**Documentation for Running TCS from SAM**

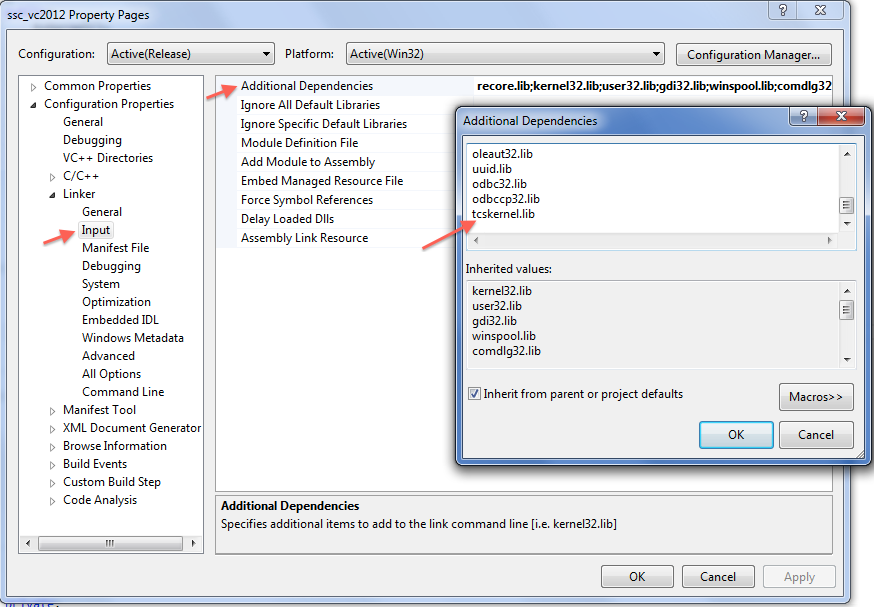
The method to call TCS models from SAM described in this document is easy to use, took very little time to implement, and requires no changes to the SAM code in the samwx SVN. This document briefly describes the methodology used and explains how a programmer would go about creating a new model. It assumes an understanding of the SAM code and the SSC interop layer.

### Methodology

Tom tested several methods for calling TCSKernal library functions from SAM, but ultimately settled on the simplest method he tried. Using this method saved development time, which seemed desirable since this was not meant to be a permanent solution for the chosen capability. All of the changes described are implemented in a new branch from the SSC project (“efmsvn.nrel.gov/ssc/svn/branches/tcsapi”) in the SVN. So the changes below refer to changes from the original SSC Visual C++ (VC) project as implemented in the branch project. (**Note that the branch has not been updated since April 7th because Tom has not had access to the SVN since then.**)

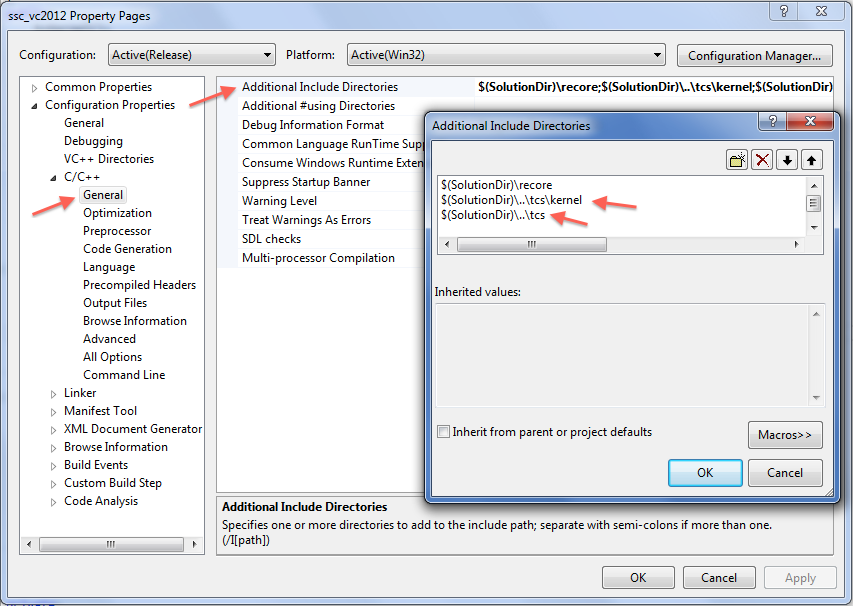
The “Target Name” and “Target Extension” properties of the VC project in SVN have not been modified. Thus, this project will create an ssc.dll that can replace the ssc.dll from the SSC project in the SVN. However, since this project uses the TCS kernel, it requires access to the *tcskernel.lib* file – so this file also has to be placed in the same folder with the new ssc.dll file. Additionally, the tcskernel relies on having access to a type library. For these tests, I used the “typelib” library compiled in the tcs\_*vc2012* VC solution – which requires access to *typelib.dll* and *waterprop32.dll*.

In order to use the TCS functions, Tom modified the original SSC VC project file to have “Additional Dependencies” and included “*tcskernel.lib*” in the list:



This file then needs to be available to the VC project when compiling. Either place a copy in the VC project “Output Directory”, or make sure the VC project will look for it in the correct place.

To include the TCS kernel header files (*tcstype.h* and *tcskernel.h*), Tom added two “Additional Include Directories”:



Finally, to simplify and standardize the method by which programmers can interact with the TCS, Tom created a wrapper class, based on the “tcKernel” class in the “*tcsmain\_vc2012*” project in the “*tcs\_vc2012*” VC solution. This class is derived from both the “tcskernel” class and the “compute\_module” class. The complete code for the header file is listed below:

#ifndef \_\_tckernel\_h

#define \_\_tckernel\_h

#include "tcskernel.h"

#include "core.h"

class tcKernel : public tcskernel, public compute\_module

{

public:

tcKernel();

virtual ~tcKernel();

virtual bool converged( double time );

void set\_store\_array\_matrix\_data( bool b ) { m\_storeArrMatData = b; }

virtual int simulate( double start, double end, double step );

void set\_unit\_value\_ssc\_string( int id, const char \*name );

void set\_unit\_value\_ssc\_double( int id, const char \*name );

void set\_unit\_value\_ssc\_double( int id, const char \*name, double x );

void set\_unit\_value\_ssc\_array( int id, const char \*name );

void set\_unit\_value\_ssc\_matrix( int id, const char \*name );

void set\_output\_array(const char \*output\_name, size\_t len);

void set\_output\_array(const char \*ssc\_output\_name, const char \*tcs\_output\_name, size\_t len);

struct dataitem {

dataitem( const char \*s ) : sval(s) { }

dataitem( const std::string &s ) : sval(s) { }

dataitem( double d ) : dval(d) { }

std::string sval;

double dval;

};

struct dataset {

unit \*u;

int uidx;

int idx;

std::string name;

std::string units;

std::string group;

int type;

std::vector<dataitem> values;

};

dataset \*get\_results(int idx);

private:

bool m\_storeArrMatData;

std::vector< dataset > m\_results;

double m\_start, m\_end, m\_step;

size\_t m\_dataIndex;

};

#endif

The main difference between this tcKernel class and the one used in the TCS VC project are the functions within the red rectangle. These functions will make the code in the individual technology modules cleaner and easier to write. Tom expects that this class will be changed as needed. Other than those additions, this class performs the same main functions as the tcKernel class on which it is based: keeping results, controlling the simulation, and reporting progress.

The above changes are the extent of the code changes necessary to implement this method of calling TCS models from SAM.

### Example technology

What follows is an example of creating a module that uses the TCS kernel to run a simulation from SAM using the empirical trough technology.

The programmer must first create a new module in the TCS SSC in the same way one is created in the SSC. First, add a new entry point in the sscapi.cpp file:

extern module\_entry\_info

/\* extern declarations of modules for linking \*/

cm\_entry\_6parsolve,

. . .

cm\_entry\_windpower,

cm\_entry\_geothermalui,

cm\_entry\_tcstrough;

/\* official module table \*/

static module\_entry\_info \*module\_table[] = {

&cm\_entry\_6parsolve,

. . .

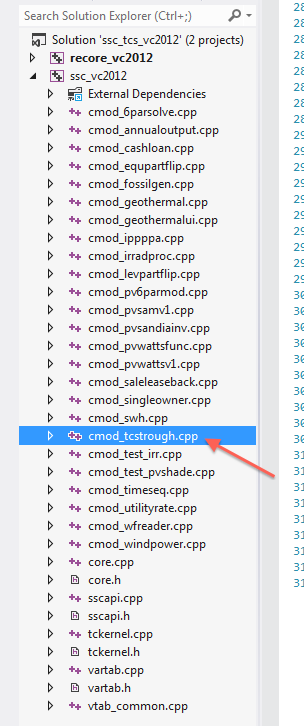
&cm\_entry\_windpower,

&cm\_entry\_geothermalui,

&cm\_entry\_tcstrough,

0 };

Next, define the module by creating a new class and its variable table, and defining its entry using the “DEFINE\_MODULE\_ENTRY” macro. This is usually done in a separate file named *cmod\_XXXXX.cpp* where the XXXXX represents the name of the module. In this example, I added a file called *cmod\_tcstrough.cpp*:



This file should be very similar to the module definition file used for the SSC technologies. The differences appear below:

1. When defining the variable table, it will be very helpful to make the variable names match those of the parameters defined in the type library used by the tcskernel. This way the programmer can avoid having to translate between the variable name used in the SSC to communicate with SAM and the variable name defined in the units of the type library.
2. The class for the module **has to be derived from the** tcKernel **wrapper class** (instead of the compute\_module class) in order to use the TCS capabilities:

//Definition of class used to interact with the TCS empirical trough model

class cm\_tcstrough : public tcKernel

{

public:

. . .

1. The constructor for the class should add a value to the tcskernel search path (using “add\_search\_path()”), in addition to calling the “add\_var\_info()” function. This tells the tcskernel object where to look for type libraries. For my tests, I created a copy of the typelib dll file in the sam deploy directory, the same directory as the ssc dll file. As such, I only added a single search directory – the current directory:

//Constructor for the cm\_tcstrough class

cm\_tcstrough()

{

add\_var\_info( \_cm\_vtab\_tcstrough );

add\_search\_path( "." );

}

1. The first order of business in the exec() function should be to load the type library into the tcskernel using the “load\_library()” function, or return an error if it cannot be loaded:

void exec( ) throw( general\_error )

{

if ( 0 >= load\_library("typelib") )

throw exec\_error( "tcstrough", util::format("could not load the tcs type library.") );

//Body of exec code follows...

. . .

1. Finally, the body of the exec function can now use both native tcskernel functions (e.g. “add\_unit”) and functions defined in the tcKernel class (e.g. “set\_unit\_value\_ssc\_array”) to interact with the TCS simulation class. Below is a partial listing of the *cmod\_tcstrough.cpp* file showing some of these functions:

//Add weather file reader unit

int weather = add\_unit("weatherreader", "TCS weather reader");

//Add Empirical Solar Field Model

int u1 = add\_unit( "sam\_trough\_model\_type805", "Test Trough" );

. . .

//Set weatherreader parameters

set\_unit\_value\_ssc\_string( weather, "file\_name" );

. . .

//Connect Solar Field Inputs

bool bConnected = connect( weather, "solazi", u1, "SolarAz", 0.1, -1 );

bConnected &= connect( weather, "beam", u1, "Insol\_Beam\_Normal", 0.1, -1 );

bConnected &= connect( weather, "tdry", u1, "AmbientTemperature", 0.1, -1 );

bConnected &= connect( weather, "wspd", u1, "WndSpd", 0.1, -1 );

. . .

// check if all connections worked

if ( !bConnected )

throw exec\_error( "tcstrough", util::format("there was a problem connecting units in the simulation.") );

// Run simulation

size\_t hours = 8760;

if (0 > simulate(3600, hours\*3600, 3600) )

throw exec\_error( "tcstrough", util::format("there was a problem simulating in tcstrough.") );

. . .

// get the outputs

set\_output\_array("enet", hours);

set\_output\_array("egr", hours);

Notice that the “simulate” function expects seconds as inputs. The lk scripts and TCS GUI program translate hours into seconds automatically. It must be done explicitly here – this allows for the possibility of sub hourly simulations.

At this point in the development, there are 7 helper functions defined in the tcKernel class specifically created to help programmers transfer data to/from the TCS kernel from/to the SSC interface. There are five for translating inputs from SAM into inputs for the units defined in the TCS, and two to help retrieve results from the TCS and send them back to SAM. The declarations of 4 of the functions follow:

void set\_unit\_value\_ssc\_string( int id, const char \*name );

void set\_unit\_value\_ssc\_double( int id, const char \*name );

void set\_unit\_value\_ssc\_array( int id, const char \*name );

void set\_unit\_value\_ssc\_matrix( int id, const char \*name );

These functions all follow the same format. Basically, they will look for an input in the TCS unit defined by “id” with the name “\*name” and fill it with the SSC\_INPUT variable with the same name. If the SSC variable type doesn’t match the type expected by the function, the SSC code will throw an error.

The other helper function for inputs has a slightly different format:

void set\_unit\_value\_ssc\_double( int id, const char \*name, double x );

This function was created to help with debugging – it makes it easy to set an exact value for a TCS unit input. It can be used like the “set\_value” function in the lk scripts used to run TCS.

The last two helper functions get data from the TCS results stored in the tcKernel class and translate them into SSC\_OUTPUT arrays to be sent back to SAM:

void set\_output\_array(const char \*output\_name, size\_t len);

void set\_output\_array(const char \*ssc\_output\_name, const char \*tcs\_output\_name, size\_t len);

The first function assumes the name of the TCS value and the SSC value are the same, it will look for a TCS\_NUMBER type parameter in any of the units matching “name” and set the SSC output array with the same name to that value for each step of the simulation. The second function serves the same purpose, but allows a programmer to use different names for the TCS parameter and the SSC array.

### Within SAM

The process for adding a technology that uses this method for running a TCS simulation from within SAM is the same as the process for adding any SSC interop model. This process is covered in the SSC documentation.