machine (JVM) at runtime. The source code for the JNI wrapper is included in the SDK, as well as a more programmer-friendly high-level Java `SSC` class that

invokes the JNI layer.

The JNI interface marshals opaque C pointers as the 64-bit `jlong` data type, ensuring compatibility in both 32 and 64 bit implementations.

Compiling the JNI interface layer requires a C compiler and the Java SDK (for the JNI header files). The example included with the SSC SDK was tested using the freely downloadable MinGW gcc compiler on 32-bit Windows (`https://www.mingw.org`), and the Java JDK 1.7. Refer to the `README.txt` in the `[sdk]/languages/java/` folder for specific information on compiling the JNI interface layers for your particular SDK runtime environment.

Once the JNI wrapper library has been compiled for your particular JDK version, the PVWatts example can be invoked from within Java as below:

```
public class SSC_Java_Example {

    public static void main(String[] args)
    {
        System.loadLibrary("sscapiJNI32");

        SSC sscobj = new SSC("pvwattsv1");

        sscobj.Set_String("file_name", "abilene.tm2" );
        sscobj.Set_Number("system_size", 4.0f );
        sscobj.Set_Number("derate", 0.77f );
        sscobj.Set_Number("track_mode", 0 );
        sscobj.Set_Number("tilt", 20 );
        sscobj.Set_Number("azimuth", 180 );

        if ( sscobj.Exec() )
        {
            float[] ac = sscobj.Get_Array( "ac" );
            float sum = 0;
            for( int i=0;ac.length;i++)
            {
                sum += ac[i];
            }
            System.out.println("ac total: " + sum);
            System.out.println("PVWatts example passed");
        }
        else
        {
```

```
            System.out.println("PVWatts example failed");
        }
    }
}
```

For more information on utilizing the Java language layer, consult Java code examples provided in the SDK.

8 C API Reference

8.1 sscapi.h File Reference

SSC: SAM Simulation Core.

Macros

- #define **SSCEXPORT**

Data types:

Possible data types for variables in an ssc_data_t:

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

Variable types:

- #define **SSC_INPUT** 1
- #define **SSC_OUTPUT** 2
- #define **SSC_INOUT** 3

Action/notification types that can be sent to a handler function:

SSC_LOG: Log a message in the handler. f0: (int)message type, f1: time, s0: message text, s1: unused. SSC_UPDATE: Notify simulation progress update. f0: percent done, f1: time, s0: current action text, s1: unused.

- #define **SSC_LOG** 0
- #define **SSC_UPDATE** 1

Message types:

- #define **SSC_NOTICE** 1
- #define **SSC_WARNING** 2
- #define **SSC_ERROR** 3

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 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 const char * ssc_module_log (ssc_module_t p_mod, int index, int *item_type, float *time)
- SSCEXPORT void __ssc_segfault ()

Assigning variable values.

The following 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.

- 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)

Retrieving variable values.

The following functions return internal references to memory, and the returned string, array, matrix, and tables should not be freed by the user.

- 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)

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.

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 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.

8.1.2.2 typedef float ssc_number_t

The numeric type used in the SSC API. All numeric values are stored in this format. SSC uses 32-bit floating point numbers at the library interface to minimize memory usage. Calculations inside compute modules generally are performed with double-precision 64-bit floating point internally.

8.1.2.3 typedef int ssc_bool_t

The boolean type used internally in SSC. Zero values represent false; non-zero represents true.

8.1.2.4 typedef void* ssc_entry_t

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

8.1.2.5 typedef void* ssc_module_t

An opaque reference to a computation module. 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.

8.1.2.6 typedef void* ssc_info_t

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

8.1.2.7 typedef void* ssc_handler_t

An opaque pointer for transferring external executable output back to SSC

8.1.3 Function Documentation

8.1.3.1 SSCEXPORT int ssc_version ( )

Returns the library version number as an integer. Version numbers start at 1.

8.1.3.2 SSCEXPORT const char* ssc_build_info ( )

Returns information about the build configuration of this particular SSC library binary as a text string that lists the compiler, platform, build date/time and other information.

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 void ssc_data_free (ssc_data_t *p_data*)

Frees the memory associated with a data object, where p_data is the data container to free.

8.1.3.5 SSCEXPORT void ssc_data_clear (ssc_data_t *p_data*)

Clears all of the variables in a data object.

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

Unassigns the variable with the specified name.

8.1.3.7 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 the data object's data type, or SSC_INVALID if that variable was not found.

8.1.3.8 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.

8.1.3.9 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. Example that iterates over all variables in a data object:

```
const char *key = ssc_data_first( my_data );
while (key != 0)
{
    int type = ssc_data_query( my_data, key );
    key = ssc_data_next( my_data );
}
```

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

Assigns value of type *SSC_STRING*

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

Assigns value of type *SSC_NUMBER*

8.1.3.12 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*

8.1.3.13 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* . 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].

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

Assigns value of type *SSC_TABLE*.

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

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

8.1.3.16 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.

8.1.3.17 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.

8.1.3.18 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. 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].

8.1.3.19 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.

8.1.3.20 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. 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) );
}
```

8.1.3.21 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.

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

Returns a short text description of a compute module.

8.1.3.23 SSCEXPORT int ssc_entry_version (ssc_entry_t *p_entry*)

Returns version information about a compute module.

8.1.3.24 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

8.1.3.25 SSCEXPORT void ssc_module_free (ssc_module_t *p_mod*)

Releases an instance of a compute module created with ssc_module_create

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

Returns references to variable info objects. Returns NULL for invalid index. 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. 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 );
}
```

8.1.3.27 SSCEXPORT int ssc_info_var_type (ssc_info_t *p_inf*)

Returns variable type information: SSC_INPUT, SSC_OUTPUT, or SSC_INOUT

8.1.3.28 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

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

Returns the name of a variable

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

Returns the short label description of the variable

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

Returns the units of the values for the variable

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

Returns any extra information about a variable

8.1.3.33 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

8.1.3.34 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 '?=

’.

8.1.3.35 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.

8.1.3.36 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.

8.1.3.37 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.

8.1.3.38 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.

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

Runs an instantiated computation module over the specified data set. Returns Boolean: 1 or 0. Detailed notices, warnings, and errors can be retrieved using the ssc_module_log function.

8.1.3.40 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 cancelation requests. Returns Boolean: 1 or 0 indicating success or failure.

8.1.3.41 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. Returns a NULL-terminated ASCII C string with the message text, or NULL if the index passed in was invalid.

8.1.3.42 SSCEXPORT void __ssc_segfault ( )

DO NOT CALL THIS FUNCTION: immediately causes a segmentation fault within the library. This is only useful for testing crash handling from an external application that is dynamically linked to the SSC library

9 Legal

The System Advisor Model (“Model”) is provided by the National Renewable Energy Laboratory (“NREL”), which is operated by the Alliance for Sustainable Energy, LLC (“Alliance”) for the U.S. Department Of Energy (“DOE”) and may be used for any purpose whatsoever.

The names DOE/NREL/ALLIANCE shall not be used in any representation, advertising, publicity or other manner whatsoever to endorse or promote any entity that adopts or uses the Model. DOE/NREL/ALLIANCE shall not provide any support, consulting, training or assistance of any kind with regard to the use of the Model or any updates, revisions or new versions of the Model.

YOU AGREE TO INDEMNIFY DOE/NREL/ALLIANCE, AND ITS AFFILIATES, OFFICERS, AGENTS, AND EMPLOYEES AGAINST ANY CLAIM OR DEMAND, INCLUDING REASONABLE ATTORNEYS’ FEES, RELATED TO YOUR USE, RELIANCE, OR ADOPTION OF THE MODEL FOR ANY PURPOSE WHATSOEVER. THE MODEL IS PROVIDED BY DOE/NREL/ALLIANCE "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT SHALL DOE/NREL/ALLIANCE BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO CLAIMS ASSOCIATED WITH THE LOSS OF DATA OR PROFITS, WHICH MAY RESULT FROM ANY ACTION IN CONTRACT, NEGLIGENCE OR OTHER TORTIOUS CLAIM THAT ARISES OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE MODEL.