

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

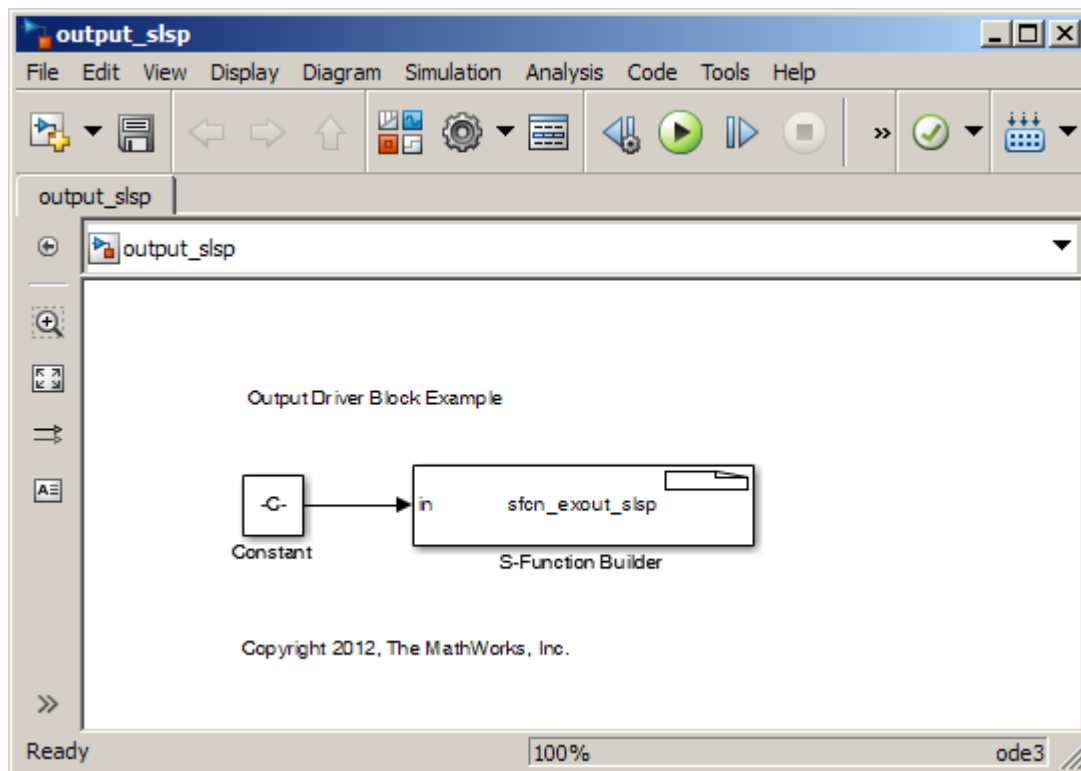
## Contents:

Writing a Simulink Device Driver block: a step by step guide.....	1
What is a device driver block? .....	2
A quick look at the selected target .....	3
Simulation vs. “on target” execution .....	4
The S-Function Builder Approach .....	6
Output driver block: Data Properties pane.....	7
Output driver block: Build Info pane .....	11
Output driver block: Initialization pane .....	12
Output driver block: Discrete Update pane.....	13
Output driver block: Outputs pane.....	15
Output driver block: Libraries pane .....	17
Auto-generated sfcn_exout_slsp_wrapper.c file:.....	19
Working with external libraries .....	20
Input driver blocks, and older versions of the Arduino IDE.....	22
Troubleshooting: Undefined Reference .....	22
Troubleshooting: Variable not defined in this scope.....	23
Masking S-Function Builder blocks .....	24
The MATLAB Function Approach .....	28
Output Driver: Data Manager and MATLAB code.....	29
Output Driver: The C code .....	32
Output Driver: Including the C-file in the build process .....	34
Output Driver: Masking the driver block .....	35

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?

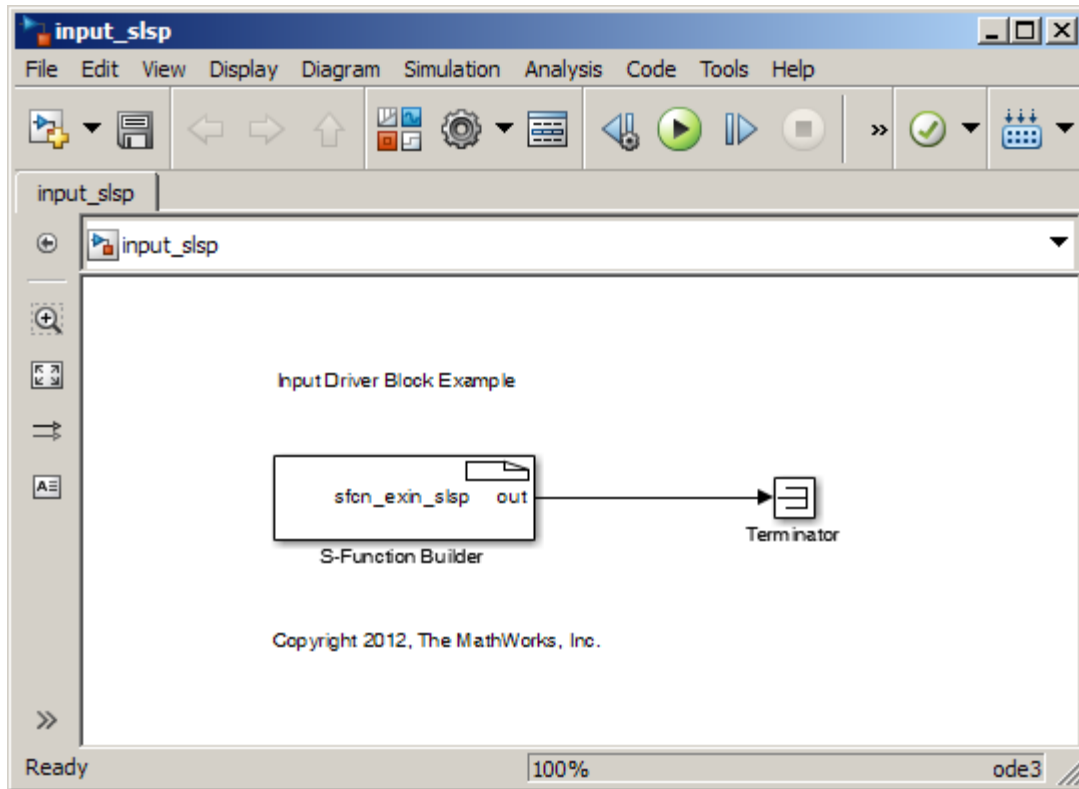
In general, an “Output Device Driver” block takes some signal as input and uses it to perform some kind of actuation on the target platform (e.g. analog or digital write) either directly with the hardware or through the real-time operating system.



**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**

## **A quick look at the selected target**

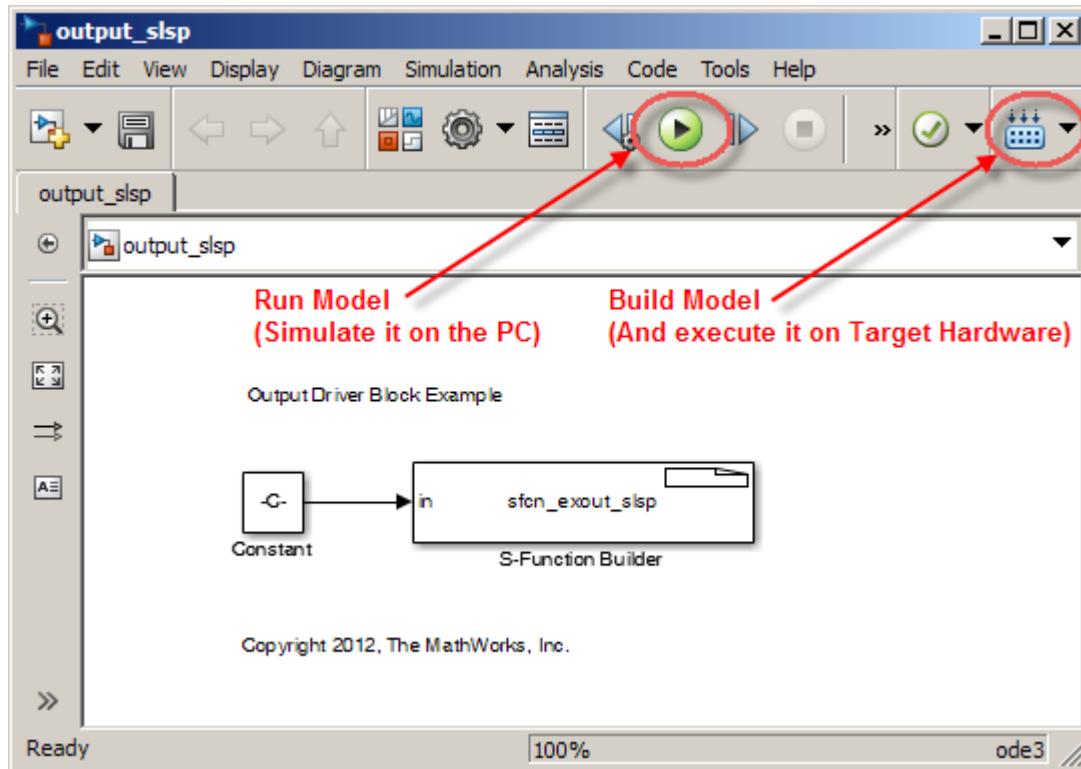
Before explaining the details of driver blocks, it is worthwhile noticing that when Simulink Coder is installed, a look at “Model Configuration Parameters -> Code Generation” shows (and allows changes to) the selected target.

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:



**Figure 3: Two different ways of executing the model**

**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*.

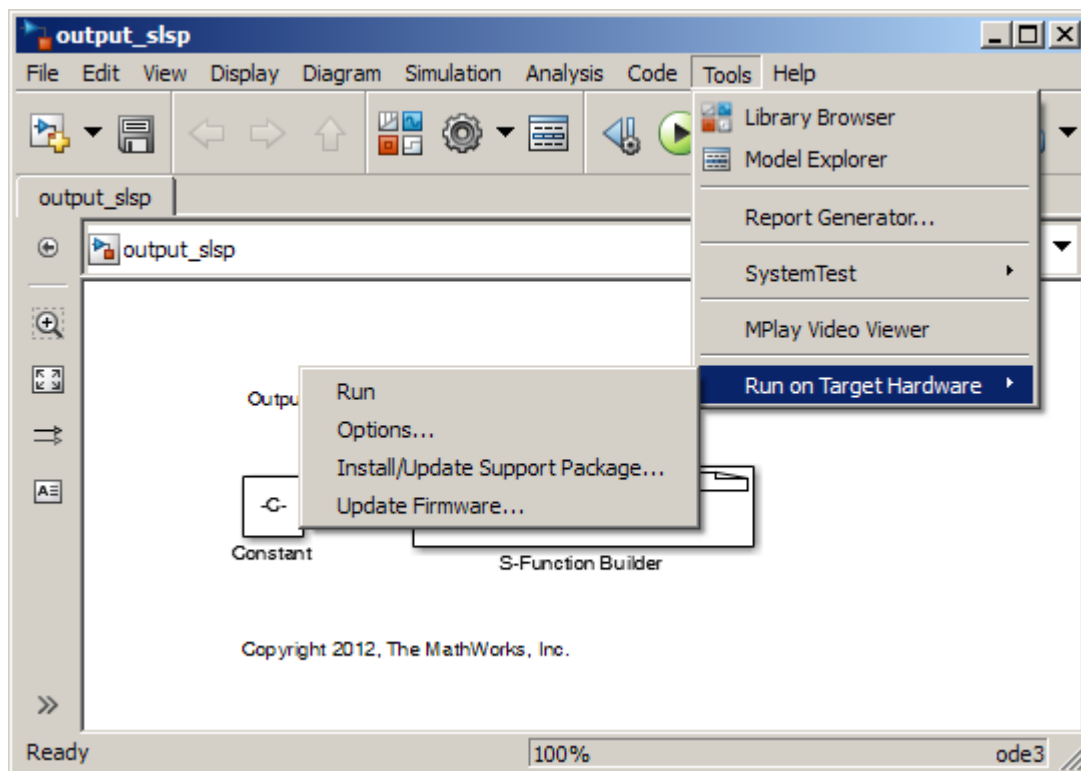
For example, when the MEX-file generated from the S-Function Builder block in Figure 2 is called, it does not do anything, and the output signal that goes to the terminator always remains at its default initial value of 0.

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).

**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).



**Figure 4: Run on Target Option**

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.

Unlike what happens when you simulate the model on the host, 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).

## The S-Function Builder Approach

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. Therefore, it might be a good idea to have a look at the help page for the S-Function Builder [dialog box](#).

**Note** that MATLAB **2013b** users will need to apply [this fix](#) for the S-Function builder before going any further (scroll down to the bottom of the page, and closely follow the instructions therein).

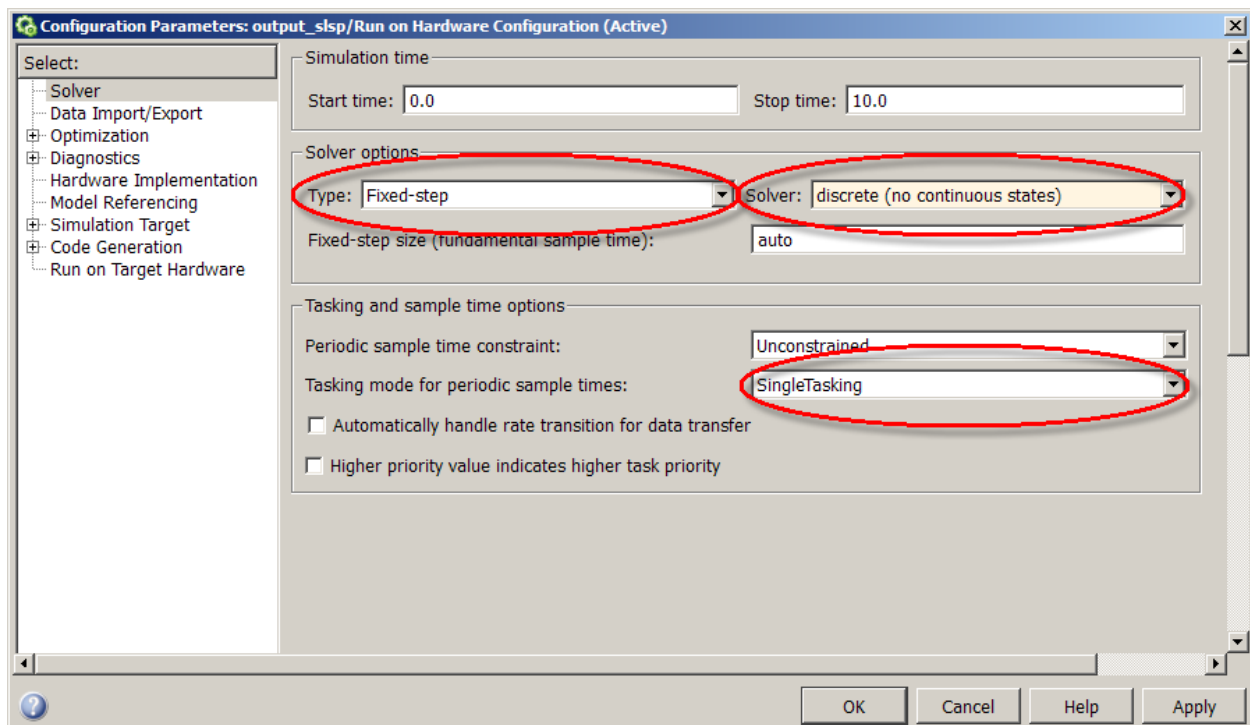
An alternative approach to develop device drivers, based on the MATLAB Function block instead of the S-Function Builder block, is also explained later in this guide (page 29), and the relative examples are given in the zip file EML2.zip. In general the MATLAB Function approach works better for developing more complex drivers and blocks that have to be redistributed and masked, while the S-Function builder approach might be better for developers who are more familiar

with C than the MATLAB Function block and need to quickly develop simple drivers that don't have a lot of parameters.

## 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).



**Figure 5: Solver Configuration Parameters**

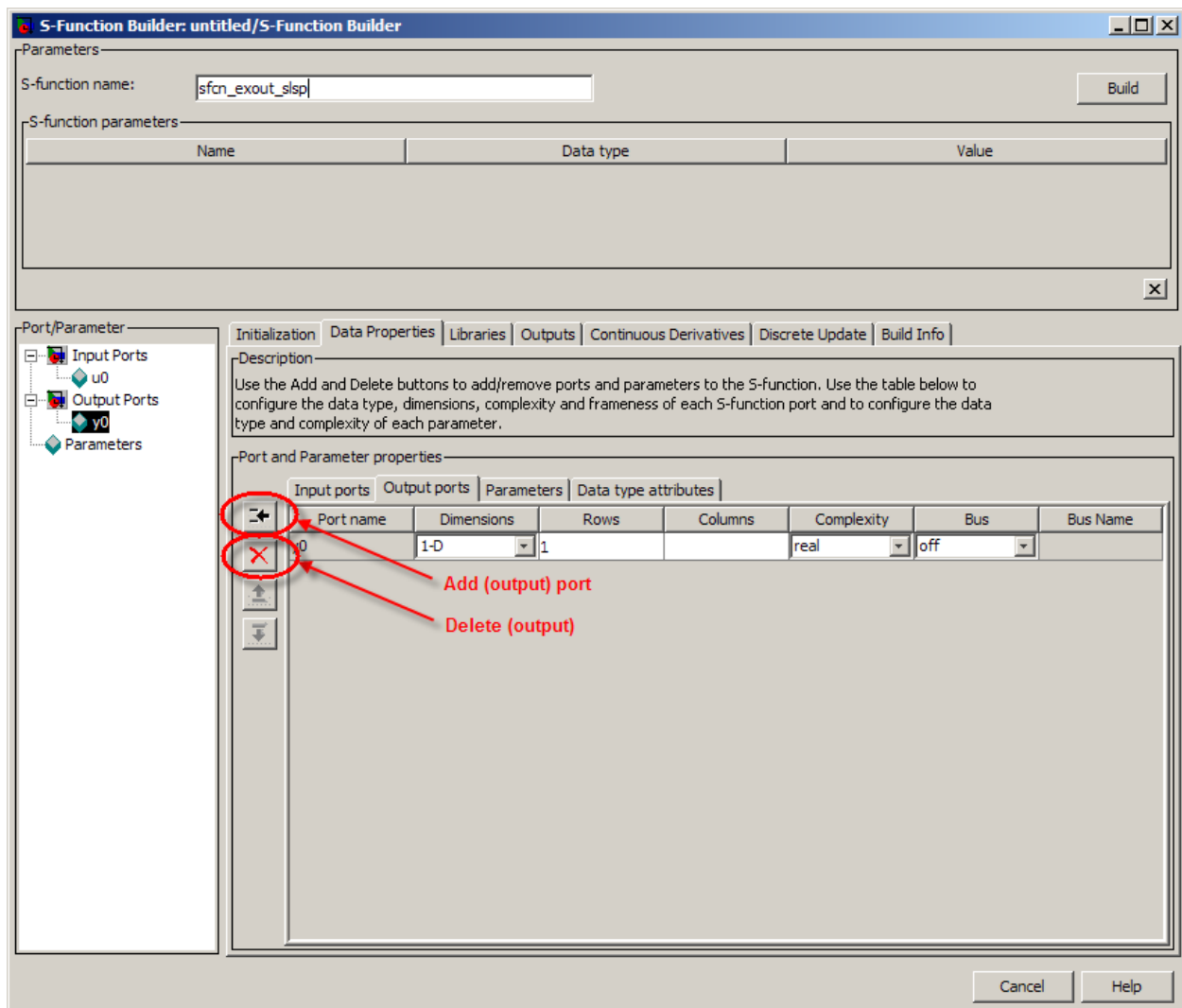
Since models that execute on embedded targets are frequently very simple and do not need to accurately integrate a differential equation, it might be also a good idea to open the Solver page of the Configuration Parameters (Simulation -> Configuration Parameters) and set the solver to “Discrete (no continuous 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 above.

The first step is to name 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 Properties 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.



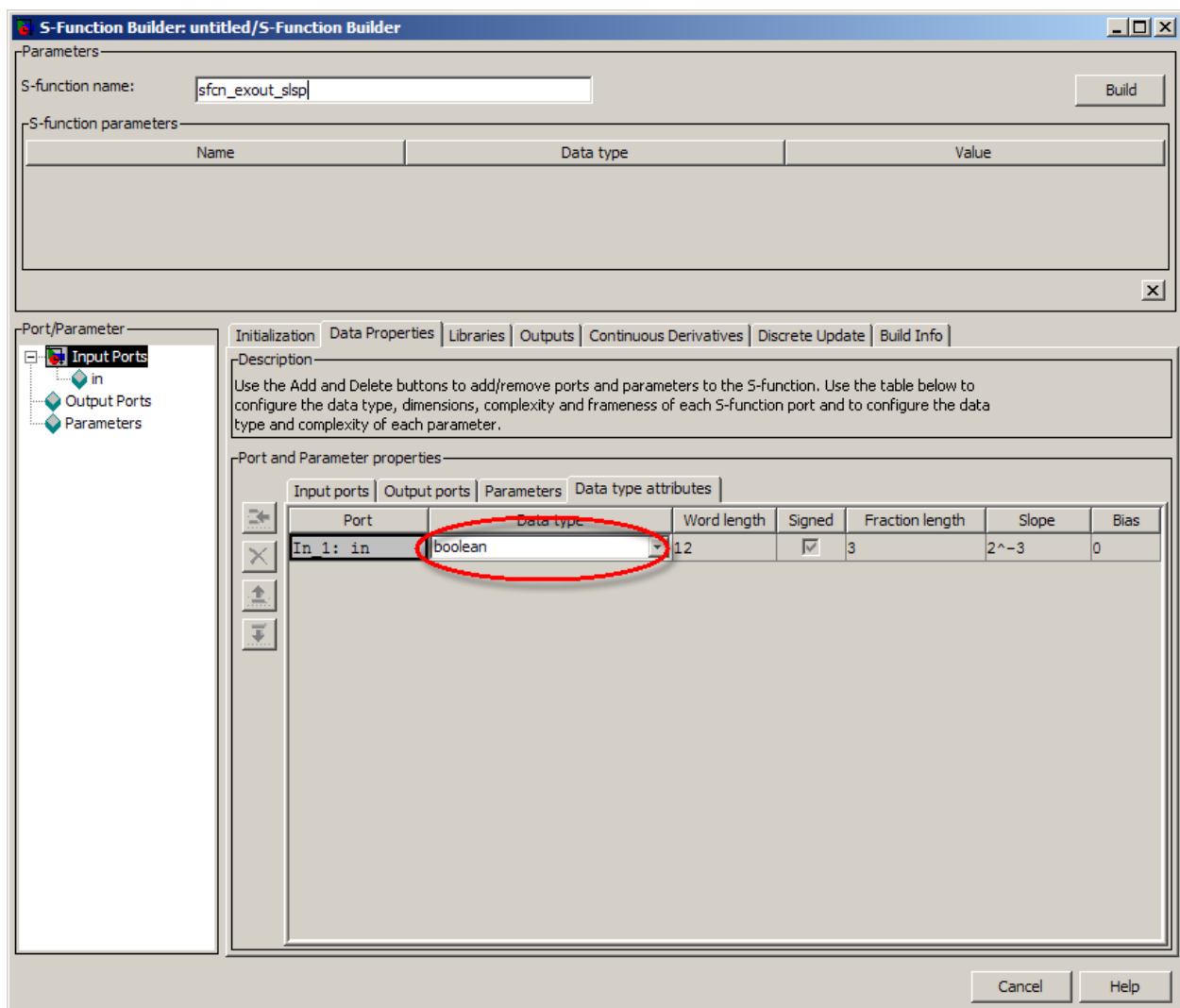
**Figure 6: Data Properties pane: Output ports**



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.

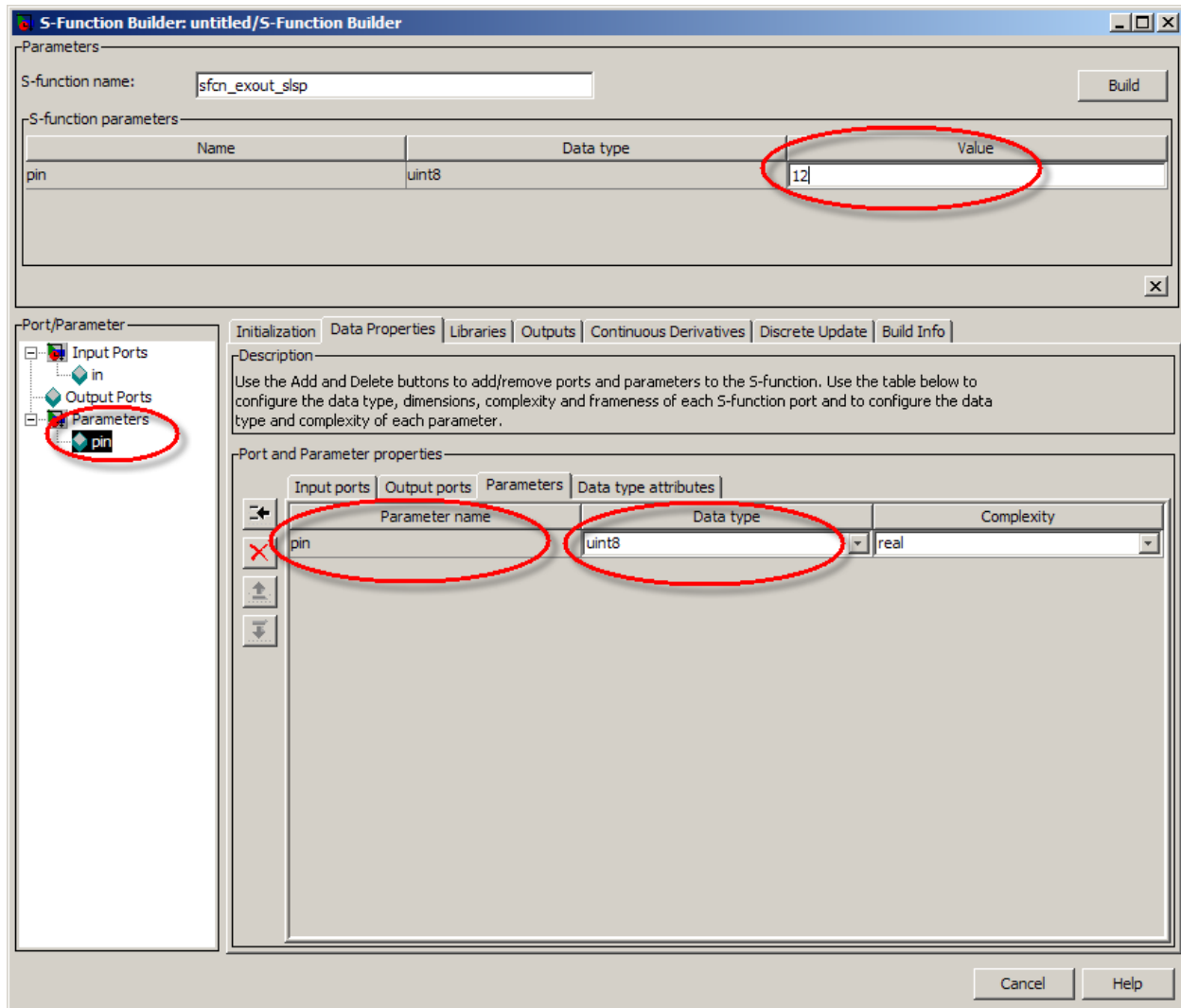
Similarly, by clicking on the “Input ports” subpane we can rename the input port to “in” instead of u0. The value coming from this input port will be later referred to as in[0] in the code.

Clicking on the “Data type attributes” subpane also allows us to change the data type of the input from double to boolean (a digital input only has two relevant values).



**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.



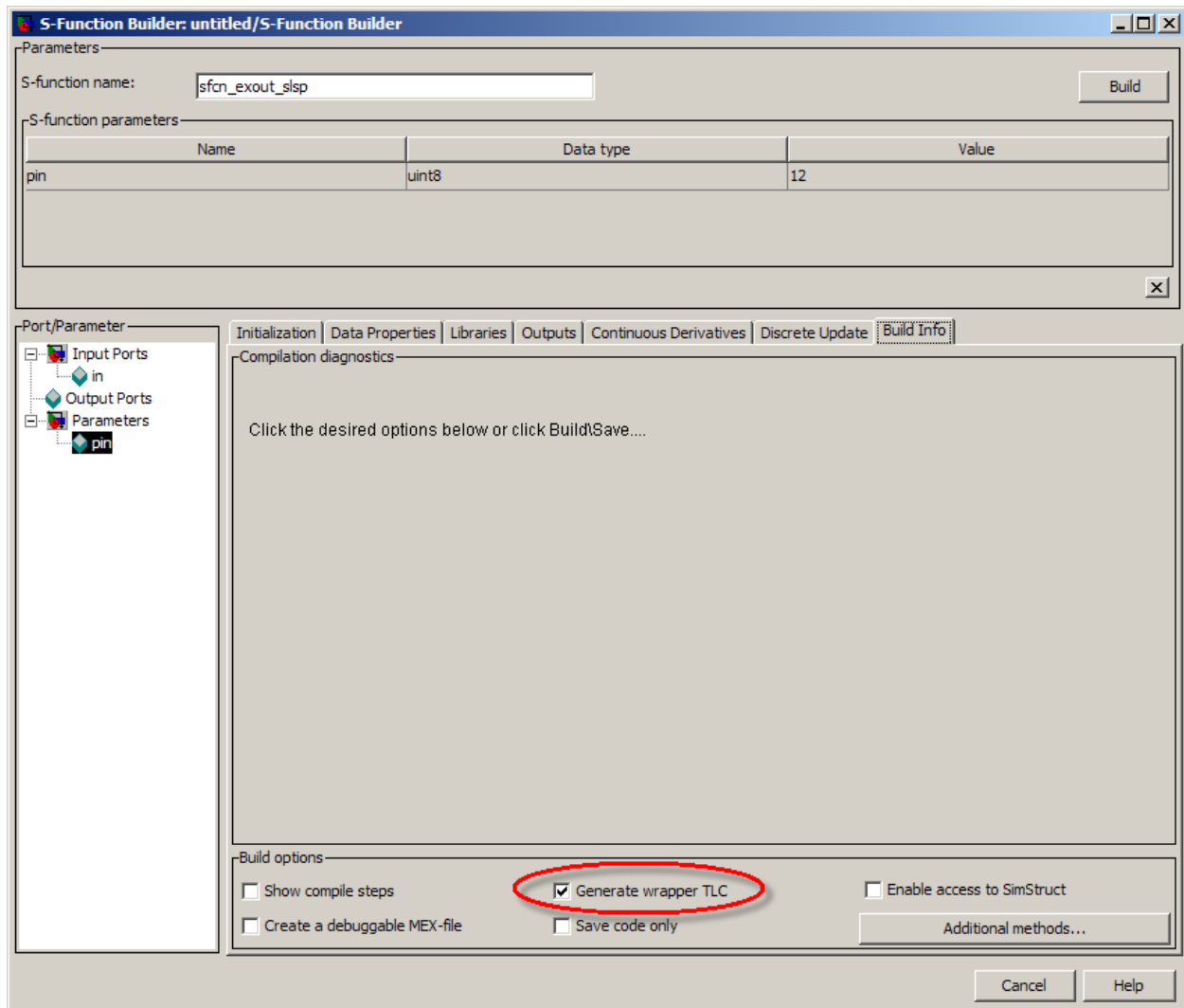
**Figure 8: Data Properties pane: Parameters**

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:

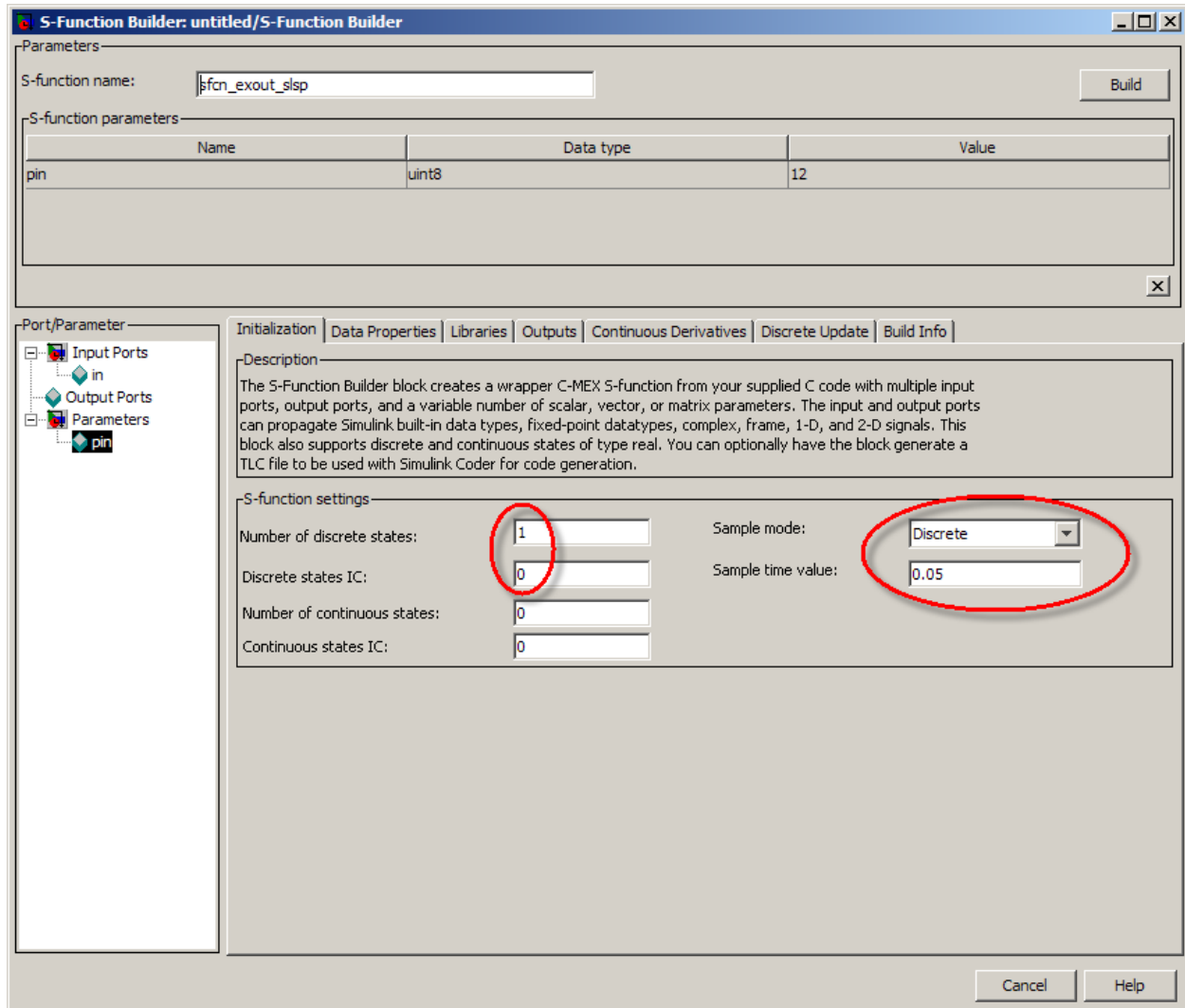


**Figure 9: Build Info pane of the output driver block**

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-functions.

## Output driver block: Initialization pane



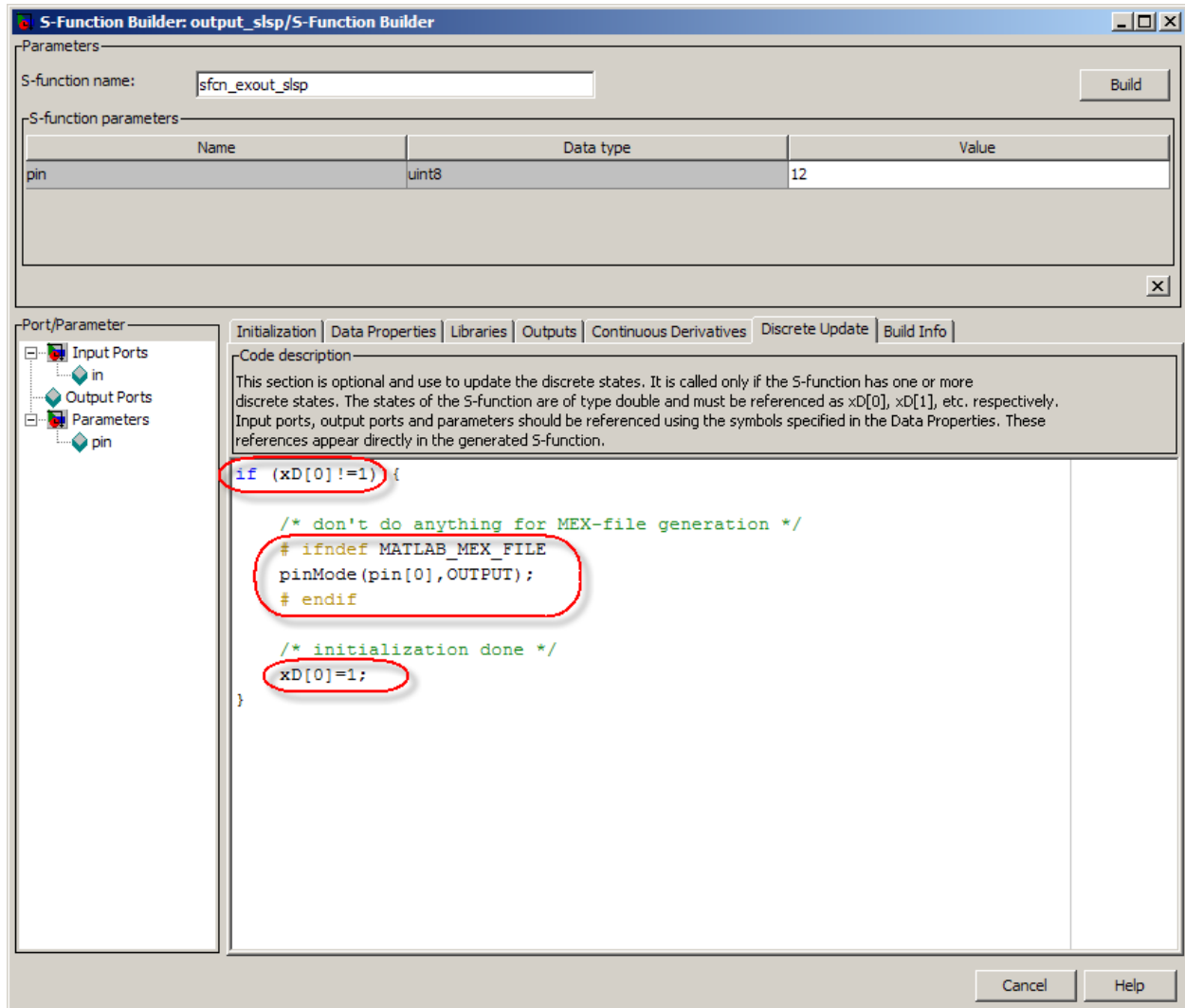
**Figure 10: Initialization pane of the output driver block**

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 to set the sample time to 50ms (see Figure 10) 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  $x_D[0]$ ) 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), therefore 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

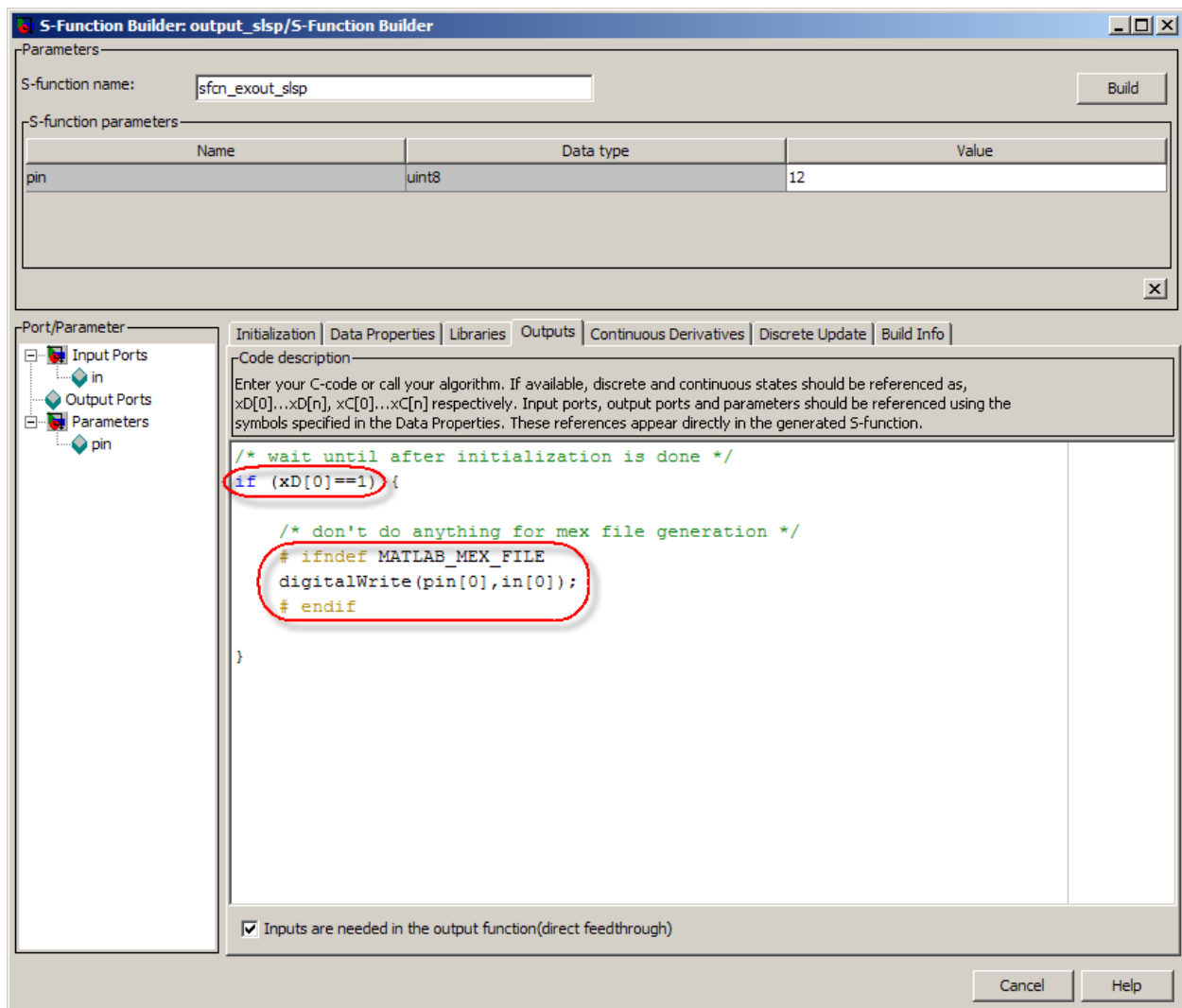
The Outputs update pane defines the actions that the block performs (in general on its outputs), when it is executed. As for the discrete update case, we can type the code directly in the edit field.

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, an 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.

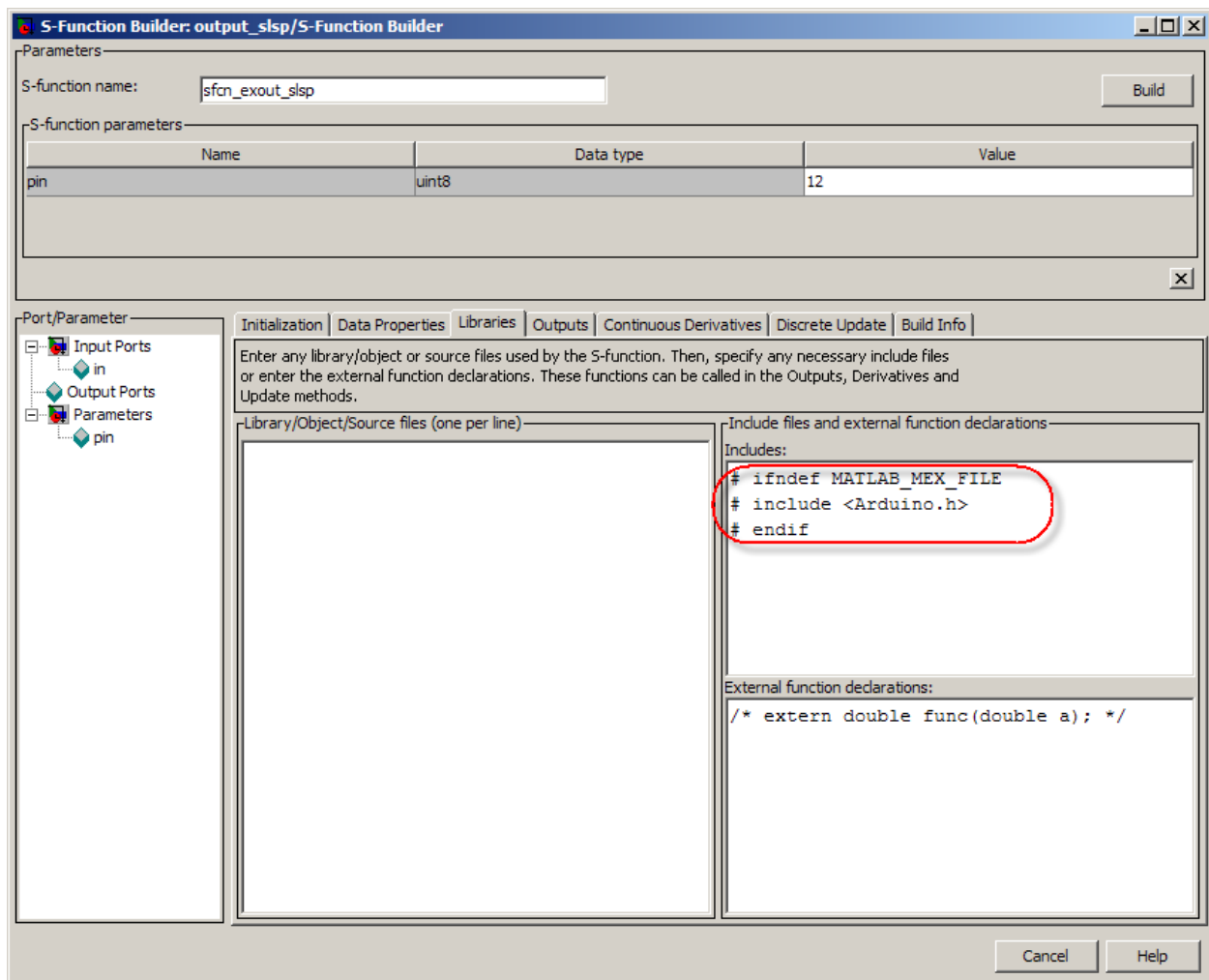
As it will be shown later, the code typed in the Outputs pane will end up inside the Outputs function of the wrapper file. The fact that it will be placed inside a function means, among other things, that any variable defined inside this code will not be accessible anywhere else (because its scope will 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 will be shown in the next section. This means that the Libraries pane is the perfect location to define global variables (which will be accessible from both the Update and Outputs functions).

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 and global variables are shared within the whole model. Therefore using two blocks with the same global variable in the same model may cause unpredictable results. For these reasons, it is 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 six files will be generated: a wrapper file (`sfcn_exout_slsp_wrapper.c`), a simulation-only S-function file (`sfcn_exout_slsp.c`), a simulation-only MEX-file (`sfcn_exout_slsp.mexw32`), two configuration files (`rtwmakecfg.m` and `SFB__sfcn_exout_slsp__SFB.mat`), and a TLC-file (`sfcn_exout_slsp.tlc`). The wrapper file, shown in the next section, contains the code from the dialog box panes which is also 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 */
# ifdef 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 */
        # ifdef 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 */
        # ifdef 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.

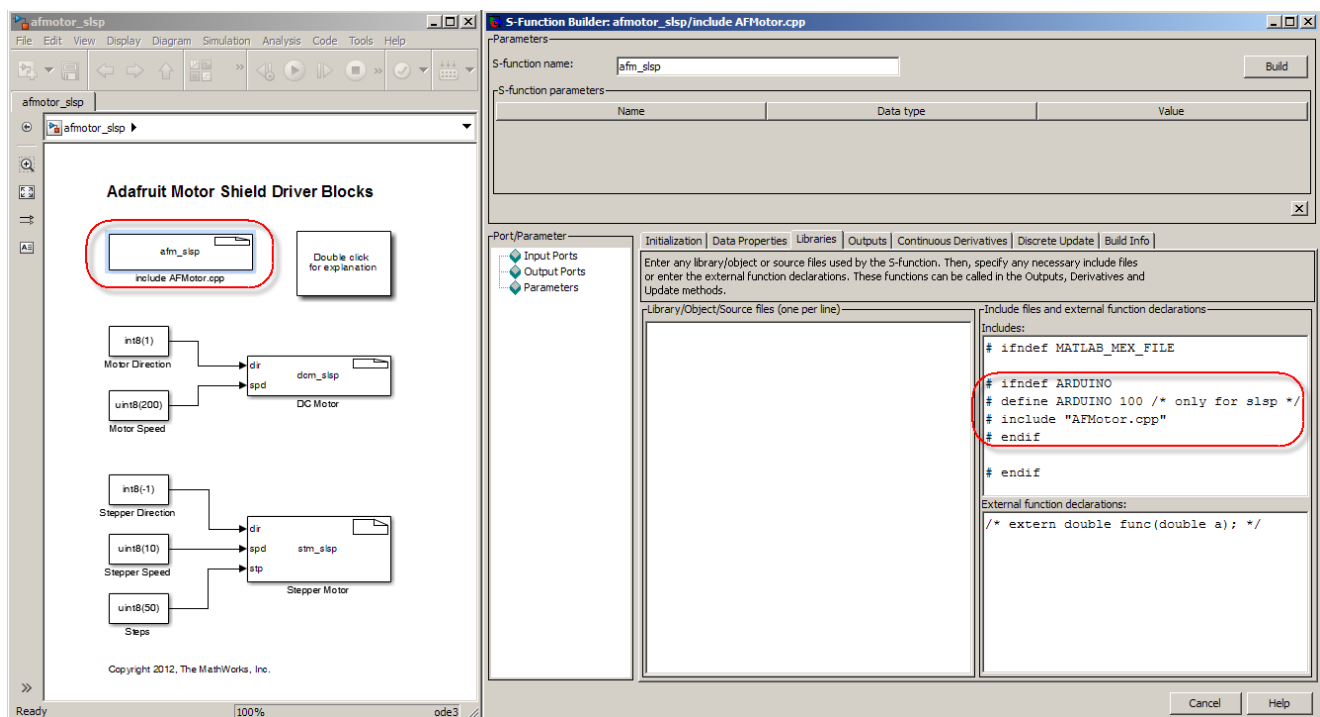


Figure 14: “Include” block for the AFMotor V1 drivers

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. For example, see the upper-left block in the model in Figure 14 above.

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, rename the `mydriver_wrapper.c` file generated by S-Function 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`”) to `mydriver_wrapper.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.

Note that the MATLAB function `renc2cpp` is also provided with this guide to automate such changes. So in this case, after building the “mydriver” S-function the user can invoke the `renc2cpp` function from the command line as follows:

```
>> renc2cpp('mydriver');
```

to rename the wrapper file and insert `extern "C"` before the Update and Outputs calls.

Another approach to handle source files or libraries that need to be included only once for the whole model is to set the model-wide parameters “CustomSource” and “CustomLibrary”. This can be done from the MATLAB command line as following:

```
>> set_param('model', 'CustomSource', 'mysrc1.cpp',  
'mysrc2.cpp');  
>> set_param('model', 'CustomLibrary', 'mylib.lib');
```

where “model” is the name of the Simulink model (without the file extension), “mysrc1.cpp” and “mysrc2.cpp” are two custom source files and “mylib.lib” a custom library file.

Note that the model-wide parameter “PostCodeGenCommand” can also be used to specify a command to execute after the code for the run on target is generated (and before the building of the executable that will run on the target is initiated):

```
>> set_param('model', 'PostCodeGenCommand', 'myfun.m');
```

For an example on how to use these parameters review the Adafruit Motor Shield V2 example in the [motorshields.zip](#) file, (specifically the files AFMotorV2.pdf, afmotor\_v2.mdl, AFMotorV2Setup.m, and setArduinoDefn.m).

## 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.

## Troubleshooting: Undefined Reference

Undefined references indicate a linker error. If you are working with external libraries and you receive 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).

If the files to be included need to be available for the whole model (e.g. because they are used by different driver blocks) then have a look at the previous section “Working with external libraries” on how to include model-wide files.

Alternatively, try to include the .c and .cpp files directly, instead of the .h files.

## **Troubleshooting: Variable not defined in this scope**

This generally occurs when a variable defined in the Update pane is used in the Outputs pane, or vice versa. This cannot 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), therefore they can’t share variables.

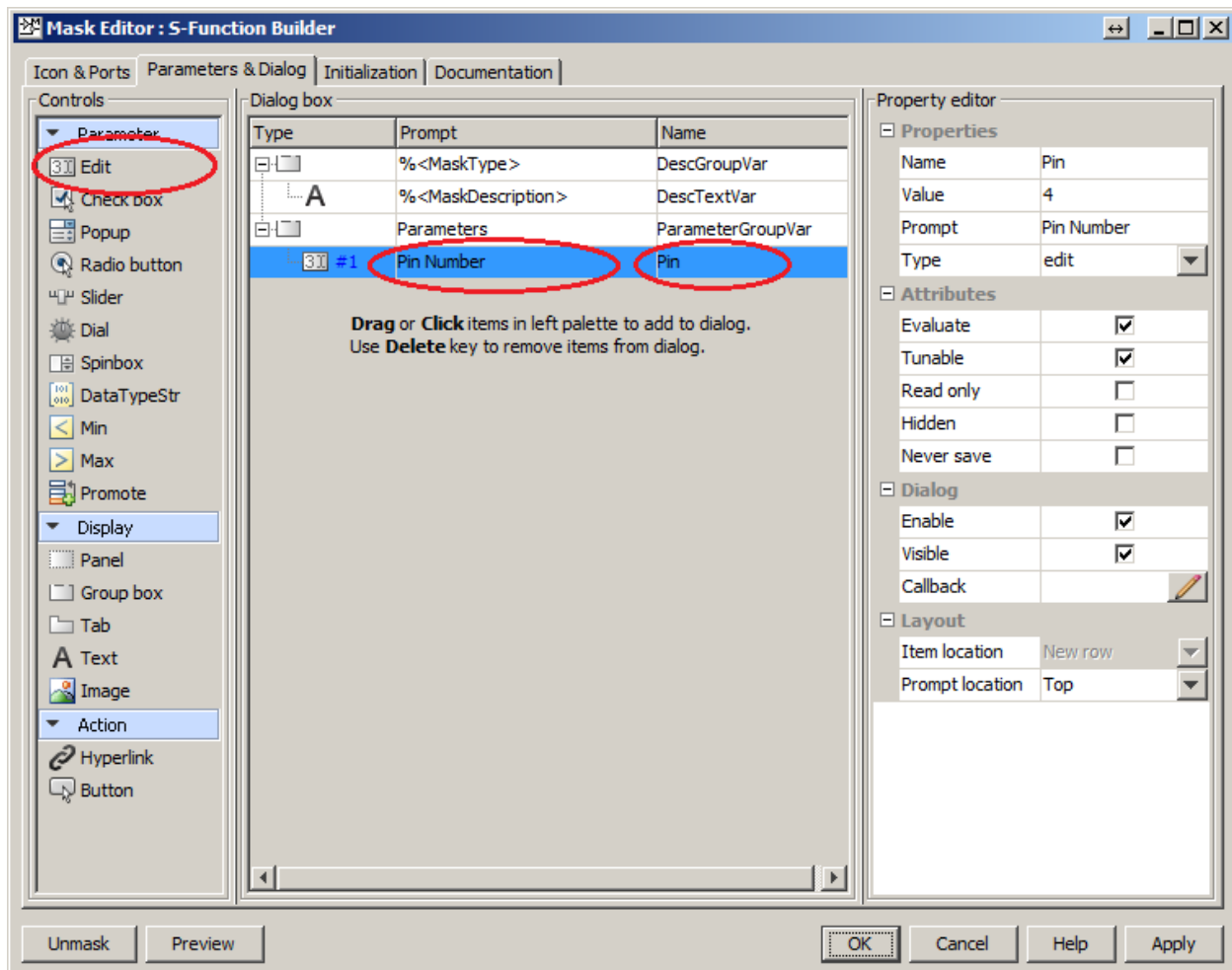
The easiest solution is to define the variable as global, (alternatively you could instead use additional elements of the state vector). Global variables can be defined after all the include directives inside the Includes field of the Libraries pane (see page 16). As an example you might also look at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

NOTE: 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 is better to make sure that you are not using that block more than once (in more than one place) in your model.

## Masking S-Function Builder blocks

It is possible to [mask](#) an S-Function Builder block as with any other Simulink block. To do so, select the block and either right click and select “Mask” then “Edit Mask” from the menu, or simply use the keyboard shortcut “Ctrl+M”.

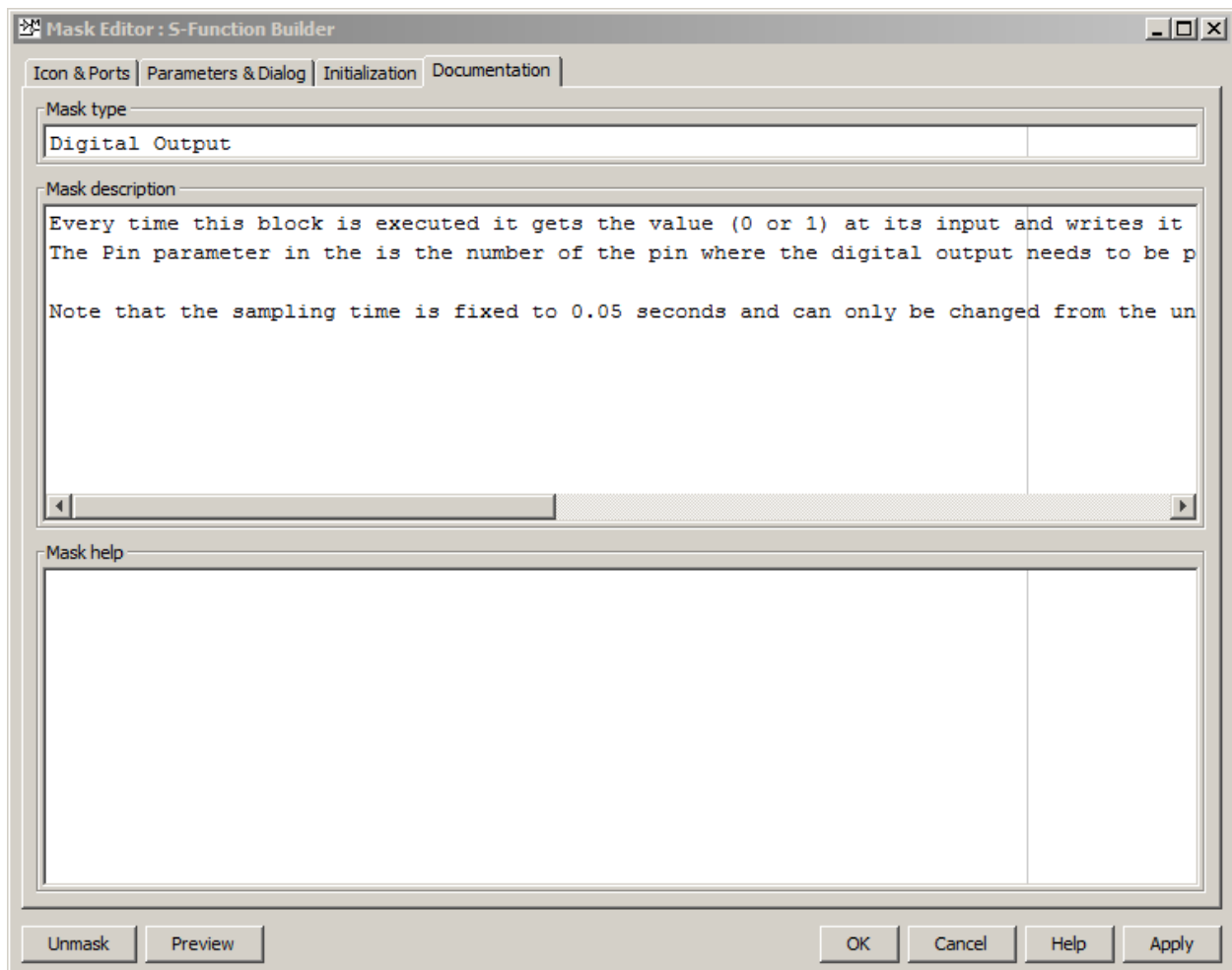
The “Parameters & Dialog” pane allows adding mask parameters. Specifically an “Edit” mask parameter can be added using the “Edit” control on the left hand side. Adding a “Pin” parameter to the digital output driver block results in the following view (Figure 15), where “Pin Number” is the prompt for the user, and “Pin” is the variable that carries the user-selected value (in this case a pin number, like 4). Such variable is visible (and usable) from the blocks below the mask (in this case from the S-function implementing the digital output).



**Figure 15: Masking the digital output S-Function Builder block.**



The “Documentation” pane allows you to add a proper description of the block to the mask (Figure 16):

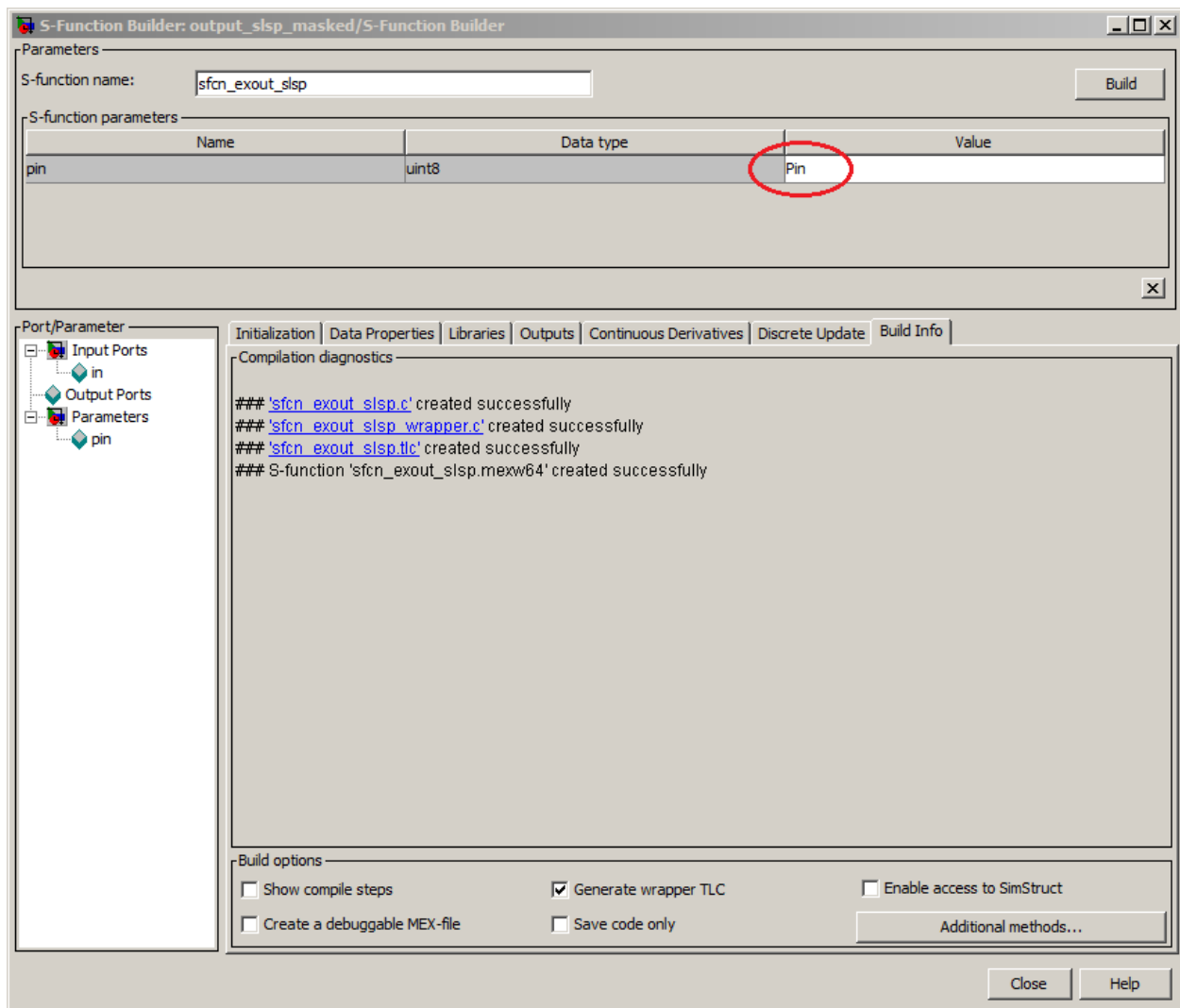


**Figure 16: Documentation pane for the digital output block mask.**

It is always good practice to write a few short sentences describing the block, what it does (i.e. its inputs, outputs, and states), as well as a description of the mask parameters and their significance.

It is important to remember that the “Pin” variable, which is passed from the mask to the block underneath, must now be used in the underlying S-Function Builder block instead of a numerical value (Figure 17).

This allows the user to change the pin number without having to rebuild the S-function.

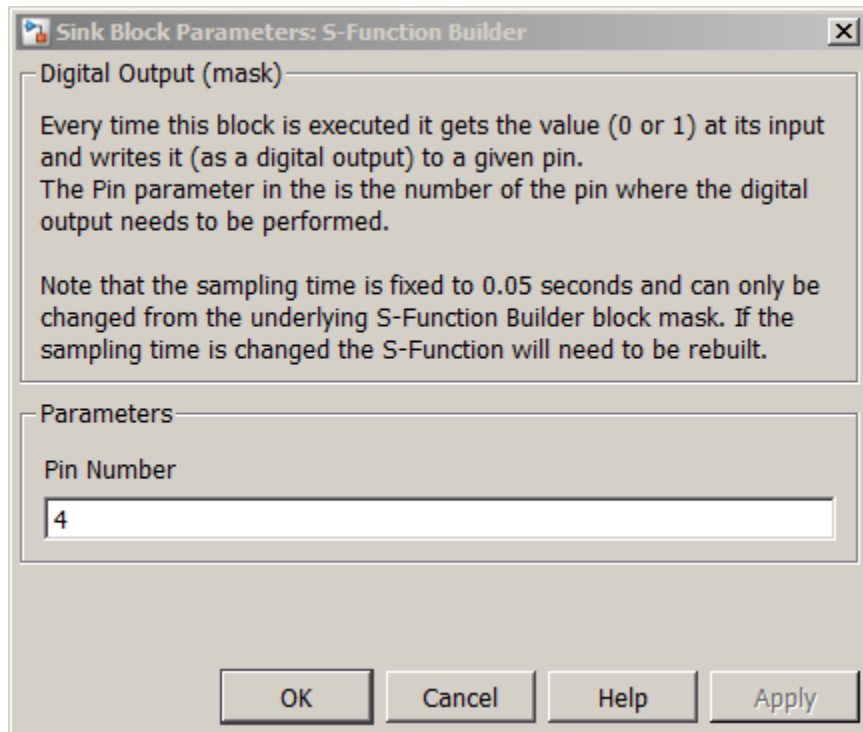


**Figure 17: “Pin” variable passed to the underlying S-Function.**

While normally double-clicking a masked block will bring up the mask, thus allowing users to change parameters, for an S-Function Builder block this can only be achieved by right clicking on the masked block and selecting “Mask” and then “Mask Parameters”.

Doing so for the masked digital output block brings up the mask in Figure 18.

Again, the main advantage of masking this driver block is that the user can change the pin parameter and deploy the whole model to the hardware without having to rebuild the S-function.



**Figure 18: Resulting mask for the digital output block**

In conclusion, it can be said that masking is especially relevant for driver blocks having many parameters that the user might want to change often. This is particularly true in cases where rebuilding the S-function involves several tedious steps such as compilation of external C++ libraries.

One important thing to remember is that the S-function settings in the “initialization” subpane of the S-Function Builder, like initial conditions and sampling time (Figure 10), are initialized as constants in the generated code and must be given as numerical values.

In other words, the sampling time of an S-Function Builder block **cannot be passed** as a variable from a mask.

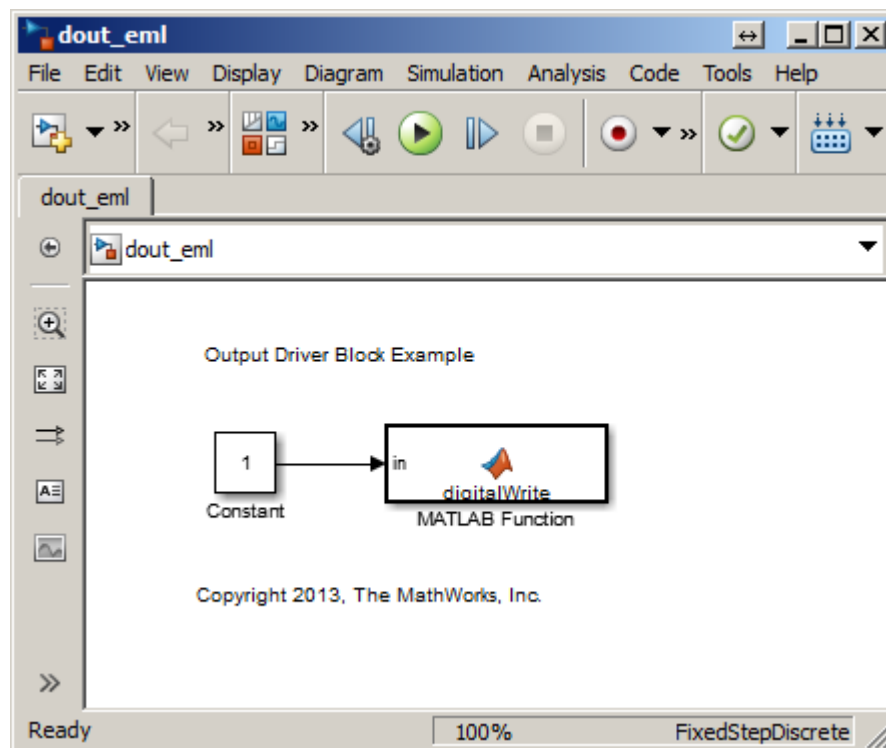
Therefore even for masked blocks, users wishing to change the sampling time must access the S-Function Builder block mask, change the numerical value, and rebuild the S-function.

This is an important limitation of the approach to creating device driver blocks presented so far.

## The MATLAB Function Approach

This approach to creating device driver blocks relies on two separate components that need to be shipped together.

The first is a Simulink model containing a [MATLAB Function](#) block (previously referred to as “Embedded MATLAB Function”, see Figure 19), which defines the dimensions and the parameters that can be changed by the user as well as handling the Simulink inputs and outputs.



**Figure 19: Output Driver using a MATLAB Function block**

The second component is a separate C (or C++) file which handles the lower level interactions with the target, contains an include section for the needed external libraries, an initialization function, and an output function. Both the initialization and the output functions are called from the MATLAB code in the MATLAB Function block.

In general, the MATLAB Function approach works better for developing more complex driver blocks that have to be masked and redistributed to a large number of users.

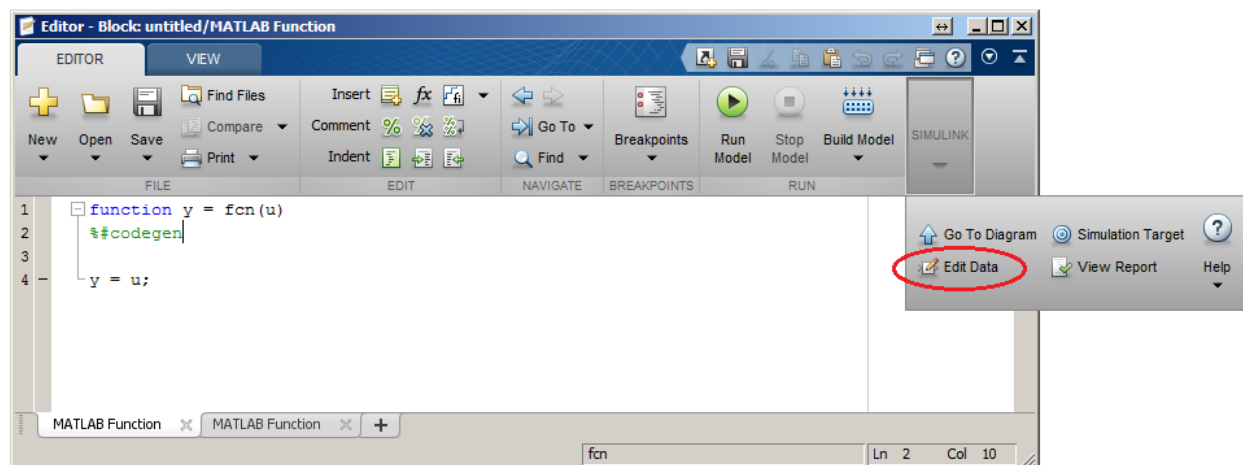
This is in part because, contrary to what happens for the S-Function Builder, double clicking on a masked MATLAB Function block brings up the mask directly (so there is no need to right click and choose “Mask” -> “Mask Parameters...” from the menu to access the block parameters).

Even more importantly, the fact that the block sampling time can also be passed as a parameter eliminates any need for a user to look or modify things under the mask, and to ever remember to directly rebuild (or actually even build in the first place) an S-function.

On the other hand, the need to use and track two separate components (the model and the C-file) as well as a few additional complexities involved in handling the MATLAB block and interfacing the MATLAB code to both Simulink and C, are some of the disadvantages of the MATLAB Function approach. Therefore the S-Function Builder approach might be better for developers who are more familiar with C than Embedded MATLAB and need to quickly develop simple drivers that don't have a lot of parameters.

## Output driver block: Data Manager and MATLAB code

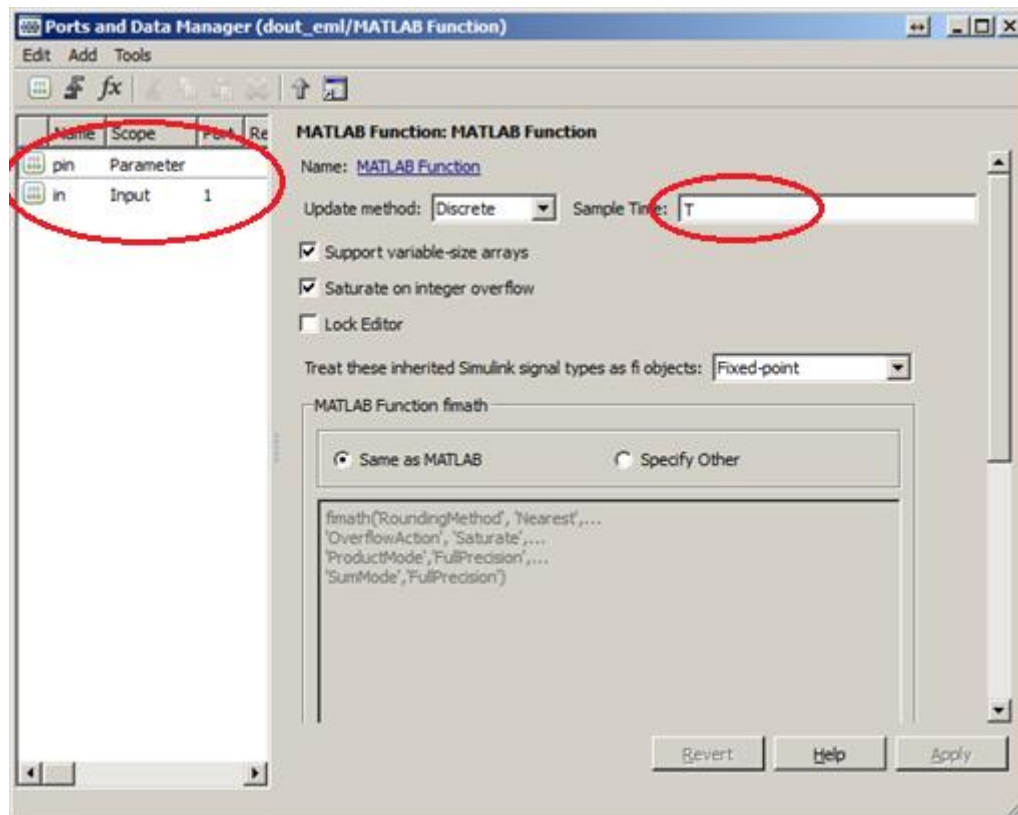
Similarly to the workflow described on page 6, we can create a new Simulink model, add a MATLAB Function block, then select the appropriate target and solver configuration for the model.



**Figure 20: Editing the MATLAB Function**

Double clicking on the block brings up the editor, which allows us to give an appropriate name to the function and start working on the MATLAB code and its inputs, outputs and parameters.

Specifically, clicking on the “Edit Data” button in the “Simulink” group (see Figure 20) brings up the data manager which allows you to define inputs, outputs, parameters, and the sample time.



**Figure 21: MATLAB Function input, outputs, and parameters**

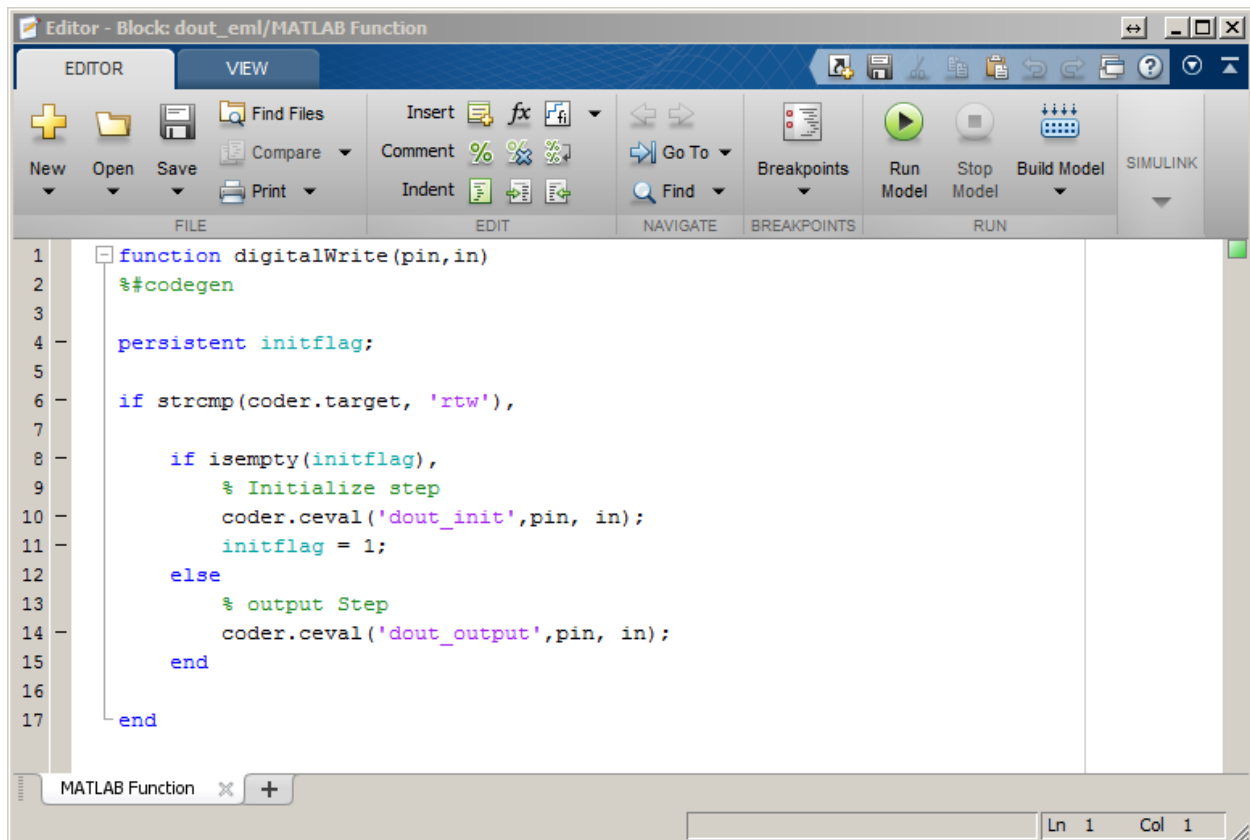
Similarly to S-Function Builder (pages 7-9), we can define an input variable (`in`), a pin parameter (`pin`), and the sampling time which is set to a variable `T` meant to be eventually passed by a mask (Figure 21). To test the driver before the mask is created, `T` can be defined as a normal variable in the MATLAB workspace.

Note that when defining both the input and parameter variables, we have the choice of specifying the data type (e.g. `unit8`, `double`, `single`, `boolean`).

Specifying the data type is in general a good idea, however, in this case we will leave the default choice “Inherit: same as Simulink” for both variables. This means that unless otherwise specified within Simulink the type is going to be double.

An example in which both parameters and inputs are instead defined as `uint8` types (and therefore handled as `char` variables in the underlying C code) is given in the “Arduino Motor Shield R3” driver which can be found in the EML2 folder of the [Motor Shield](#) zip file, available on File Exchange.

We are now free to actually use both the `pin` and `in` variables in the MATLAB code which we can just type in the editor (Figure 20).



The screenshot shows the MATLAB Function Editor window titled "Editor - Block: dout\_eml/MATLAB Function". The editor has tabs for "EDITOR" and "VIEW". The "EDITOR" tab is active, showing a MATLAB function named `digitalWrite` with parameters `pin` and `in`. The code is as follows:

```
1 function digitalWrite(pin,in)
2     %#codegen
3
4     persistent initflag;
5
6     if strcmp(coder.target, 'rtw'),
7
8         if isempty(initflag),
9             % Initialize step
10            coder.ceval('dout_init',pin, in);
11            initflag = 1;
12        else
13            % output Step
14            coder.ceval('dout_output',pin, in);
15        end
16    end
17 end
```

The editor includes a toolbar with icons for New, Open, Save, Find Files, Compare, Print, Insert, Comment, Indent, Go To, Find, Breakpoints, Run Model, Stop Model, Build Model, and SIMULINK. The status bar at the bottom shows "MATLAB Function" and "Ln 1 Col 1".

**Figure 22: MATLAB Function code**

The resulting MATLAB code for an output driver block (which works as an interface between Simulink and the actual initialization and output driver calls written in C) is shown in Figure 22.

The function `digitalWrite` takes in the parameter (`pin`) and the input (`in`) both defined in the data manager with an “Inherited” type. The persistent variable

`initflag` (which is empty at the beginning and is really a state variable meant to be used as an initialization flag) is then defined.

In order for the whole model to be simulated in Simulink, a MEX-file is generated from the MATLAB Function block. Within this MEX-file, the function “`coder.target`” will return the string “MEX”. Since “MEX” is different than “rtw” (see line 6 in Figure 22) the bulk of the function is skipped and nothing is done. This is what is normally expected from a driver block in simulation mode.

On the other hand, within an executable that *runs on the target* hardware, the function “`coder.target`” returns the string “rtw”.

Since `initflag` is empty the first time this function is executed on the target (see line 8) the C **initialization** function `dout_init(pin,in)` is called by the `coder.ceval` instruction (line 10) and then `initflag` is then set to 1 (line 11).

From that point on, every successive time the function is executed on the target, the C **output** function `dout_output(pin,in)` is called by the `coder.ceval` instruction (line 14).

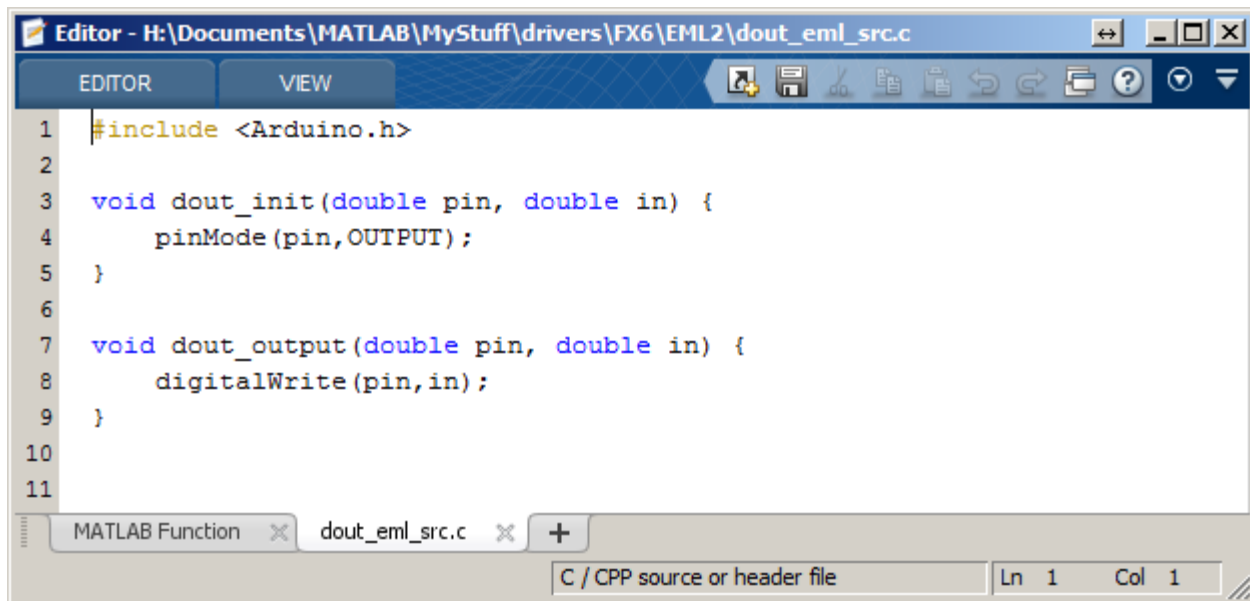
Note that this behavior is perfectly equivalent to the one described for the S-Function Builder block in pages 13-15 with the small difference that since an S-function must have two separate calls for output and discrete update, the conditional compilation instructions checking on the compilation target must be repeated in both calls.

## Output driver block: the C code

At this point, we can have a look at the C-file containing the actual initialization and output calls (Figure 23).

This code is also pretty straightforward as it consists of a header section where libraries can be included and global variables can be defined followed by an initialization function and an output function respectively.





```
1 #include <Arduino.h>
2
3 void dout_init(double pin, double in) {
4     pinMode(pin, OUTPUT);
5 }
6
7 void dout_output(double pin, double in) {
8     digitalWrite(pin, in);
9 }
10
11
```

**Figure 23: C driver code containing initialization and output calls**

In this specific example (digital output for the Arduino board), the initialization call `dout_init` uses `pinMode(pin, OUTPUT)` to initialize to OUTPUT a certain pin. The value `pin` is specified by the user in the mask and passed from the mask to the MATLAB function which in turn passes it down to this initialization call.

Similarly, in the output call `dout_output`, the Arduino specific instruction `digitalWrite(pin, in)` is used to write the content of the variable `in`. This value, which is either 0 or 1 is passed from Simulink to the MATLAB function which in turn passes it to this output call to the pin specified by the parameter `pin`.

Note that the Arduino functions `pinMode` and `digitalWrite` accept variables of type `uint8_t` for both their inputs. Therefore we are relying on an automatic casting from `double` to `uint8_t` because both `pin` and `in` are defined and passed down as `double` (unless otherwise specified in Simulink).

Had we instead defined `pin` and `in` as variables of type `uint8` in the data manager (Figure 21), then we would have had to use `char` or `uint8_t` instead of `double` before the definitions of the `pin` and `in` variables in the C function headers on lines 3 and 7 of the C source file in Figure 23.

It is also important to remember that if we include any C++ library in the first lines of the source file in Figure 23, then the file needs to have a “cpp” extension and we

need to write extern “C” before the two function calls on lines 3 and 7 (similarly to what happens to the S-function generated by S-Function Builder described on page 20).

## Output driver block: including the C-file in the build process

In order for the executable file to be built and deployed to the target, the compiler has to be able to find the initialization and output calls referenced by the MATLAB `coder.ceval` instructions (in Figure 22).

As already mentioned in page 20, this can be achieved using the `CustomSource` model parameter. Specifically, executing the following instruction from the MATLAB command line:

```
>> set_param('dout_eml','CustomSource','dout_eml_src.c');
```

will set the `CustomSource` parameter to the string “`dout_eml_src.c`”, for the Simulink model “`dout_eml`”.

As a consequence, if the file `dout_eml_src.c` can be found at compilation time in the current folder, it will be included in the build process and the initialization and output driver calls defined in that file will be located, compiled, and linked.

Note that if a user drags and drops the driver block into a new Simulink model, she will need to again execute a `set_param` command to set the `CustomSource` parameter to the string “`dout_eml_src.c`”, for the new model.

The fact that users need to remember to do this is a specific shortcoming of the MATLAB Function approach (in comparison to the S-Function Builder approach).

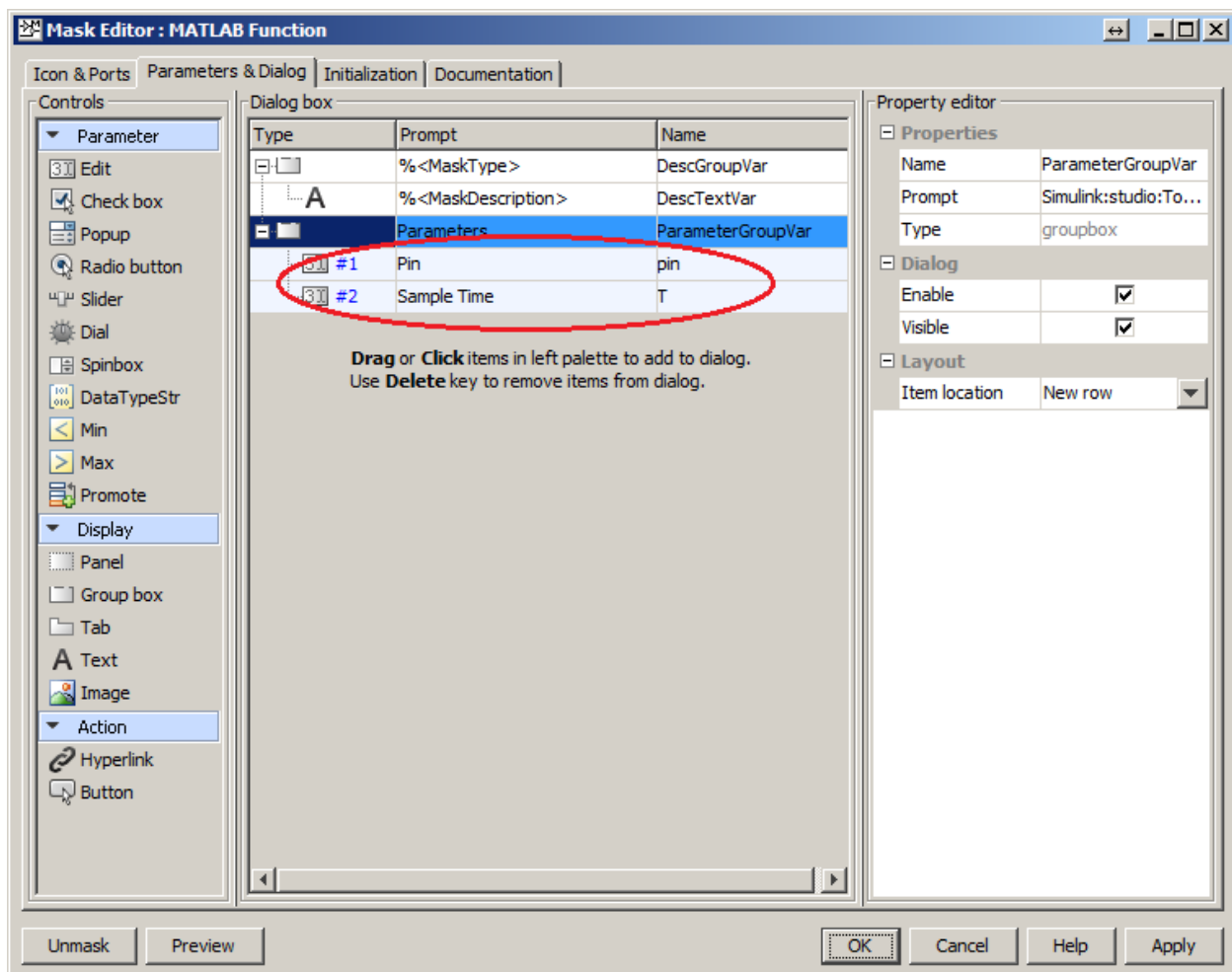
This (the necessary MATLAB command to be executed) can be written in the mask documentation (or even in a text box in the Simulink model), so that users are reminded to properly set the `CustomSource` parameter.

Another option is to add a [callback](#) to the Simulink block so that the instruction is executed at loading or initialization time (right click on the block, chose “Properties ...”, go to the callback tab, and select, for example “InitFcn”).

## Output driver block: masking the MATLAB Function

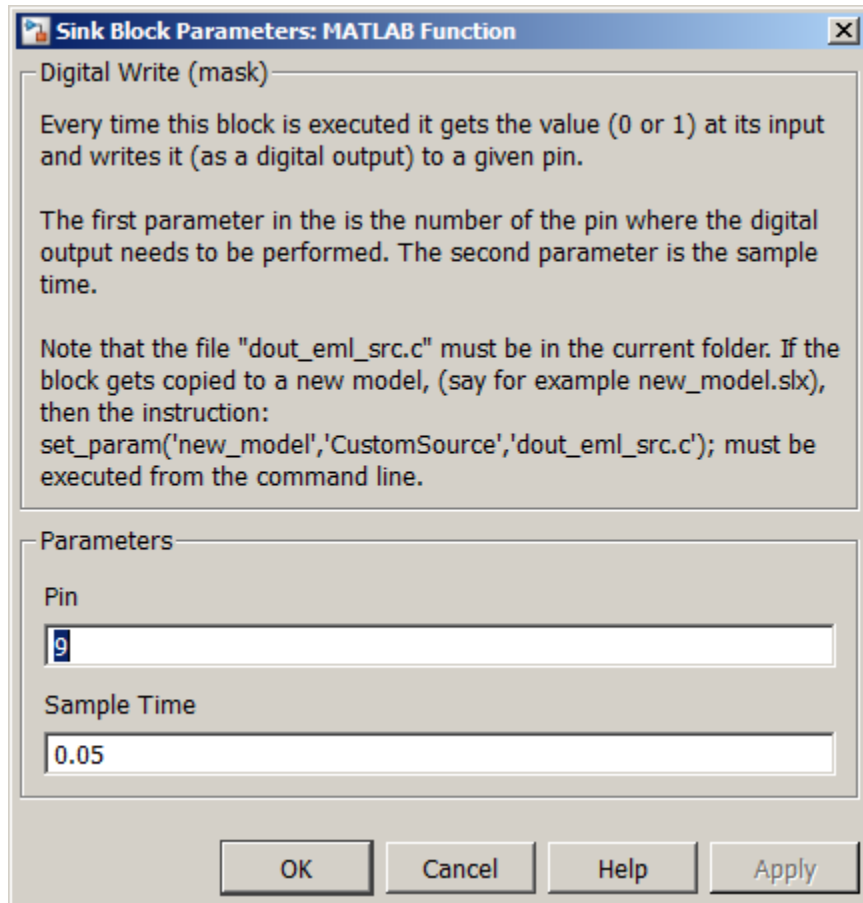
The MATLAB Function block can be masked following pretty much the same procedure described in the previous pages (23-26) for the S-Function Builder block.

As shown in Figure 24, both a Pin (`pin`) and a Sample Time (`T`) parameter can be added to the mask so that their values can be selected by the user and passed to the underlying MATLAB Function block. Some pertinent documentation can be also added in the Documentation pane.



**Figure 24: Masking the MATLAB Function block**

Differently from the case of the S-Function Builder block, the user can access the mask (Figure 25) just by double clicking on the block.



**Figure 25: MATLAB Function mask**

Note that, for driver blocks created using the MATLAB Function approach, users will never need to build or re-build any S-function before deploying the whole Simulink model to the target.

This is indeed a considerable advantage over the S-Function Builder approach, in which users need to remember to explicitly build the S-function before using the block, and indeed need to remember to rebuild the S-function every time a parameter is changed (although if the S-Function Builder block is masked, the S-function needs to be rebuilt only when the Sample Time is changed).