Writing a Simulink Device Driver block: a step by step guide

This document explains, in a step by step fashion, how to create "device driver blocks" that is, blocks that perform target specific functions when executed on the target OS or platform. The Arduino Support from Simulink package is used to build the examples, but the method is the same for any other supported target.

What is a device driver block?

In general, an "Output Device Driver" block takes some signal as input and uses it to perform some kind of actuation on the target platform (e.g. analog or digital write) either directly with the hardware or through the real-time operating system.

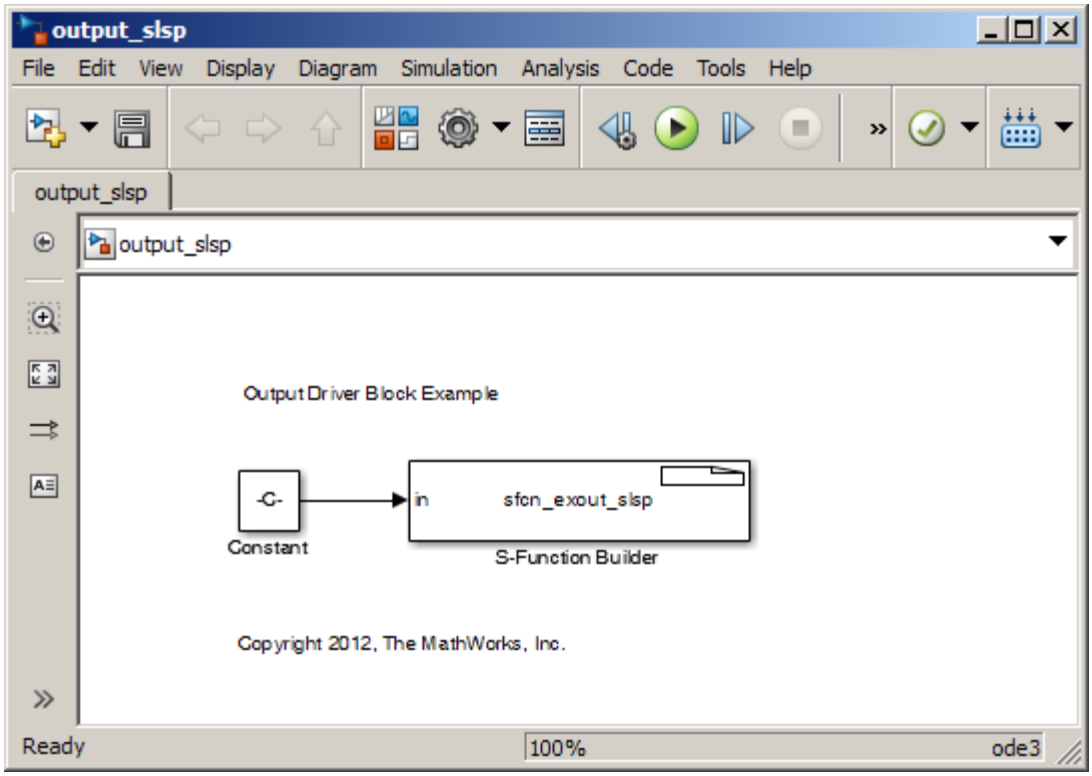

Figure 1: Model containing a custom output driver block

The model in Figure 1 contains a simple output driver block implemented through an S-Function Builder block, which takes in the constant "1" as input.

An “Input Device Driver” block is instead a block that performs some kind of sensing on the target platform and makes the result available for computation (e.g. analog or digital read):

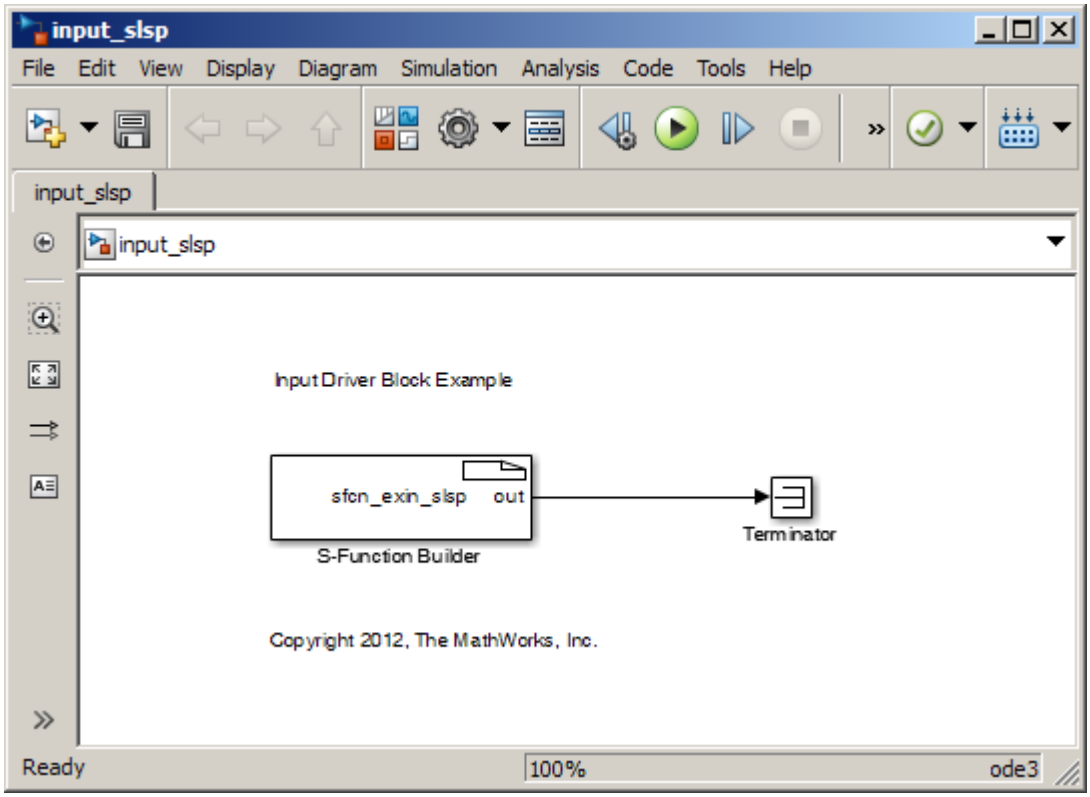

Figure 2: Model containing a custom input driver block

In this guide, we will make extensive use of the S-Function Builder block, which is found in the Simulink Library under “User-Defined Functions”, and allows you to generate S-Functions (specifically, a wrapper C file is generated which contains only header information and two functions) without writing a C file that explicitly interacts with their API. The wrapper file is then used to create executable files for simulation and on-target execution.

It might be a good idea to have a look at the help page for the S-Function Builder dialog box.

A quick look at the selected target

Before explaining the details of driver blocks, it is worthwhile noticing that when Simulink Coder is installed, a look at "Model Configuration Parameters -> Code Generation" shows (and allows changes to) the selected target.

If the Simulink Support Package for Arduino is installed then one needs to select "Tools -> Run on Target Hardware -> Prepare to Run" to specify some board-related configuration parameters that are needed to upload and launch the executable. Selecting "Tools -> Run on Target Hardware -> Options…" allows changing said parameters after they have been selected.

In any case, the method shown in this document is not specific to this particular target configuration and will guide you towards building your own driver block independent of the chosen target.

Simulation vs. "on target" execution

It is important to understand that models such as the ones in the previous figures could be executed in two different ways:

First, they can be simulated (this happens when the green "Play" or "Run" button in the Tool Strip is pressed). When a model is simulated, it is executed on your computer (as a matter of fact it is executed by the Simulink engine as a part of the MATLAB process).

In order to execute the S-Function Builder block, Simulink calls the block's MEX (MATLAB Executable) file. This file is generated from the C-code written in the S-Function block when the button "build" of the S-Function dialog box is pressed.

Note that in general a driver block does *not* perform *any* operation *in simulation*.

For example, when the MEX-file generated from the S-Function Builder block in Figure 2 is called, it does not do anything, and the output signal that goes to the terminator always remains at its default initial value of 0.

Similarly, when the MEX-file generated from the S-Function Builder block in Figure 1 is called, it does not do *anything* with the value received in its input. In other words, one should not expect anything to happen (not on the computer, let alone on the target hardware), when the model is *simulated* (e.g., in this case, no LED on the Arduino can light up when the model in Figure 1 is simulated).

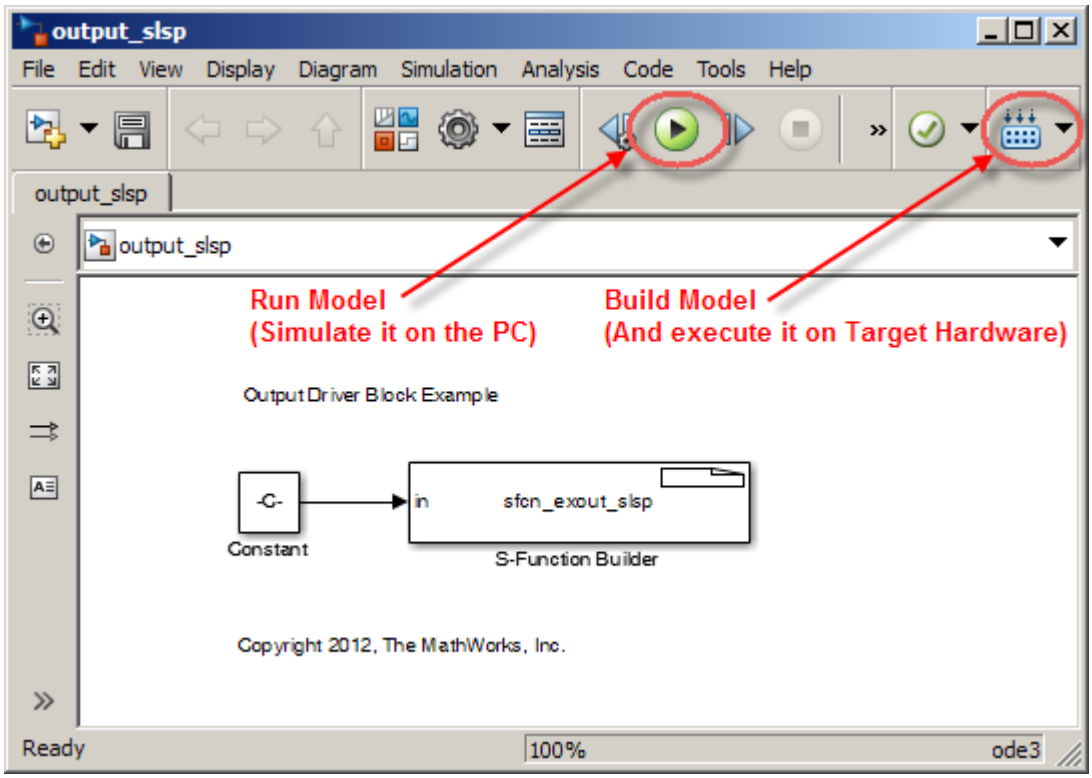

Figure 3: Two different ways of executing the model

The other way in which a model like the ones in the previous pictures can be executed is by generating (from the model) an executable that runs (typically in real time) on the target platform.

If Simulink Coder is installed, this happens when one presses the "Build Model" button shown in the upper right corner in Figure 3. Note that both the keyboard shortcut "Ctrl-B" and the MATLAB command "rtwbuild" have exactly the same effect as the build button.

If Simulink Coder is not installed, then the build button will not be there, but you can use (after a relatively quick automatic installation procedure) the "Tools ->

Run on Target Hardware -> Run" feature, which allows for a similar functionality for a few selected target boards (see Figure 4).

In general, for embedded software integration tasks such as creating device driver blocks, the ability to examine, debug, and optimize the C-code is highly desirable. Therefore, it is generally preferable to have both Simulink Coder and Embedded Coder when authoring device driver blocks.

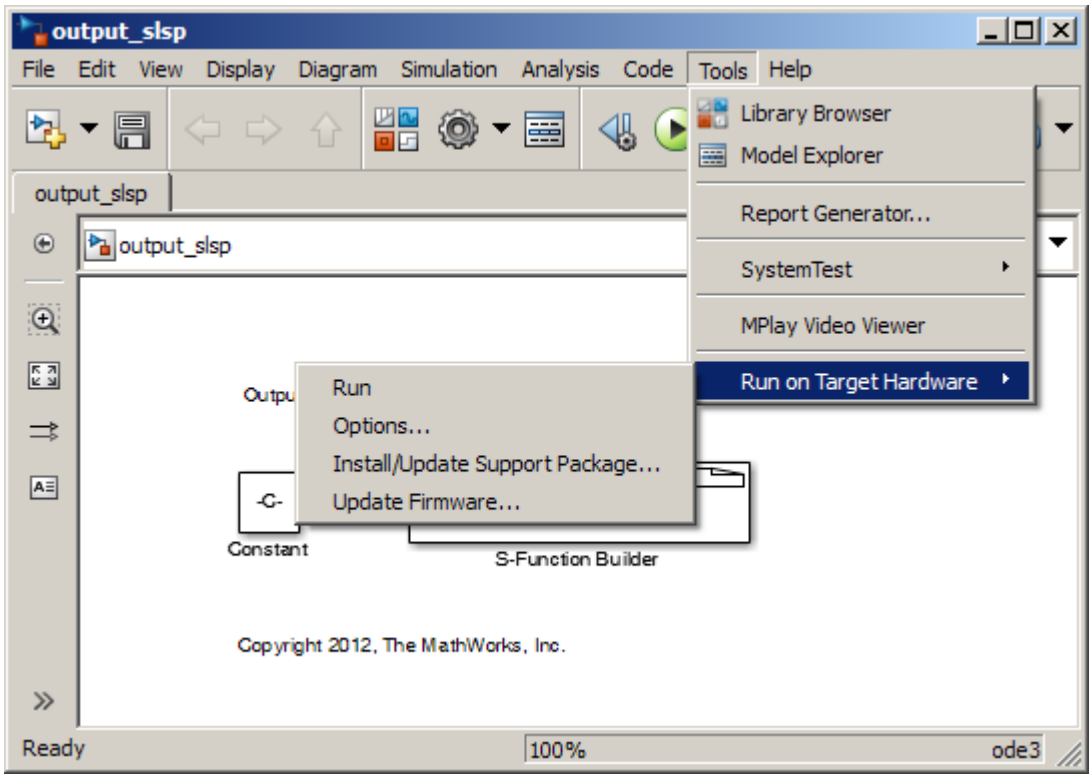

Figure 4: Run on Target Option

Unlike what happens when you simulate the model, when the executable runs *on the target platform*, one typically needs the device driver blocks to actually *do something*.

In this case, we want the S-Function Builder block in Figure 2 to actually perform a sensing operation, (i.e. a digital read on pin 4). Similarly, when executed on the target platform, we want the S-Function Builder block in Figure 1 to actually perform some actuation (i.e. a digital write on pin 12, which if everything is connected correctly will light up an LED connected between pin 12 and ground).

Output driver block: Data Properties pane

To begin, create a new Simulink Model and add the S-Function Builder block.

The next step is to select the right "Target" for the model. You can do this by either selecting "Tools -> Run on Target Hardware -> Prepare to Run" (if the relevant Support Package is installed) or, "Model Configuration Parameters -> Code Generation" (if Simulink Coder is installed).

Since typically models that execute on embedded targets are very simple and don't need to accurately integrate a differential equation, it might be also a good idea to open the Solver page of the Configuration Parameters (Simulation -> Configuration Parameters) and the solver to "Discrete (no continuous state" (if your model does not have any integrators). Changing the Tasking Mode to "Single Tasking" (if you don't have multiple sample times) imposes further simplifications, so it might be a good idea as well.

Ideally the Solver pane for the model should look like Figure 5 below:

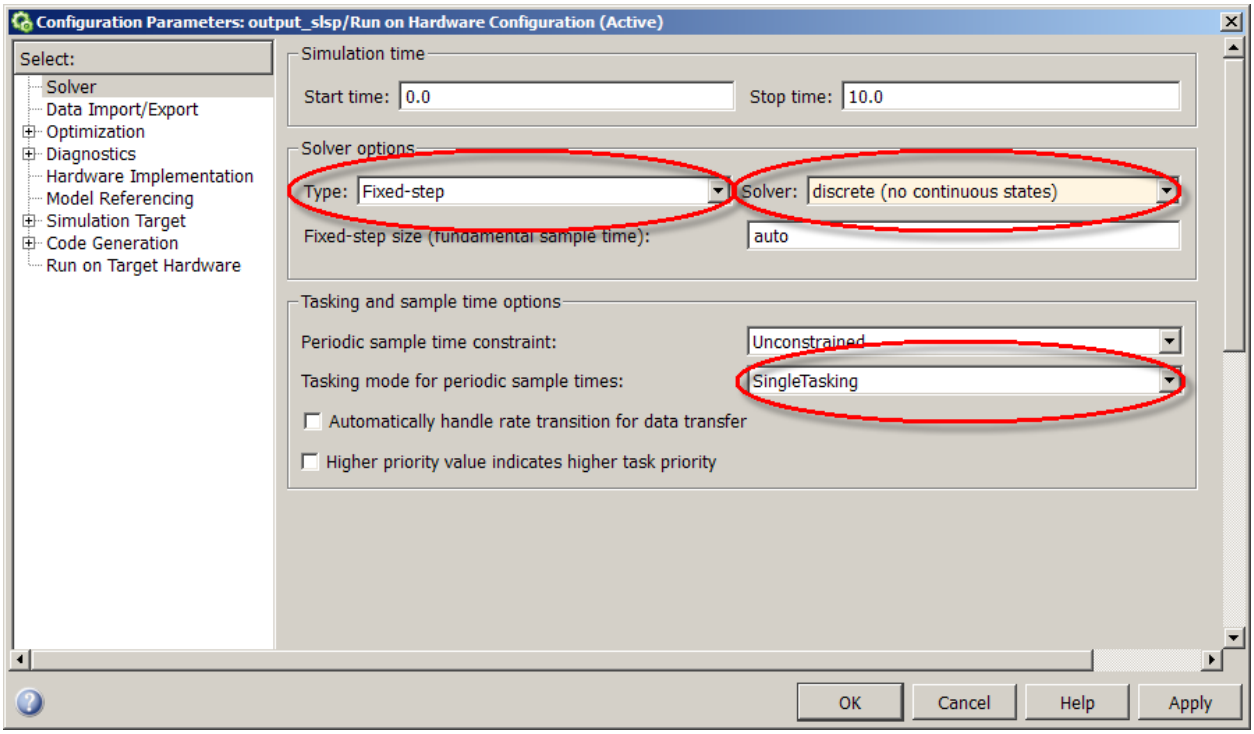

Figure 5: Solver Configuration Parameters

The first thing to do is to give a name to the S-Function, in this case I choose "sfcn_exout_slsp" (for S-Function-example-output-Simulink-support-package).

The first pane is the initialization pane, which we'll cover later, for now let's start from the second one, that is the Data Proprieties pane, which allows us to define the number and dimensions of input and output ports, as well as the parameters passed to the S-Function block.

The default S-Function block has one input port named u0, one output port named y0, and no parameters. By clicking on the "Output ports" subpane of the "Data Properties" pane we can delete the output port (because we want this to be a "sink" block) as shown in Figure 6:

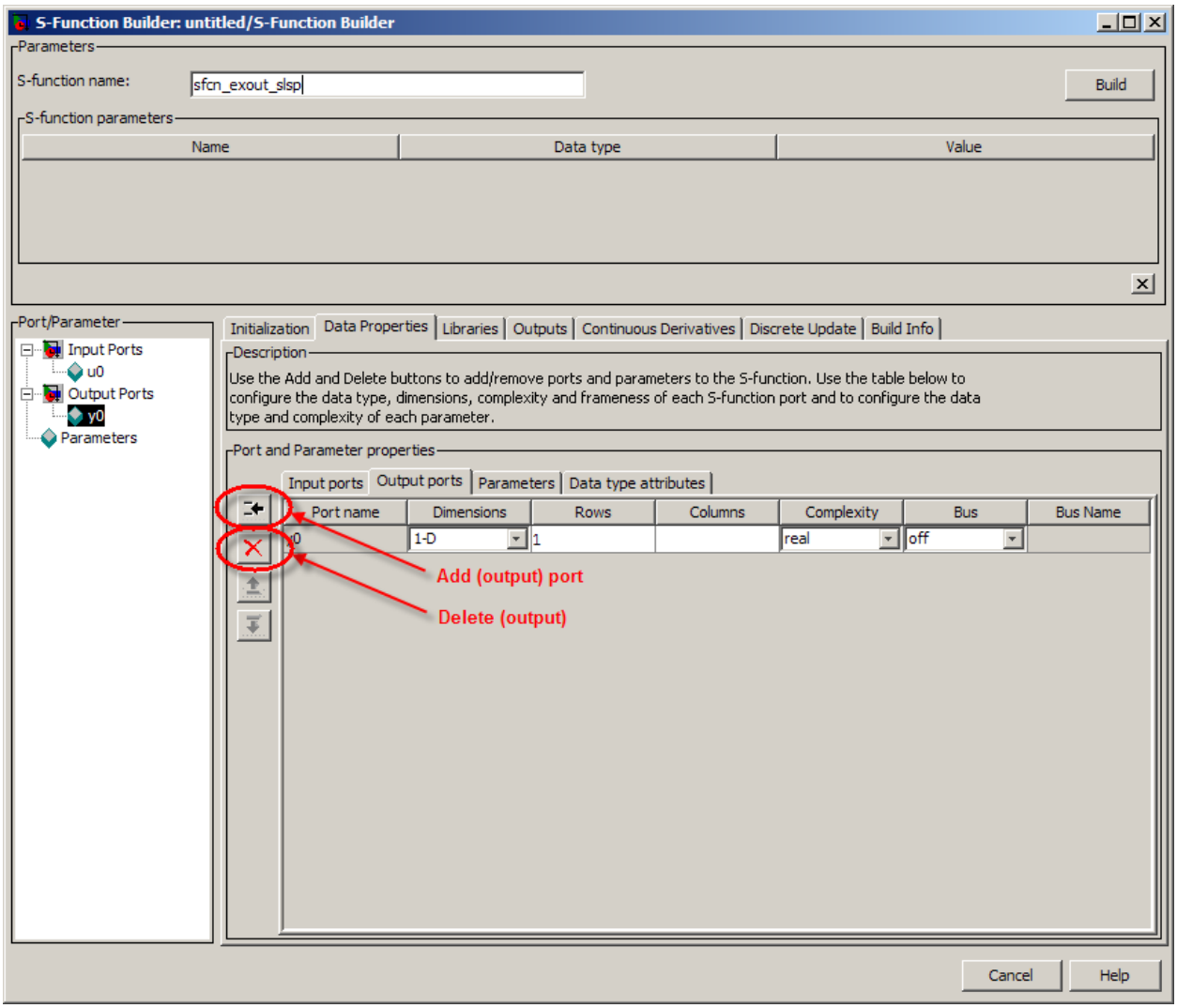

Figure 6: Data Properties pane: Output ports

Similarly, by clicking on the “Input ports” subpane we can rename the input port to “in” instead of u0. The value coming from this input port will be later referred to as in[0] in the code.

Clicking on the “Data type attributes” subpane also allows us to change the data type of the input from double to boolean (this needs to be an digital output block, so a boolean value is what is needed as input).

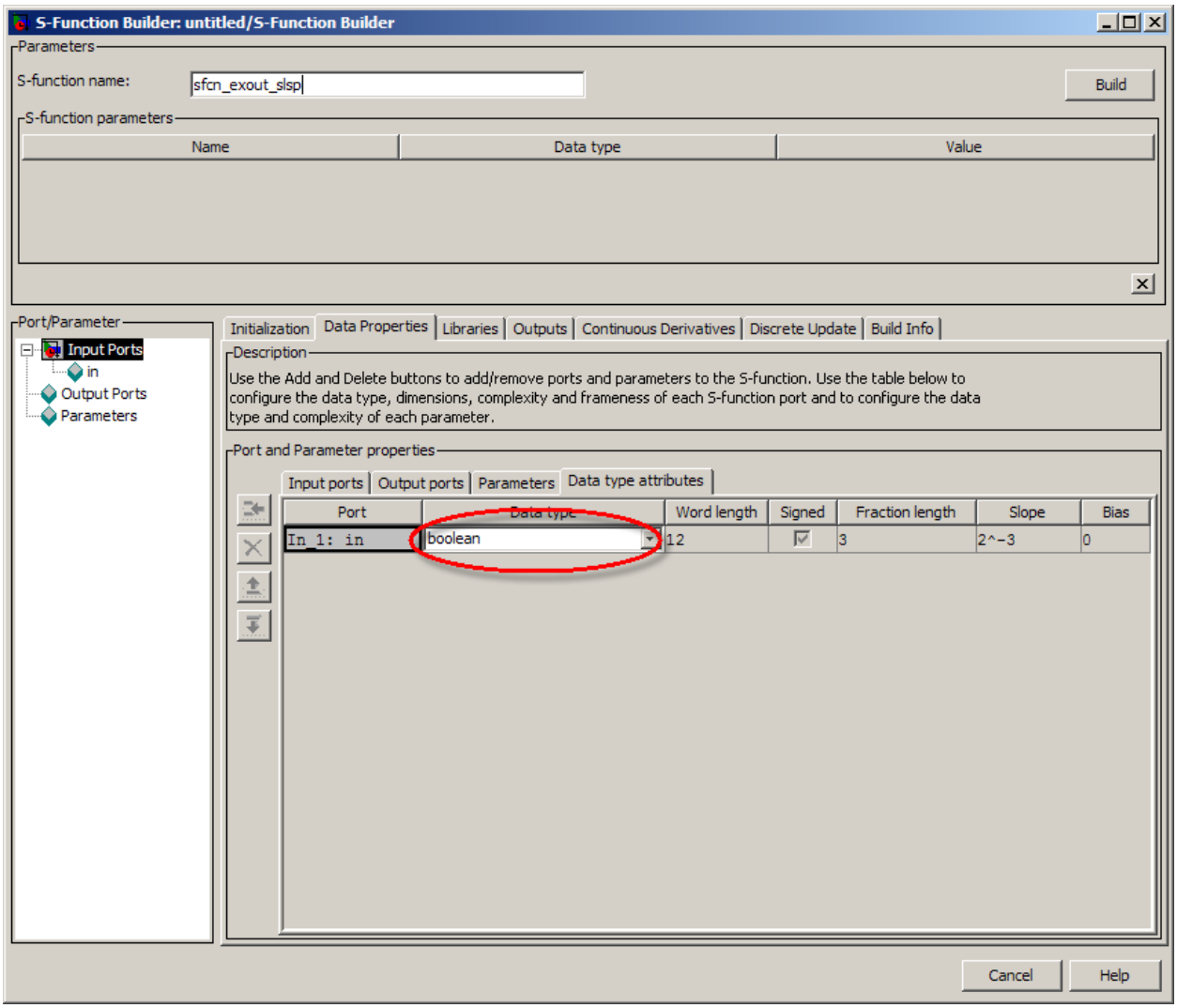

Figure 7: Data Properties pane: Data Type Attributes

The “Parameters” subpane allows us to insert a parameter (using the “Add” button on the left side). We insert a parameter named “pin”, and define its type as an unsigned 8-bit integer (uint8), as shown in Figure 8.

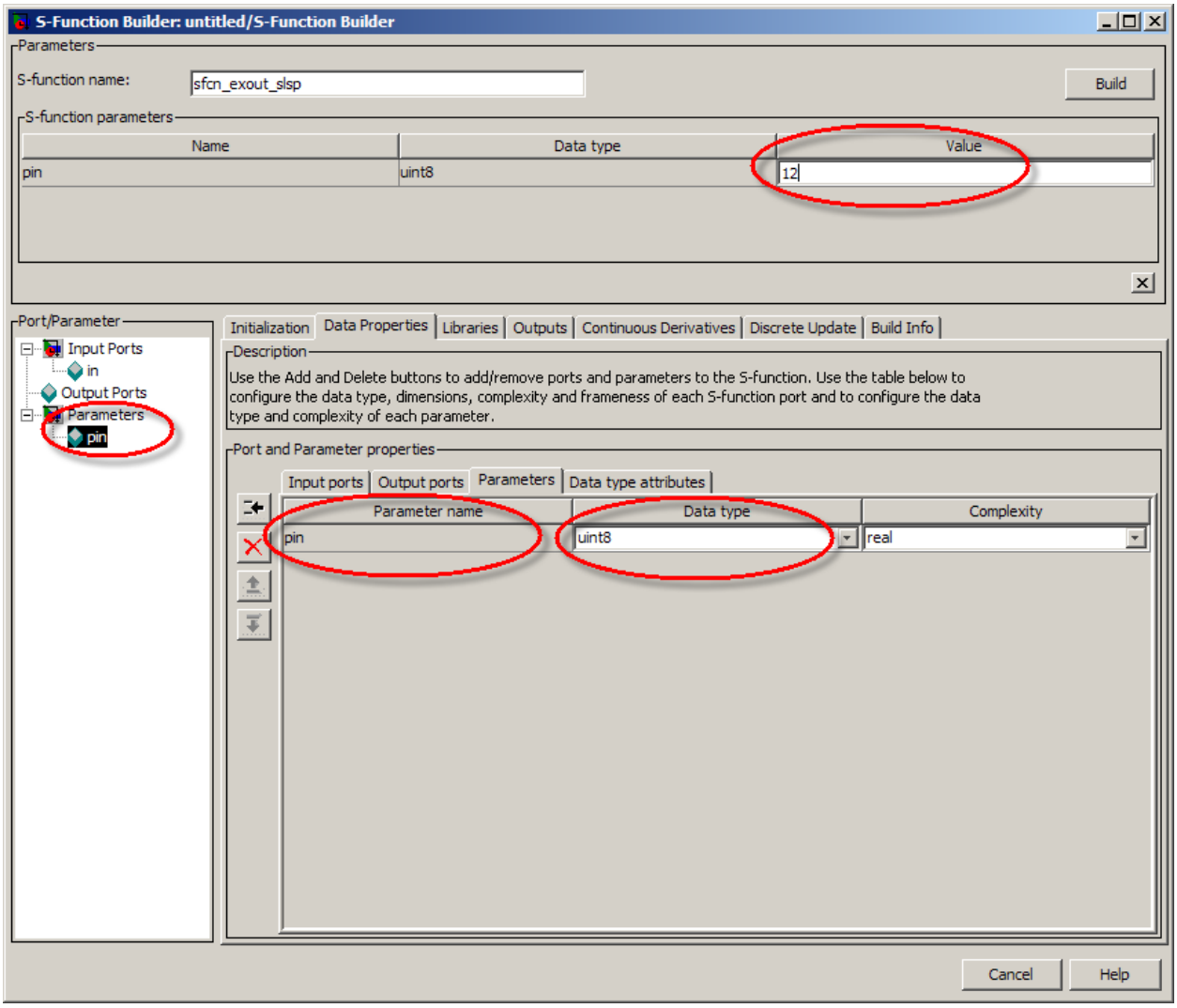

Figure 8: Data Properties pane: Parameters

The actual value of the parameter is passed by the S-Function dialog box (in the upper part of the S-Function Builder GUI). In this case a value of 12 has been selected, which means we want to perform digital output on pin #12.

It’s important to note that if any value in the parameter dialog box is changed then the S-Function needs to be built again for the change to take effect. This is why it might actually be a good idea to use inputs (instead of mask parameters) to carry values that need to be changed relatively often.

Output driver block: Build Info pane

The last (rightmost) pane is called “Build Info” and is shown in Figure 9 below:

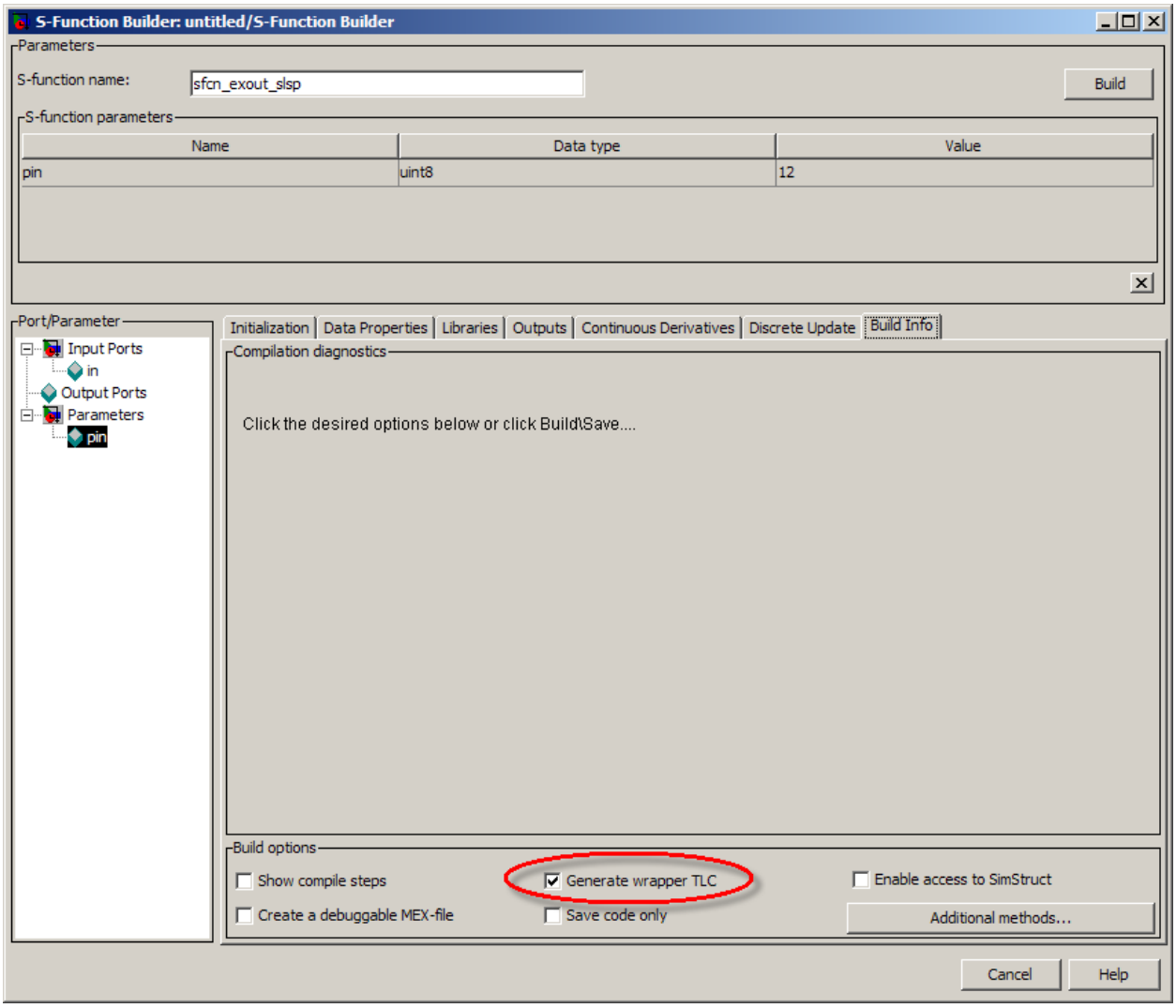

Figure 9: Build Info pane of the output driver block

The “Generate wrapper TLC” checkbox must be checked. This will generate the TLC file which will then be used to build the executable that will run on the target.

On the other hand, the “Enable access to SimStruct” check should be left unchecked (unless you really need it) because it might prevent the block from working on targets that do not support non-inlined s-funcions.

Output driver block: Initialization pane

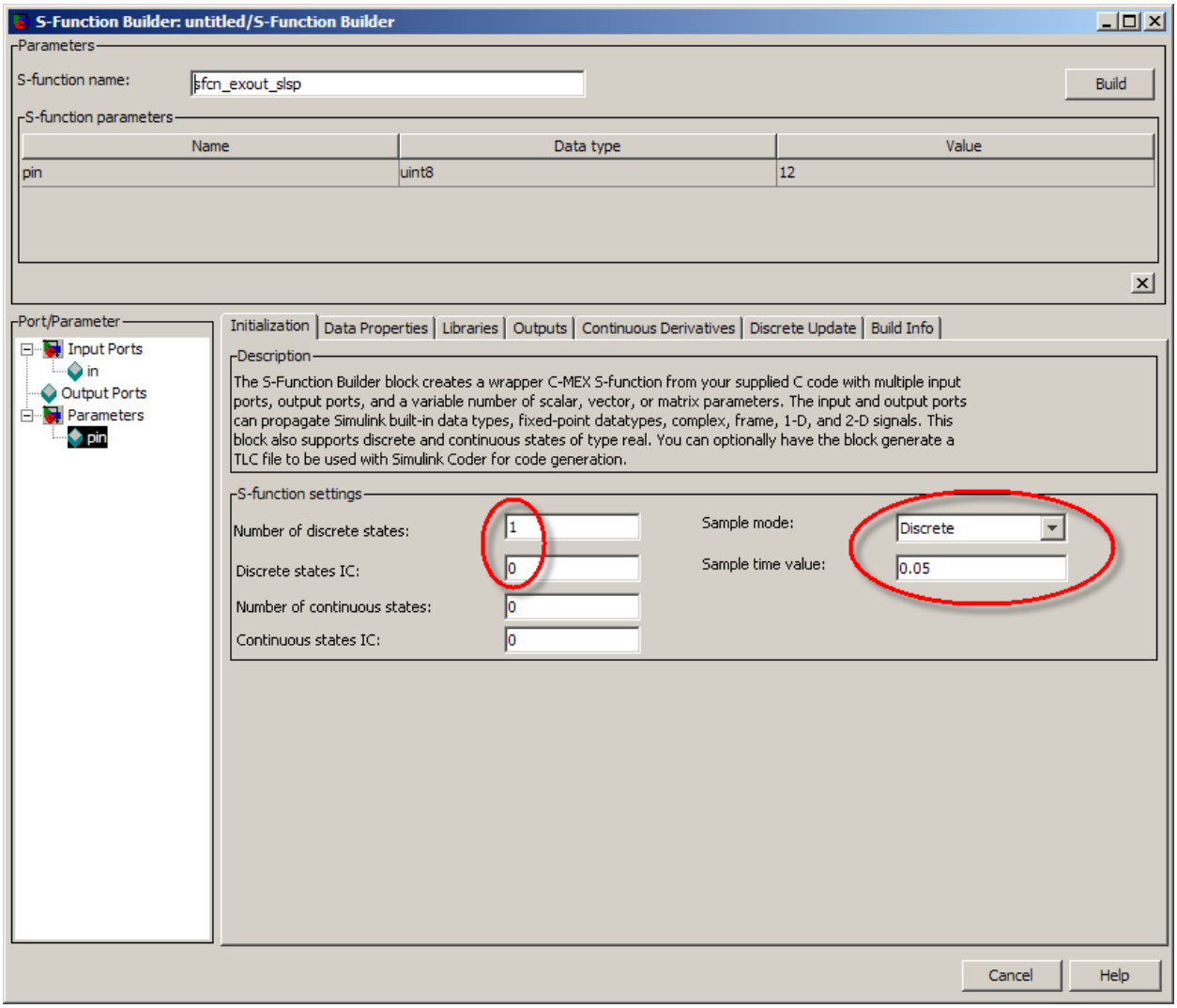

Figure 10: Initialization pane of the output driver block

The Initialization pane establishes the block sample time and its number of continuous and discrete states. Normally driver blocks execute in discrete time and have no continuous states. In this case we have chosen (see Figure 10) to set the sample time to 50ms but one could very well select the sample time to be inherited.

This implementation of a driver block requires that we set at least a single discrete-time state, which must be initialized to 0. One could add more states if needed, but the first element of the discrete state vector (that is xD[0]) must be initialized to 0 in order for the initialization part (which we'll see shortly) to work.

Output driver block: Discrete Update pane

The Discrete Update pane defines, in general, the evolution laws for the discrete state vector; however, as shown in Figure 11, here it is used to run some initialization code, which we type directly in the edit field.

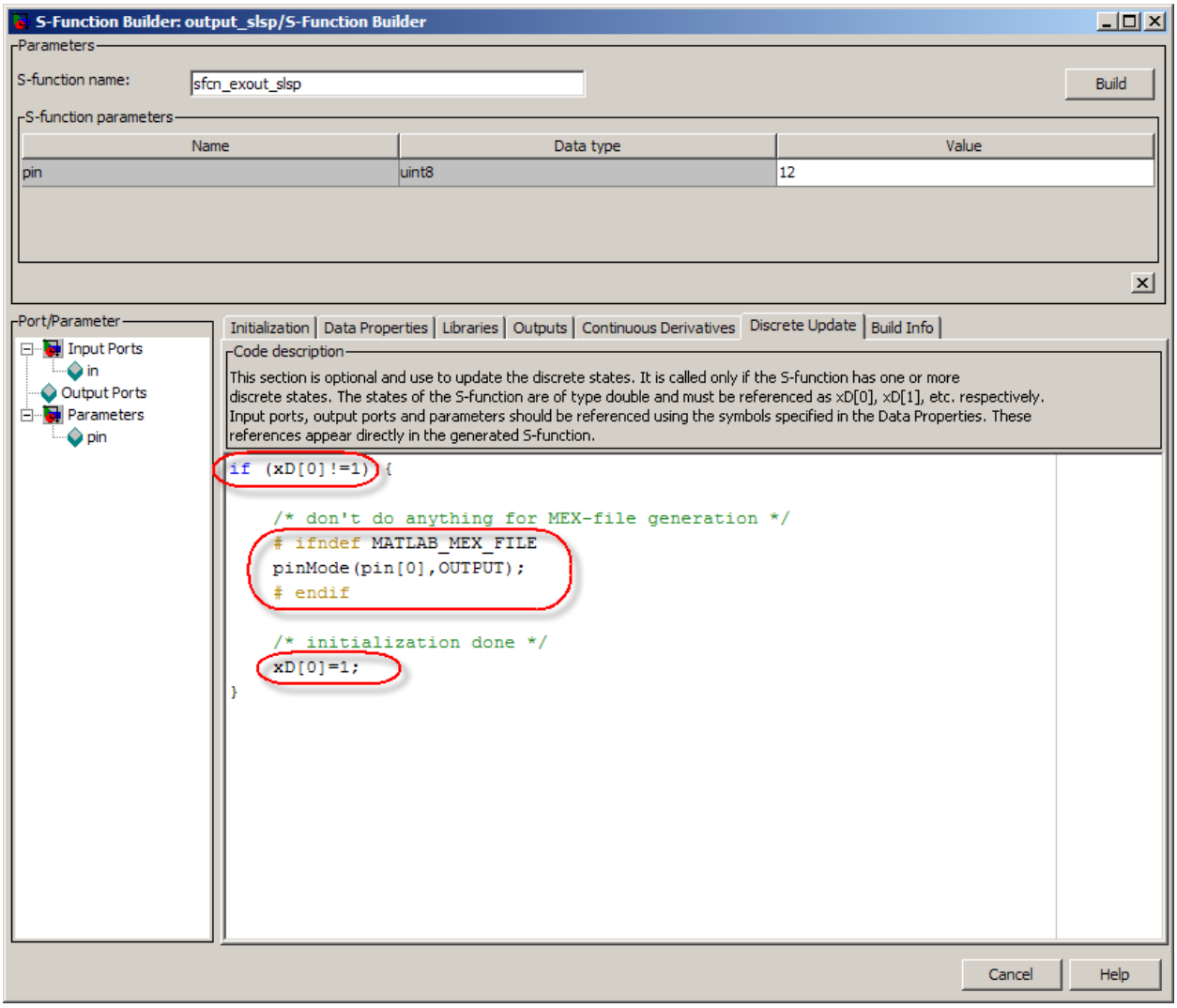

Figure 11: Discrete Update pane

The initial condition for the discrete state is 0 (this is set up by the initialization pane seen in the previous page), so the first time this Discrete Update function is called xD[0] is 0, and the code inside the brackets following the "if" condition is executed. The last line inside the brackets sets xD[0] to 1, which prevents anything inside the brackets from being executed ever again.

Let's now have a look at the 3 central lines inside the brackets:

```
# ifndef MATLAB_MEX_FILE
pinMode(pin[0],OUTPUT);
# endif
```

When a MEX file is generated from the S-Function Block (in order for the whole model to be simulated in Simulink), the identifier "`MATLAB_MEX_FILE`" is defined at compilation time.

The conditional compilation instruction `# ifndef MATLAB_MEX_FILE` prevents all the code that follows (until `# endif`) from being included in the compilation when the `MATLAB_MEX_FILE` identifier is defined.

As a result, when generating the executable *for the simulation*, the central line "`pinMode(pin[0],OUTPUT);`" will not be included in the compilation, and the resulting code will look like this:

```
if (xD[0]!=1) {

    xD[0]=1;
}
```

This code will simply set xD[0] to 1 the first time it is executed and then do nothing else ever again.

On the other hand, when an executable that needs to *run on the target* hardware is generated, the identifier "`MATLAB_MEX_FILE`" will not be defined, and as a consequence the central line will be included in the compilation, and the resulting code will look like this:

```
if (xD[0]!=1) {

    pinMode(pin[0],OUTPUT);

    xD[0]=1;
}
```

This code will call the Arduino "pinMode" function which will set the mode of the pin specified by the parameter pin[0] (12 in this case) to "`OUTPUT`" (for more information about what this means see http://arduino.cc/en/Reference/pinMode).

When writing your own output block, it is a good idea to start with this block and replace the line "`pinMode(pin[0],OUTPUT);`" with any initialization code you might need. If no initialization code is needed, then this line (but only this line) should be deleted. Note that any initialization code that is placed within the brackets but outside the conditional compilation directives `#ifndef` and `#endif` will execute *both* in the MEX file (at the beginning of the simulation) and on the target (when the target executable is launched on the target).

As it will be shown later, the code typed in the Discrete Update pane will end up inside the Update function of the wrapper file. The fact that it will be placed inside a function means, among other things, that any variable defined inside this code will not be accessible anywhere else (because its scope will be limited to the function). On the other hand, global variables (defined in the wrapper file, but outside of any function), will be accessible in the code typed in this pane.

Output driver block: Outputs pane

The Outputs update pane defines the actions that the block performs (in general on its outputs), when it is executed. As for the discrete update case, we can type the code directly in the edit field.

The first thing to notice (see Figure 12) is that the code in the brackets follows the condition xD[0]==1. Since xD[0] is 0 at the beginning and is then set to 1 by the first discrete update call, this means that the code in the brackets is executed only *after* the initialization code has already been executed.

The second thing to notice is that, similarly to what happens for the discrete update call, the Arduino specific instruction "`digitalWrite(pin[0],in[0]);`", (which writes the content of the variable in[0] to the pin specified by the parameter pin[0]) is wrapped up in the same conditional compilation statements seen before.

Again, this means that the MEX file generated for simulation purposes does not include any output code, and therefore does not do anything. Conversely, the executable that is generated for execution on the Arduino includes the digital write line. When this code is executed on the Arduino, assuming that in[0] is equal to 1 and pin[0] is equal to 12, a LED connected between the pin #12 and ground will light up.

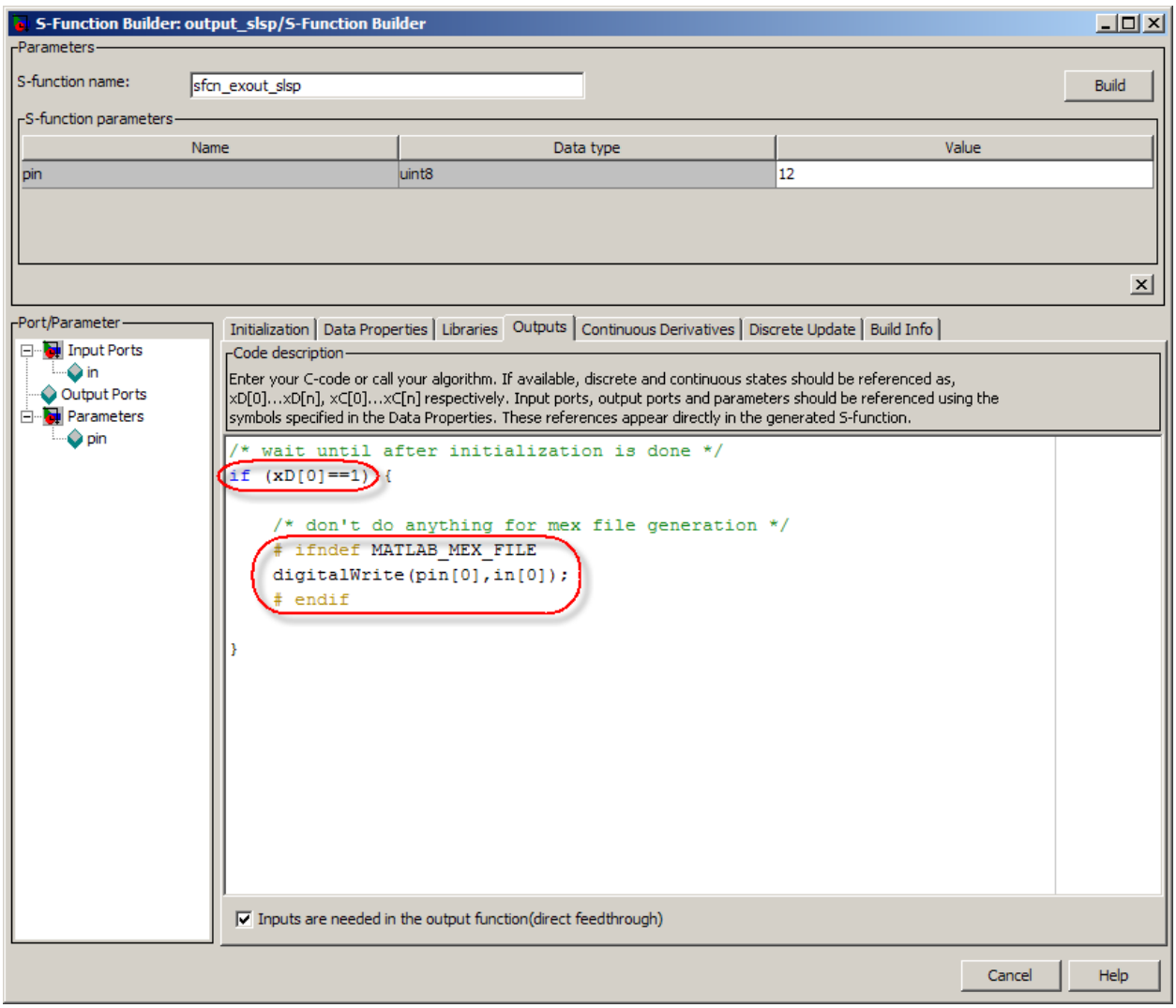

Figure 12: Outputs pane

When using this block as a starting point to create your own driver, the Arduino specific instruction "`digitalWrite(pin[0],in[0]);`" should be replaced with your own custom target specific code.

As it will be shown later, the code typed in the Outputs pane will end up inside the Outputs function of the wrapper file. The fact that it will be placed inside a function means, among other things, that any variable defined inside this code will not be accessible anywhere else (because its scope will be limited to the function). On the other hand, global variables (defined in the wrapper file, but outside of any function), will be accessible in the code typed in this pane.

Output driver block: Libraries pane

The last pane that needs to be taken in consideration is the Libraries pane. This pane allows you to specify external libraries, include files, and global variables that are needed by the custom code written in the other panes.

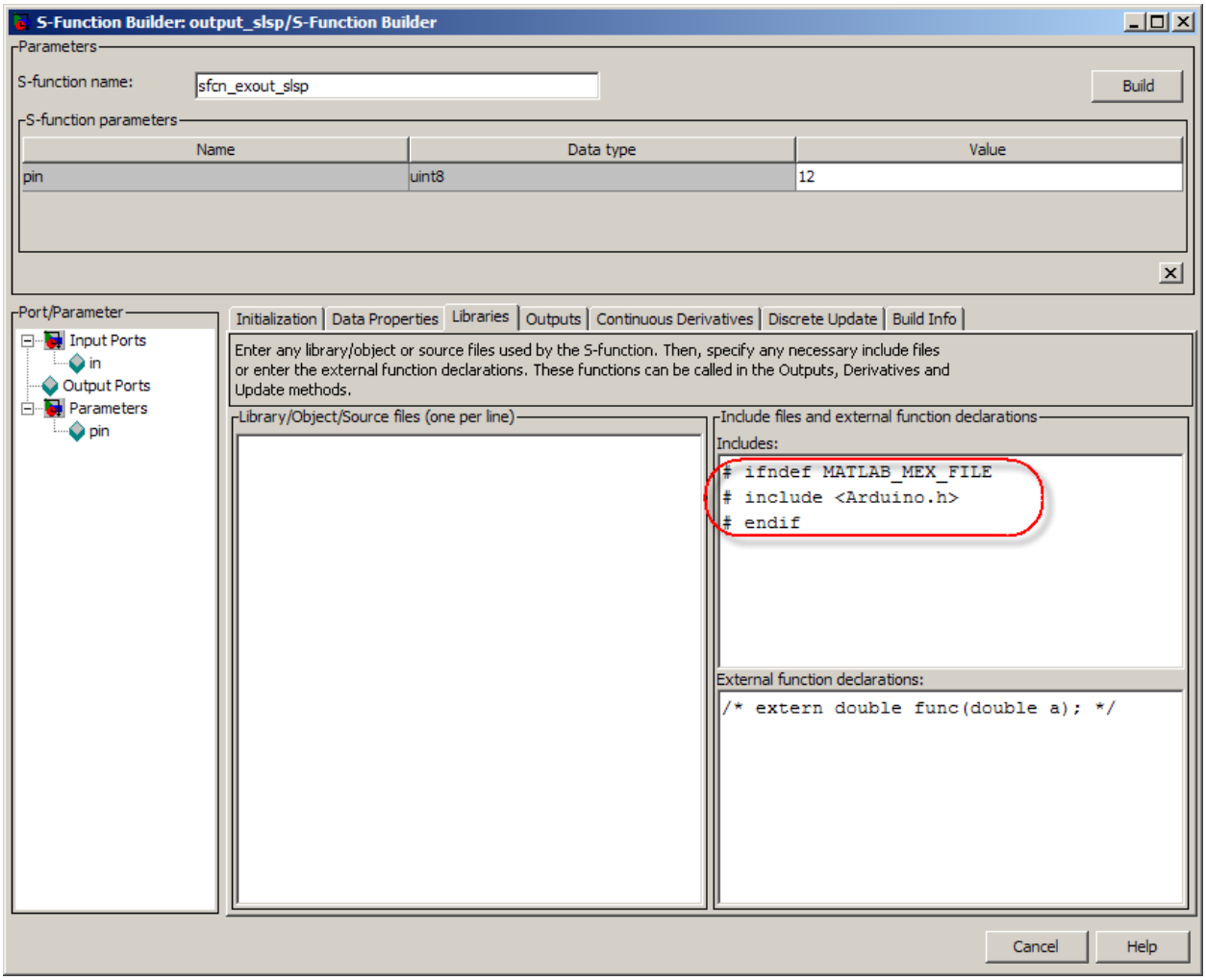

Figure 13: Libraries pane

For our purposes only the upper right “Includes” field needs to be considered (note that the Library/Object/Source field on the left hand side expects a lib file, e.g. .lib, .a, .so, .obj, .cpp, .c, files having .h or .hpp extension are ignored).

The three lines of code:

```
# ifndef MATLAB_MEX_FILE
# include <Arduino.h>
# endif
```

specify the conditional inclusion of the file Arduino.h, which contains, among other things, declaration for the functions pinMode and digitalWrite used in the Discrete Update and Outputs panes.

Differently from the code typed in the Update and Outputs pane, the code typed in the Libraries pane will not end up inside any function, but it will be placed directly at the beginning of the wrapper file, as it will be shown in the next section. This means that the Libraries pane is the perfect location in which global variables (which will be accessible from both the Update and Outputs functions) could be defined.

Note that, since the state vector xD is passed to both the Update and Outputs functions and it is local to the specific block, it is better to use xD as a way of sharing information among the two functions, instead of using global variables. Using global variables will result in code that is less clear, but, most importantly, global variables be shared within the whole model, so if two blocks which use the same global variable run in the same model, unpredictable results could follow. For these reasons, it’s better to use global variables only when unavoidable (e.g. variables belonging to a class which is not numeric).

At this point, we are ready to click on the “Build” button. If everything goes well four files will be generated: a wrapper file (sfcn_exout_slsp_wrapper.c), a TLC file (sfcn_exout_slsp.tlc) a MEX S-Function file (sfcn_exout_slsp.c) and a MEX-file (e.g. sfcn_exout_slsp.mexw32). The wrapper file, shown in the next section, contains the code that you typed in the dialog box panes, and it is referenced both in the MEX-file (for simulation), and in the TLC file (used to generate the executable which will run on the target).

Auto-generated sfcn_exout_slsp_wrapper.c file:

```
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
# ifndef MATLAB_MEX_FILE
# include <Arduino.h>
# endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
#define y_width
/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output functions
 *
 */
void sfcn_exout_slsp_Outputs_wrapper(const boolean_T *in,

                  const real_T  *xD,
                         const uint8_T  *pin, const int_T p_width0)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
/* wait until after initialization is done */
if (xD[0]==1) {

    /* don't do anything for MEX-file generation */
    # ifndef MATLAB_MEX_FILE
    digitalWrite(pin[0],in[0]);
    # endif

}
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
}

/*
  * Updates function
  *
  */
void sfcn_exout_slsp_Update_wrapper(const boolean_T *in,

                           real_T *xD,
                          const uint8_T  *pin, const int_T  p_width0)
{
  /* %%%-SFUNWIZ_wrapper_Update_Changes_BEGIN --- EDIT HERE TO _END */
if (xD[0]!=1) {

    /* don't do anything for MEX-file generation */
    # ifndef MATLAB_MEX_FILE
    pinMode(pin[0],OUTPUT);
    # endif

    /* initialization done */
    xD[0]=1;
}
/* %%%-SFUNWIZ_wrapper_Update_Changes_END --- EDIT HERE TO _BEGIN */
}
```

Include files from the "Includes:" field of the "Libraries" pane.

Outputs function, called at every sample time. It contains the code inserted in the "Outputs" pane.

Update function, called at every sample time to calculate the next value of the internal state vector xD. It contains the code inserted in the "Update" pane. In this example is used only for initialization purposes.

Working with external libraries

One of the challenges in working with external libraries (that is libraries added later that are not part of the standard distribution of the target) is that the compiler might not know where to find them.

One approach to solving this problem is placing the library files in the current MATLAB folder, and then refer to them as:

```
# include "myheader.h"
# include "mylibrary.c"
```

in the include field of the Libraries pane seen in the previous section. Note that an "undefined reference" error occurs when there are some library files that can't be linked (because the linker does not know where they are). In these cases you must make sure that these files are all in the current MATLAB folder and that they are all included if `MATLAB_MEX_FILE` is not defined.

Sometimes there are files that need to be included exactly once when the executable for the whole model is generated. One approach to handle these cases is to create (using the S-Function Builder) a block with no input, no states, and no outputs, which is used just to include the files that need to be included once per model. See for example the upper-right block in the model in Figure 14 below:

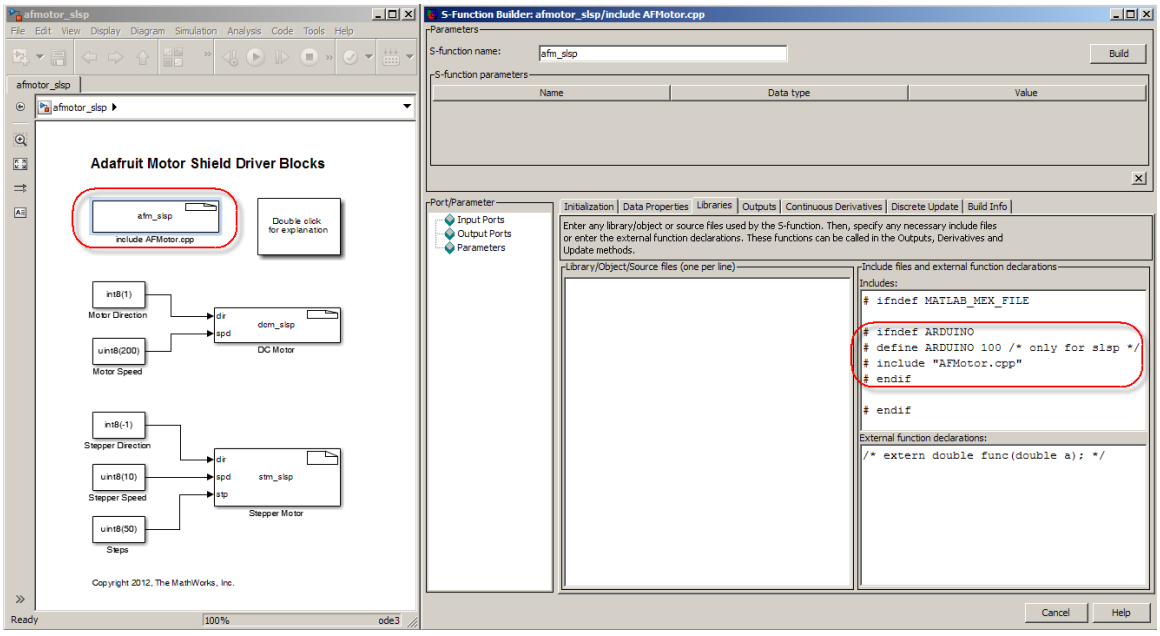

Figure 14: "Include" block for the AFMotor drivers

If the external library is in C++, then a few tweaks are necessary to make sure that the compiler and linker know how to handle interoperating C and C++ source files.

Specifically, you need to take the mydriver_wrapper.c file generated by the Builder, (where "mydriver" is the name of the S-Function chosen within the S-Function Builder, e.g. in the case above the name was "sfcn_exout_slsp"), and rename it to .cpp. Then open the file and add:

```
extern "C"
```

right before the definition of the two functions (before the "void") so it looks like this:

```
extern "C" void mydriver_Update_wrapper (const …
 ...
extern "C" void mydriver_Outputs_wrapper (const …
```

At this point the executable can be generated. It is important to remember to redo the above changes every time that, for any reason, the S-Function is rebuilt by the S-Function Builder.

Input driver blocks, and older versions of the Arduino IDE

The structure of an input driver is very similar (see the model "input_slsp" for reference). The differences are that an output port "out" is defined in the Data Properties pane instead of the input port "in", pin[0] is initialized as INPUT in the Discrete Update pane, and the instruction `out[0]=digitalRead(pin[0]);` is used in the Update pane (instead of the digitalWrite function).

When writing your own input driver block, it is a good idea to start with the block in the "input_slsp" model and replace the initialization and output part with the initialization and output code you might need.

Finally note that, if one is using versions of the Arduino IDE environment prior to 1.0 (with the Embedded Coder Support Package for Arduino, aka **Arduino Target**) the file that needs to be included (in the Libraries pane) is "WProgram.h" instead of "Arduino.h".

Troubleshooting: Undefined Reference

Undefined reference is a linker error. If you are working with external libraries and you get this kind of error that means that your code references objects defined elsewhere (in other files) and, at linking time, the linker cannot find where those objects are.

In this case you need to make sure that all the .c and .cpp files of the libraries you are using are in the current MATLAB folder and that they are all included in the "Includes" field of the "Libraries" pane of the S-Function Builder (alternatively, copying and pasting the whole file in the pane also works).

Note, include the .c and .cpp files directly, not the .h files!

Troubleshooting: Variable not defined in this scope

You have defined a variable in the Update pane code and are trying to use it in the Outputs pane code, or vice versa. This can't work because what you write in the Update pane ends up in the "update wrapper" function (see page 18) and what you write in the Outputs pane ends up in the "outputs wrapper" function (again, see page 18). These functions have separate scopes (they only see variables passed to them as inputs or defined inside themselves), so they can't share variables.

The easiest solution is to define the variable as global, (alternatively you could use instead additional elements of the state vector). Global variables can be defined after all the include directives inside the Includes field of the Libraries pane, (see page 16). As an example you might also look at the Library pane of the Encoder block in the "encoder_slsp.mdl". There the Includes field is used to define the encoder structure, the global encoder variables, as well as several auxiliary functions (including the interrupt service routines).

NOTE that one drawback of global variables is that they are shared among all the instances of the same block belonging to the same Simulink model, so, as a general rule, if your block has global variables it's better to make sure that you are not using that block more than once (in more than one place) in your model.