SSC Reference Manual

January 21, 2013

Abstract

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

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

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

Contents

1 Overview

The SSC software 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, and code samples.

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 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` 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 output variable values from the data container, or error messages from the module.

1.2 Modeling an Energy System

The SSC modules process weather data files, simulate energy system component performance, and calculate project cash flows. Table 1 lists some of the SSC modules.

For a complete list, see?

Creating a technical model of a renewable energy system or a technoeconomic model of a renewable energy project in SSC involves choosing a set of modules. For example, a model of a residential photovoltaic project might use the following modules:

Would this example require a weather reader too?

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

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

Table 1: Some commonly used SSC modules

energy, this module calculates the cost of energy, net present value, and other metrics of the system based on specified system cost, incentives, operation and maintainance costs, taxes, and loan parameters.

In this example, the modules would run in the order shown above, with outputs of each module serving as inputs for the next. Because the modules do not necessarily use the same names or units for variables representing the same quantities, lines of code in the program calling the modules may be required to translate values. For example, the `pvwattsv1` module has an output variable `ac` which is the hourly energy in kWh produced by the system, while the `annualoutput` module requires an input with the name `energy_net_hourly`./marginparMust all modules use the same data container, or can you set up a separate container for each module? It is up to the user to ensure that variables are translated appropriately between the outputs of one module and the inputs of another, including any units conversions that may be necessary. Luckily, SSC provides exhaustive documentation of all of the input variables required by a module and all of the outputs that will be calculated. Working with input and output variables will be the topic of a subsequent chapter./marginparWhere is the documentation of input and output variables?

2 A Basic Example: PVWatts

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

2.1 Program Skeleton

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

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

```
int main(int argc, char *argv[])
{
    if ( argc < 2 )
    {
        printf("usage: pvwatts.exe <weather-file>\n");
        return -1;
    }

    // run PVWatts simulation for the specified weather file

    return 0;
}
```

2.2 Setting up data inputs

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

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

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

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

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

2.3 Simulating and retrieving outputs

Next, an instance of the module itself must be created, per below. The name of the desired module must be passed to the `ssc_module_create()` function. If the specified module name is not

recognized, or the system is out of memory, the function may return NULL.

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

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

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

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

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

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

2.4 Cleaning up

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

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

2.5 Compiling and running

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

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

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

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

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

2.6 Some additional comments

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

3 Data Variables

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

The variable name is mechanism by which data are identified by SSC, and the user is required to supply input variables with names that SSC has predefined. Names can contain letters, numbers, and underscores. Although names can be defined with upper and lower case letters, SSC

does not distinguish between them internally: hence `Beam_Irradiance` is the same variable as `beam_irradiance`.

3.1 Variable Types

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

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

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

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

3.2 Data Types

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

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

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

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

megabytes of computer memory.

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

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

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

3.3 Variable Documentation

As stated before, each module defines all of the input variables it requires and output variables it produces. Since SSC is a model simulation framework that contains within it numerous calculation modules that each may have hundreds of variables, it is impractical to separate to the documentation of variables from their definitions. Otherwise, the documentation would tend to be consistently out of date as modules are updated and new ones are added. Section 4.3 details how to create modules that can be queried for this type of information.

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

Table 2: Variable information provided by SSC

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

To obtain information for a variable, a module is queried using the `ssc_module_var_info()` function, which returns a `ssc_info_t` reference. The `ssc_module_var_info()` function is called repeatedly with an index variable that is incremented by the user to cycle through all of the variables

defined by the module. An example is shown below, assuming that the variable `module` has been successfully created for a particular simulation module (e.g., see §2.3). Querying SSC for available modules is discussed in §4.6.

```
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 );
}
```

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

3.4 Default Values and Constraints

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

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

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

In addition to providing some default values, SSC performs some data input validation before running a simulation using the *constraints* field. This field may optionally define any number of flags recognized by SSC that may limit the valid range of a numeric input variable, or define the required length of an input array. A selection of the flags used by SSC is listed in Table 3. Note that these flags are defined internally by the modules, and are not editable by users. Also, in the case that the *constraints* field is empty, that does not necessarily imply that any values are acceptable - the module writer may have inadvertently omitted a constraint flag, and may instead manually check each variable when the module runs.

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

Table 3: Some constraints recognized by SSC

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

3.5 Manipulating Data Containers

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

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

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

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

To find out the names of all variables in a container, functions ssc_data_first() and ssc_data_next() are used. These functions return the name of a variable, or NULL if there are no more variables in the container. The ssc_data_query() function returns the data type of the specified variable,

which is useful for subsequently retrieving its value. The example below prints out the text of all the string variables in a data container:

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

    if ( type == SSC_STRING )
    {
        const char *text = ssc_data_get_string( data, name );
        printf("string variable %s: %s\n", name, text );
    }

    name = ssc_data_next( data );
}
```

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

3.6 Assigning and Retrieving Values

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

3.6.1 Numbers

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

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

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

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

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

3.6.2 Arrays

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

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

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

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

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

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

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

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

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

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

3.6.3 Matrices

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

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

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

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

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

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

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

if( mat != NULL && nrows == 4 && ncols == 3 )
{
    int r, c, index = 0;
    for( r=0; r<nrows; r++ )
        for( c=0; c<ncols; c++ )
            printf( "mat[%d,%d]=%f\n", r, c, mat[index++] );
}
```

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

3.6.4 Strings

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

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

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

Retrieving strings is similarly straightforward:

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

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

3.6.5 Tables

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

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

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

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

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

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

Retrieving a table is a straightforward operation:

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

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

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

4 Simulation Modules

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

There are four different ways to invoke simulation modules. 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. The four methods are subsequently presented, followed by sections on retrieving error messages and querying the SSC library to learn the names of available compute modules.

4.1 Simple Method #1

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

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

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

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

ssc_data_free( data );
```

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

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

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 method is introduced with the PVWatts example in Section 2.3, and is discussed in more detail in this section.

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 Section 4.5. Alternatively, if the goal is simply to obtain information about the input and output variables associated with a module as in Section 3.3, 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 the graphical SSCdev tool (based on the excellent wxWidgets GUI library) is reproduced below, showing how messages and progess can be reported to the user.

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

        switch( (int)f0 )
```

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

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

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

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

SSCdev is the developer interface for SSC. It 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 SSCdev, it displays all of the variables in the data container with their values.

Use SSCdev for the following tasks:

- 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 the LK script using the SSCdev script editor.

6.1 Using SSCdev

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

1. Run the executable file for your operating system shown in Table 4. For example, the 32-bit Windows executable `sscdev32.exe` is in the `win32` directory of the SDK package.
2. On the **File** menu, click **Choose SSC library**. Be sure to choose the correct library for your version of SSCdev. For example, for 32-bit Windows, choose the `ssc32.dll` library.

   A list of modules should appear in the module browser, and the library path and name should appear in the status bar at the bottom of the SSCdev 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 46 variables.

> **Note:** To see the SSCdev version information, on the **Help** menu, click **About**.

SSCdev 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 SSCdev to help you understand how the modules and variables work.

Operating System	Directory Name	SSCDev Version	Library
Windows 32-bit	win32	sscdev32.exe	ssc32.dll
Windows 64-bit	win32	sscdev64.exe	ssc64.dll
OS X 64-bit	osx64	sscdev.app	ssc64.dylib
Linux 64-bit	linux64	sscdev64.bin	ssc64.so

Table 4: SSCdev and Library Versions

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 is labeled "Percent of annual output." 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, ask a question as a comment on the SAM website's SDK page. 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 SSCdev 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.

6.2 SSCdev Menus

File

Start simulation Runs the active module in the module browser.

Load/unload SSC 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 SSCdev.

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

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

Recent files Display a list of recent data (`.bdat`) and script (`.lk`) files.

Exit Quit SSCdev.

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.

Help

The **About** command displays information about the current version of SSCdev.

6.3 Module Browser

The module browser displays the modules available in the current SSC library. When you choose a module from the list, it also shows a list of the module's variables. It can:

- Display a list of available SSC modules.
- Display module variables.
- Export a list of a module's variables to Excel.
- Run a single module using variable values defined in the data container.

Windows only?

> **Note:** If the module browser is empty, check the status bar at the bottom of the SSCdev window to see whether the correct SSCdev library is loaded. See Section 6.1 for details.

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 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.

6.4 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 current variable indicated with a blue highlight, or on selected variables indicated by a check mark:

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 shown in the module browser.

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

Delete Remove the current variable from the data container.

Del checked Remove all selected variables 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 Select (check) all variables in the data container. This command may take several seconds to complete.

Unselect all Clear all variables in the data container.

Copy to clipboard Copy a tab-delimited list of selected variables and their values to the clipboard.

Show stats

Show stats does not seem to work.

Timeseries graph Display graphs of all selected variables with 8,760 values in the time series data viewer.

> **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.

6.5 Script Editor

The script editor allows you to write, edit, and run scripts in the LK scripting language to build and test models using the SSC modules.

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 (**Ctrl-F**) Find text in the current script.

Find next (**F3**) Find the next instance of text in the current script.

Help Display a list of script functions.

Run (**F5**) Run the current script.

6.6 SSC Dev Example 1: Run a Module without Scripting

The following example shows how to use the module browser to expore and run the textttpvwattsv1 module.

You can use the module browser to explore the pvwattsv1 and annualoutput modules:

Click the pvwattsv1 name in the module browser to see the list of the module's 33 input and output variables. Of those variables, 22 are inputs (SSC_INPUT), and only 6 are required and do not have default values (*).

Now click the annualoutput name to see the module's 11 variables. It requires 6 input values.

This model will use the pvwattsv1 module to read data from a weather file, define the system's size, array orientation and type of tracking, and derating factor, and calcualte values for the 1 by 8760 array storing the system's hourly electricity output over a single year. The annualoutput module will use the array as input, and calculate the system's total annual electricity output over a multi-year period.

In the module browser, click pvwattsv1 to display its variables. The 6 required variables are file_name, system_size, derate, track_mode, azimuth, and tilt. These are the variables that SAM displays on the PVWatts Array input page.

Press the F5 key to start simulations. Nothing happens because you have to check a module to run it. Check the pvwatts1 module and press F5 again. This time, a message about the file_name variable appears in the SSCdev notification panel. That is because file_name is a required variable, but it does not yet have a value.

In the variable viewer, click Add, 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).

Next, add the remaining 5 required variables shown in the module browser, 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 5. This is the model's data container, which so far contains the input and output variables for the pvwattsv1 module.

Press the F5 key. The data container now includes the original input variables from Table 5 and the output variables generated by the simulation. You can show the variable values by checking them. The system's hourly output over a single year is ac, which is an array of 8,760 values.

To add the annualoutput module to the model, on the module browser, check annualoutput. Its name should appear second after pvwattsv1 in the simulation sequence to ensure that it runs after the pvwattsv1 module.

Click the annualoutput module name to display its variables. Of the 6 input variables, only analysis_years is optional, so we must add the 5 required values to the data container. (These input variables are the same ones that SAM displays on the Annual Performance input page.) For example 6 shows the inputs to model the system's performance over 30 years with an annual degradation

have to use script to build a model because variable names do not match (eg, ac does not match energy_bet_net_ho

file_name	system_size	derate	track_mode	azimuth	tilt
F:/SSC/examples/daggett.tm2	4.000000	0.860000	1.000000	180.000000	25.000000

Table 5: The 6 required input variables for the pvwattsv1 module with their values

file_name	system_size	derate	track_mode	azimuth	tilt
F:/SSC/examples/daggett.tm2	4.000000	0.860000	1.000000	180.000000	25.000000

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

of 0.5% and no other adjustments to the system's output. Some of the input variables are arrays because they can store a different value for each year.

Need to make a note that reader must be familiar with SAM to use modules

Now we can expand our model to include two modules, the pvwattsv1 module and the annualoutput module.

In the module browser, check annualoutput. To display its variables, click the module name. It has 6 required inputs, which we have to add to the data container:

analysis_years energy_availability energy_degradation

shading_hourly shading_mxh shading_azal shading_diff

6.7 SSC Dev Example 2: Use a Script to Build a Model

7 Language Interfaces

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

7.1 Python

Python is a popular object-oriented scripting language that is either interpreted directly from source code or compiled into an intermediate bytecode. Most installations of the Python interpreter include the *Ctypes* package, which provides a *foreign function interface (FFI)* capability for directly utilizing dynamic libraries.

The Python language interface has been tested on Windows using the popular ActivePython distribution, as well as on Linux and OSX using the pre-installed 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 souce 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, since 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, peruse 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 SSC MATLAB interface has been tested with MATLAB 7.5 (2007b) 32-bit on Windows. The built-in MATLAB *foreign function interface (FFI)* library routines that are used to interface to SSC require a C compiler to be installed on the computer alongside MATLAB to automatically preprocess the SSC header file. Some versions of MATLAB include a C compiler such as LCC installed by default, while other versions and/or platforms may require one to be installed separately. See `http://www.mathworks.com/support/compilers/R2012b/win64.html` for more information.

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');
```

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, consult the MATLAB code examples supplied with the SSC SDK, as well as the content of the `ssccall.m` file itself.

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. Then, you should be able to create an instance of the `CS_SSC_API` class, as in the example below.

```
private void btnPVWatts_Click(object sender, EventArgs e)
{
    CS_SSC_API.SSC sscobj = new CS_SSC_API.SSC("pvwattsv1");

    txtData.Clear();
```

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

    if ( sscobj.Exec() )
    {
        float[] ac = sscobj.GetArray( "ac" );
        float sum = 0;

        for( int i=0;i<ac.Count();i++)
        {
            sum += ac[i];
        }

        txtData.AppendText("length returned: " + ac.Count() + "\n");
        txtData.AppendText("ac total (get array): " + sum + "\n");
        txtData.AppendText("PVWatts example passed" + "\n");
    }
    else
    {
        txtData.AppendText("PVWatts example failed" + "\n");
    }
}
```

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.

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() )
        {
            int len[] = {0};
            float[] ac = sscobj.Get_Array( "ac", len );
            float sum = 0;
            for( int i=0;i<len[0];i++)
            {
                sum += ac[i];
            }
            System.out.println("length returned: " + len[0]);
            System.out.println("ac total (get array): " + 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**
- #define SSC_INVALID 0
- #define **SSC_STRING** 1
- #define **SSC_NUMBER** 2
- #define **SSC_ARRAY** 3
- #define **SSC_MATRIX** 4
- #define **SSC_TABLE** 5
- #define SSC_INPUT 1
- #define **SSC_OUTPUT** 2
- #define **SSC_INOUT** 3
- #define SSC_LOG 0
- #define SSC_UPDATE 1
- #define SSC_EXECUTE 2
- #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 void ssc_data_set_string (ssc_data_t p_data, const char *name, const char *value)
- SSCEXPORT void ssc_data_set_number (ssc_data_t p_data, const char *name, ssc_number_t value)
- SSCEXPORT void ssc_data_set_array (ssc_data_t p_data, const char *name, ssc_number_t *pvalues, int length)
- SSCEXPORT void ssc_data_set_matrix (ssc_data_t p_data, const char *name, ssc_number_t *pvalues, int nrows, int ncols)
- SSCEXPORT void ssc_data_set_table (ssc_data_t p_data, const char *name, ssc_data_t table)
- SSCEXPORT const char * ssc_data_get_string (ssc_data_t p_data, const char *name)
- SSCEXPORT ssc_bool_t ssc_data_get_number (ssc_data_t p_data, const char *name, ssc_number_t *value)
- SSCEXPORT ssc_number_t * ssc_data_get_array (ssc_data_t p_data, const char *name, int *length)
- SSCEXPORT ssc_number_t * ssc_data_get_matrix (ssc_data_t p_data, const char *name, int *nrows, int *ncols)
- SSCEXPORT ssc_data_t ssc_data_get_table (ssc_data_t p_data, const char *name)
- SSCEXPORT ssc_entry_t ssc_module_entry (int index)
- SSCEXPORT const char * ssc_entry_name (ssc_entry_t p_entry)
- SSCEXPORT const char * ssc_entry_description (ssc_entry_t p_entry)
- SSCEXPORT int ssc_entry_version (ssc_entry_t p_entry)
- SSCEXPORT ssc_module_t ssc_module_create (const char *name)
- SSCEXPORT void ssc_module_free (ssc_module_t p_mod)
- SSCEXPORT const ssc_info_t ssc_module_var_info (ssc_module_t p_mod, int index)
- SSCEXPORT int ssc_info_var_type (ssc_info_t p_inf)
- SSCEXPORT int ssc_info_data_type (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_name (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_label (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_units (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_meta (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_group (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_required (ssc_info_t p_inf)
- SSCEXPORT const char * ssc_info_constraints (ssc_info_t p_inf)

- SSCEXPORT const char * ssc_info_uihint (ssc_info_t p_inf)
- SSCEXPORT ssc_bool_t ssc_module_exec_simple (const char *name, ssc_data_t p_data)
- SSCEXPORT const char * ssc_module_exec_simple_nothread (const char *name, ssc_data_t p_data)
- SSCEXPORT ssc_bool_t ssc_module_exec (ssc_module_t p_mod, ssc_data_t p_data)
- SSCEXPORT ssc_bool_t ssc_module_exec_with_handler (ssc_module_t p_mod, ssc_data_t p_data, ssc_bool_t(*pf_handler)(ssc_module_t, ssc_handler_t, int action, float f0, float f1, const char *s0, const char *s1, void *user_data), void *pf_user_data)
- SSCEXPORT void ssc_module_extproc_output (ssc_handler_t p_mod, const char *output_line)
- SSCEXPORT const char * ssc_module_log (ssc_module_t p_mod, int index, int *item_type, float *time)
- SSCEXPORT void **__ssc_segfault** ()

8.1.1 Detailed Description

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

Note

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

Copyright

Authors

Aron Dobos, Steven Janzou

------— Doxygen notes -------—

Detailed description of what the function does with any useful tips or warnings about using it effectively.

Description can be one or more paragraphs. This description has three paragraphs

Only the brief description is required.

Returns

Description of function's return value.

Parameters

	Descripton	of function parameter.
`in`	*P1*	Description of input paramter P1.
`in,out`	*P2*	Description of parameter P2 that is input and output.
`out`	*P3*	Description of P3 output parameter.

Note

Note for important warnings.

```
sample code line 1
  sample code line 2
sample code line 3
...
```

8.1.2 Macro Definition Documentation

8.1.2.1 #define SSC_EXECUTE 2

request to run an external executable. f0: unused, f1: unused, s0: command line, s1: working directory

Note

On an external executable request, the handler function should:

1. save the current working directory
2. switch to the desired working directory (passed in arg2)
3. run the specified command synchronously, redirecting the standard output stream of the child process
4. for each line of output from the child process, call ssc_module_extproc_output(..) passing the computation module reference and the process output text
5. when the process is done executing, restore the previously used working directory
6. return 1 if everything went fine, or 0 if there was an error executing the child process

8.1.2.2 #define SSC_INPUT 1

Variable types include SSC_INPUT SSC_OUTPUT SSC_INOUT

8.1.2.3 #define SSC_INVALID 0

Possible data types for variables in an ssc_data_t.

8.1.2.4 #define SSC_LOG 0

action/notification types that can be sent to a handler function SSC_LOG SSC_UPDATE SSC-_EXECUTE log a message. f0: (int)message type, f1: time, s0: message text, s1: unused

8.1.2.5 #define SSC_NOTICE 1

Messages types SSC_NOTICE SSC_WARNING SSC_ERROR

8.1.2.6 #define SSC_UPDATE 1

notify simulation progress update. f0: percent done, f1: time, s0: current action text, s1: unused

8.1.3 Typedef Documentation

8.1.3.1 typedef int ssc_bool_t

The boolean type used internally in SSC.

Zero values represent false; non-zero represents true.

8.1.3.2 typedef void* ssc_data_t

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

This structure can contain any number of variables referenced by name, and can hold strings, numbers, arrays, and matrices. Matrices are stored in row-major order, where the array size is nrows*ncols, and the array index is calculated by r*ncols+c. An ssc_data_t object holds all input and output variables for a simulation. It does not distinguish between input, output, and input variables - that is handled at the model context level.

8.1.3.3 typedef void* ssc_entry_t

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

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

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

8.1.3.4 typedef void* ssc_handler_t

An opaque pointer for transferring external executable output back to SSC

8.1.3.5 typedef void* ssc_info_t

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

8.1.3.6 typedef void* ssc_module_t

An opaque reference to a computation module.

Note

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

8.1.3.7 typedef float ssc_number_t

The numeric type used internally in SSC.

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.

8.1.4 Function Documentation

8.1.4.1 SSCEXPORT const char* ssc_build_info ( )

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

Returns

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

8.1.4.2 SSCEXPORT void ssc_data_clear (ssc_data_t *p_data*)

Clears all of the variables in a data object.

8.1.4.3 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.

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

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

8.1.4.5 SSCEXPORT void ssc_data_free (ssc_data_t *p_data*)

Frees the memory associated with a data object.

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

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

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

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

Note

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

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

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

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

Note

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

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

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

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

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

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

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

Returns

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

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

Assigns value of type *SSC_ARRAY*

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

Assigns value of type *SSC_MATRIX*

Note

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

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

Assigns value of type *SSC_NUMBER*

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

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

    key = ssc_data_next( my_data );
}
```

Note

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

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

Assigns value of type *SSC_TABLE*.

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

Unassigns the variable with the specified name.

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

Returns a short text description of a compute module.

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

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

8.1.4.21 SSCEXPORT int ssc_entry_version (ssc_entry_t *p_entry*)

Returns version information about a compute module.

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

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

8.1.4.23 SSCEXPORT int ssc_info_data_type (ssc_info_t *p_inf*)

Returns the data type of a variable: SSC_STRING, SSC_NUMBER, SSC_ARRAY, SSC_MATRIX, SSC_TABLE

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

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

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

Returns the short label description of the variable

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

Returns any extra information about a variable

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

Returns the name of a variable

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

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

'.

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

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

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

Returns the units of the values for the variable

8.1.4.31 SSCEXPORT int ssc_info_var_type (ssc_info_t *p_inf*)

Returns variable type information: SSC_INPUT, SSC_OUTPUT, or SSC_INOUT

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

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

Returns

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

8.1.4.33 SSCEXPORT ssc_entry_t ssc_module_entry (int *index*)

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

Returns

0 (NULL) for an invalid index.

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

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

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

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

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

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

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

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

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

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

8.1.4.39 SSCEXPORT void ssc_module_free (ssc_module_t *p_mod*)

Releases an instance of a compute module created with ssc_module_create

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

Retrive notices, warnings, and error messages from the simulation

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

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

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

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

Returns

Returns NULL for invalid index.

Note

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

8.1.4.42 SSCEXPORT int ssc_version ( )

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

Returns

SSC version number.