

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

## Contents:

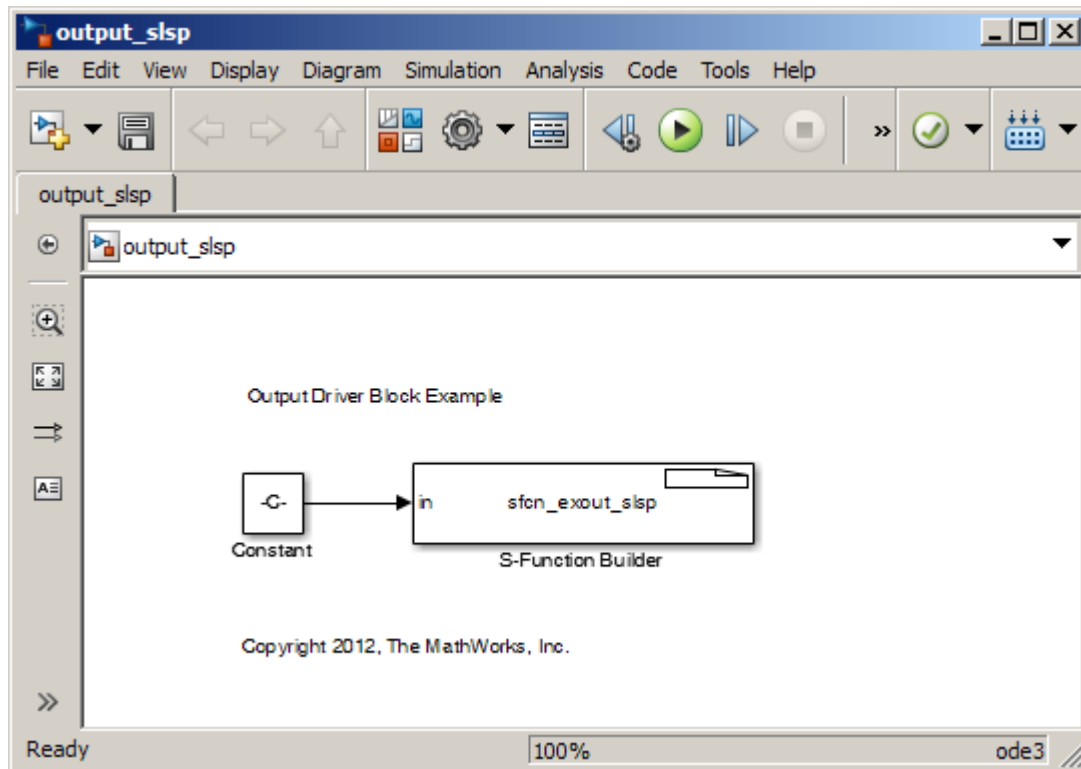
Writing a Simulink Device Driver block: a step by step guide.....	1
What is a device driver block? .....	3
Programming Hardware with Targets.....	4
A quick look at the selected target .....	5
Where is my Simulink model running? .....	5
External Mode.....	8
An overview of approaches for creating device drivers .....	8
The S-Function Builder Approach .....	9
Output driver block: Data Properties pane.....	9
Output driver block: Build Info pane .....	14
Output driver block: Initialization pane .....	15
Output driver block: Discrete Update pane.....	16
Output driver block: Outputs pane.....	18
Output driver block: Libraries pane .....	20
Auto-generated sfcn_exout_slsp_wrapper.c file:.....	22
Working with external libraries .....	23
Input driver blocks, and older versions of the Arduino IDE .....	25
Troubleshooting: Undefined Reference .....	25
Troubleshooting: Variable not defined in this scope.....	26
Masking S-Function Builder blocks .....	27
The Legacy Code Tool Approach.....	31
Example: Creating an Analog Output driver block.....	32
Example: Creating a Digital Input driver block.....	35

The MATLAB Function Approach .....	39
Output driver block: Data Manager and MATLAB code .....	41
Output driver block: the C++ code.....	44
Output driver block: including the C-file in the build process .....	46
Output driver block: masking the MATLAB Function .....	47
The MATLAB System Block Approach .....	50
Anatomy of a System Object -based Device Driver .....	51
Creating a Digital Write Block Using System Objects .....	53
Developing the Wrapper Functions Implementing Digital Write .....	54
Create a System Object.....	55
Define Properties .....	56
Define Inputs.....	56
Define Initialization, Output and Termination Functions .....	57
Define Build Artifacts .....	58
Testing the System Object .....	59
Using System Objects in Simulink with the MATLAB System Block.....	60
Run on Target Hardware.....	61
Additional System Object Examples .....	63

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](#) is used to build the examples, but the method is the same for any other supported (Simulink or Embedded Coder) target.

## What is a device driver block?

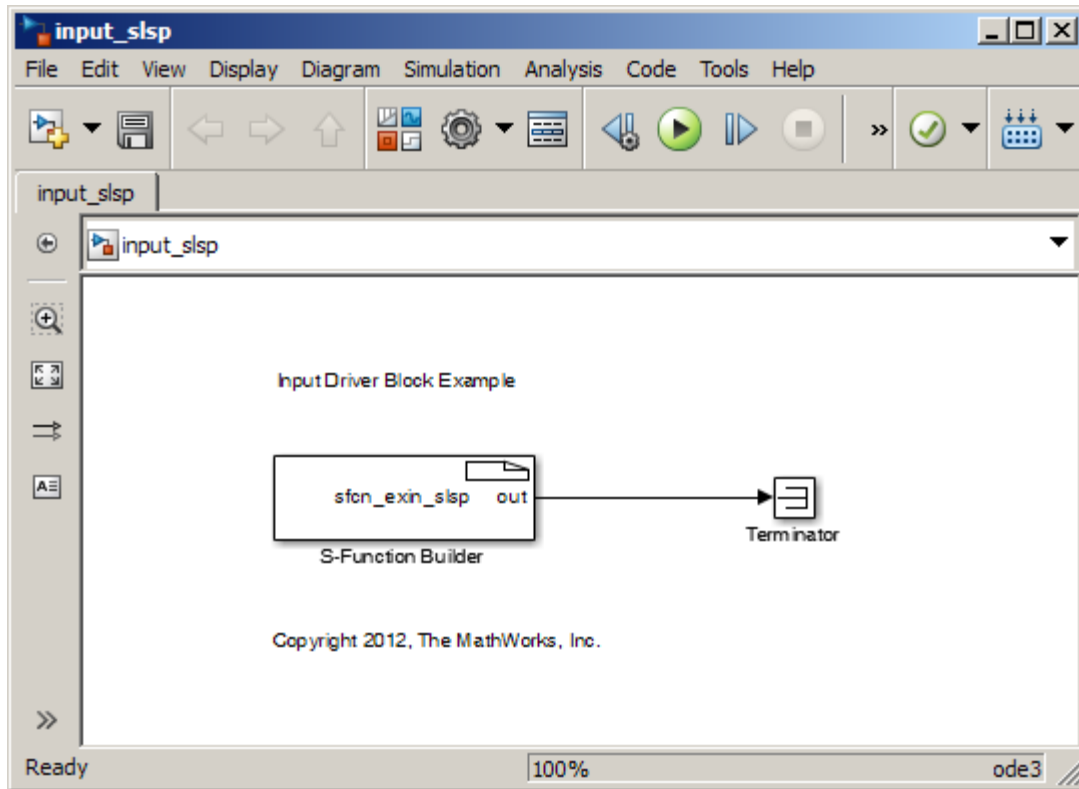
In general, an “Output Device Driver” block takes some signal as input and uses it to perform some kind of real world output (i.e. actuation) on the target platform (e.g. analog or digital write).



**Figure 1: Model containing a custom output driver block**

The model in Figure 1 contains a simple output driver block (implemented here 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**

## Programming Hardware with Targets

Algorithms created in MATLAB and Simulink can be translated into a language suitable to be run locally on hardware platforms. Targets are packages that can perform this translation and deploy algorithms on an embedded platform, where they are ultimately intended to run independently and completely outside of MATLAB/Simulink.

In general, Simulink targets only need Simulink, while Embedded Coder targets also need the Embedded Coder. The naming structure simply references the product needed to have access to the target in question.

Although the examples here use the Arduino support, (which happens to be a Simulink Target), there is no difference in the context of integrating device driver code to any Target.

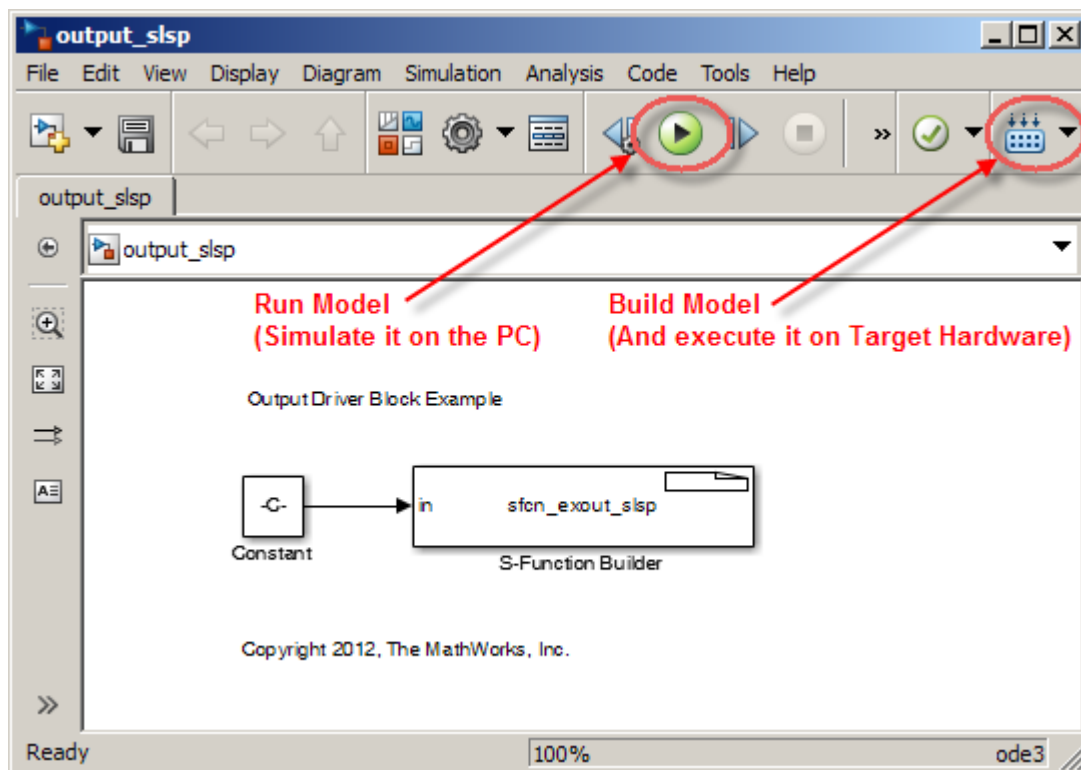
## A quick look at the selected target

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.

## Where is my Simulink model running?

It is important to understand that models such as the ones in the previous figures can be executed (e.g. run) 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 with the “mode” set to “Normal”). 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).

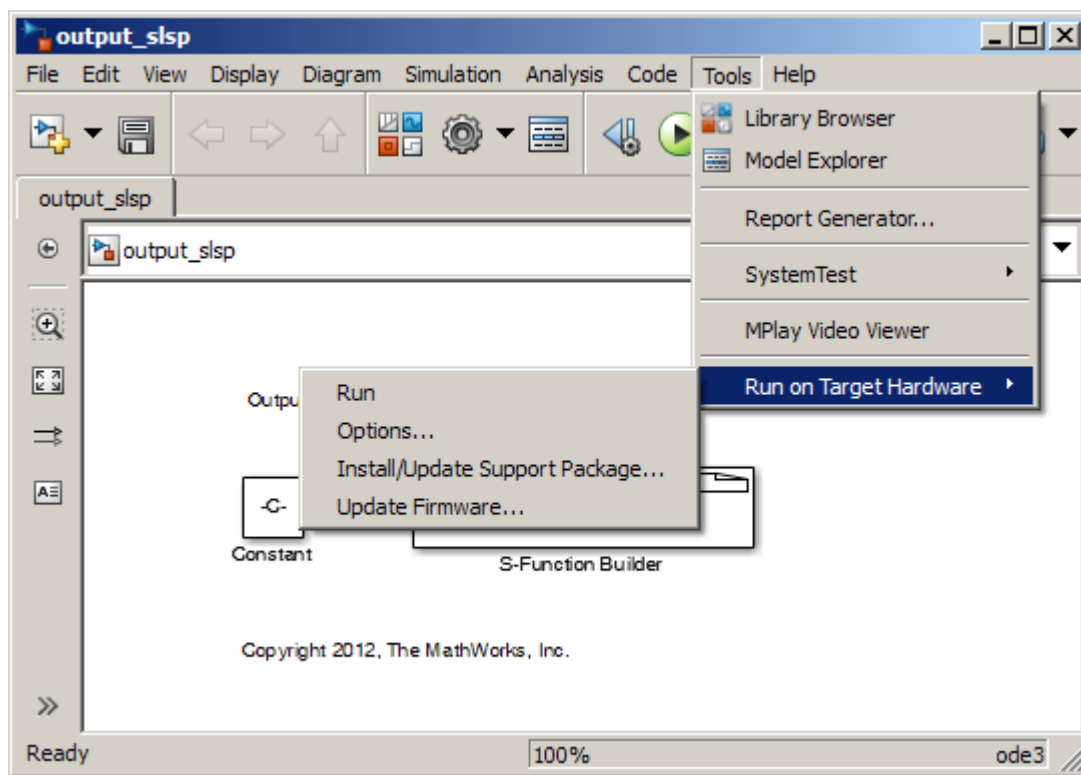
**The other way** in which a model like the ones in the previous pictures can be executed is by generating (from the model) an executable that runs (typically in real time) on the target platform.

Typically – this workflow is initiated with the ‘Build’ button on the right hand side. At the most general case this means generating code for the model, and often includes compiling the code, downloading it to the hardware platform, and initiating execution on that hardware 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).

## External Mode

Note that there is also a mode of execution called ‘External’, which is available on the same pull-down menu as ‘Normal’.

In this execution mode, while the algorithm runs *on the target hardware*, the Simulink diagram on your computer (the host) is also active and acts *as a graphical front end and monitoring tool* for the algorithm executing on the board.

Essentially, additional code is generated (for both the host and the target) which allows for the communication between the Simulink diagram on the host and the executable on the target. This for example allows for a Scope block *running on the board* to *beam back signal data* to the computer, so that the data can be visualized using the Scope GUI in the *Simulink diagram on the computer*. In addition, whenever some parameters in the Simulink diagram on the computer are changed, these changes are automatically downloaded (without recompiling the whole model) to the executable running on the board.

## An overview of approaches for creating device drivers

In this guide, the first method to develop device drivers is based on the S-Function Builder block. Following chapters also describe different methods based respectively on the Legacy Code Tool (LCT), the MATLAB function block, and the System Object block.

In general, the MATLAB Function and System Object approaches tend to work better for developing complex drivers and blocks that have to be redistributed and masked (e.g. because the end user needs to change parameters often). In fact, we generally recommend using the System Object approach, as it is the most flexible, reliable, and powerful method (note however that the System Object approach does not work for versions prior to R2015a).

On the other hand, the S-Function Builder and LCT approaches might be better for developers that are more familiar with C/C++ than (Embedded) MATLAB or System Objects and need to quickly develop simple drivers that don't have a lot of parameters.



Of these last couple of methods, the one based on the LCT is clearer, more elegant and better suited for larger projects, while the S-Function builder has the advantage of being self-contained. Specifically, in the latter approach, all the needed code is contained in the S-Function Builder block and no other file than the Simulink model needs to be given to another user wishing to use the driver block.

An overview of the pros and cons of these approaches (featuring an Extended Kalman Filter example implemented in different ways) can also be found here: <http://www.mathworks.com/matlabcentral/fileexchange/46786>

## The S-Function Builder Approach

In this chapter, 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 compiler then uses the wrapper file to build executable files for both simulation and on-target execution. If you are unfamiliar with the S-Function Builder, it is definitely a good idea to have a look at the help page for the 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).

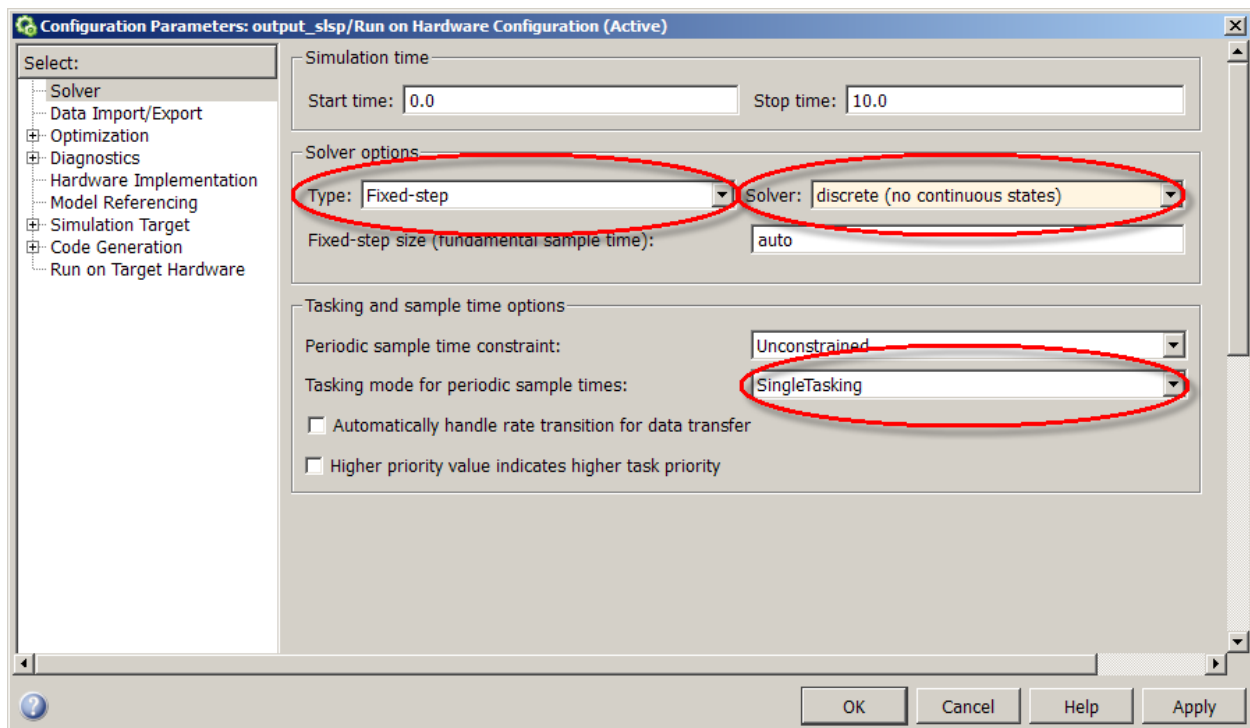
As mentioned previously, alternative approaches to developing device drivers, (based on the Legacy Code Tool, the MATLAB Function block, and the System Object block), are also explained later in this guide, and the relative examples are included in the main zip file downloaded from the File Exchange MathWorks site.

## 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).



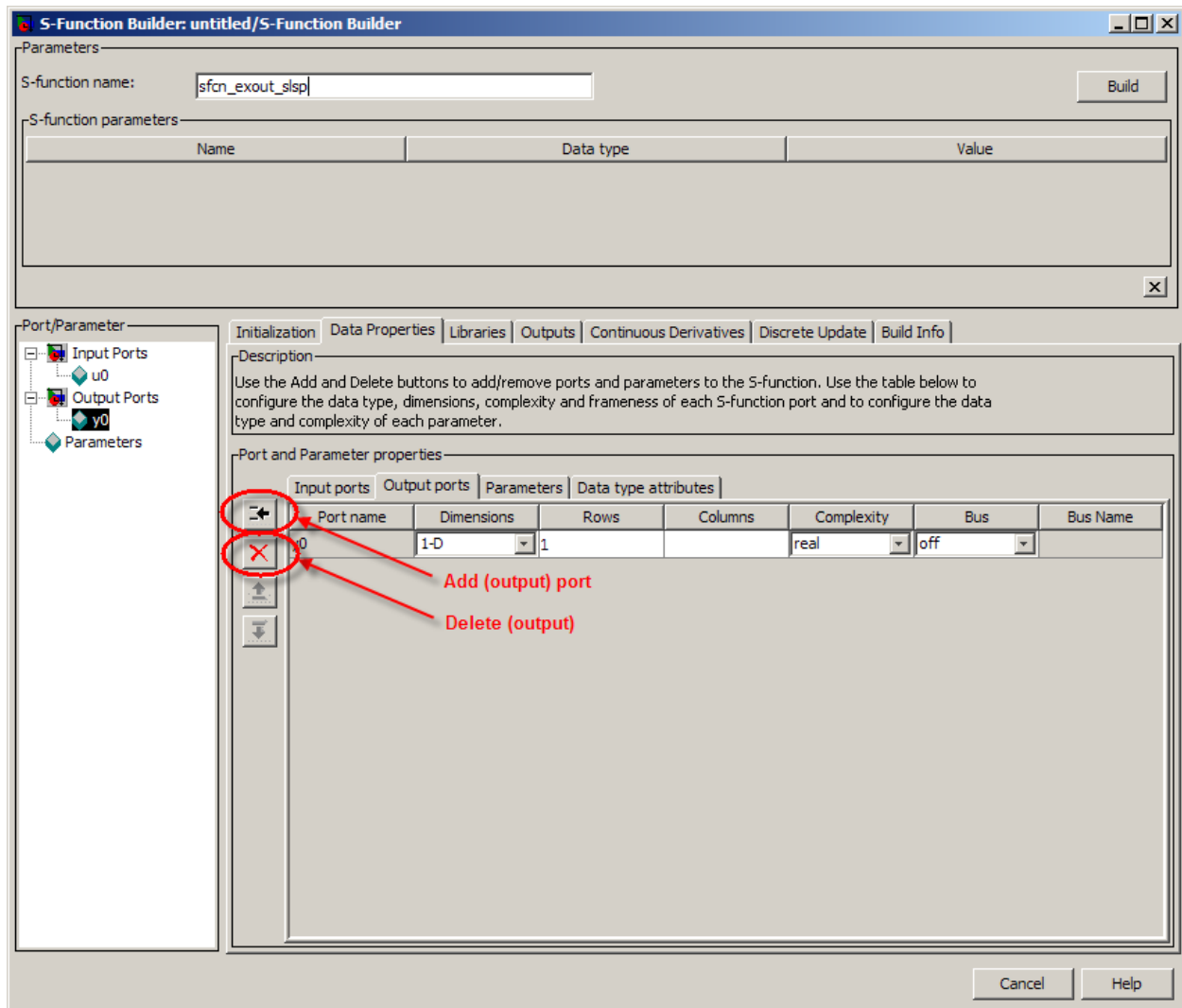
**Figure 5: Solver Configuration Parameters**

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.

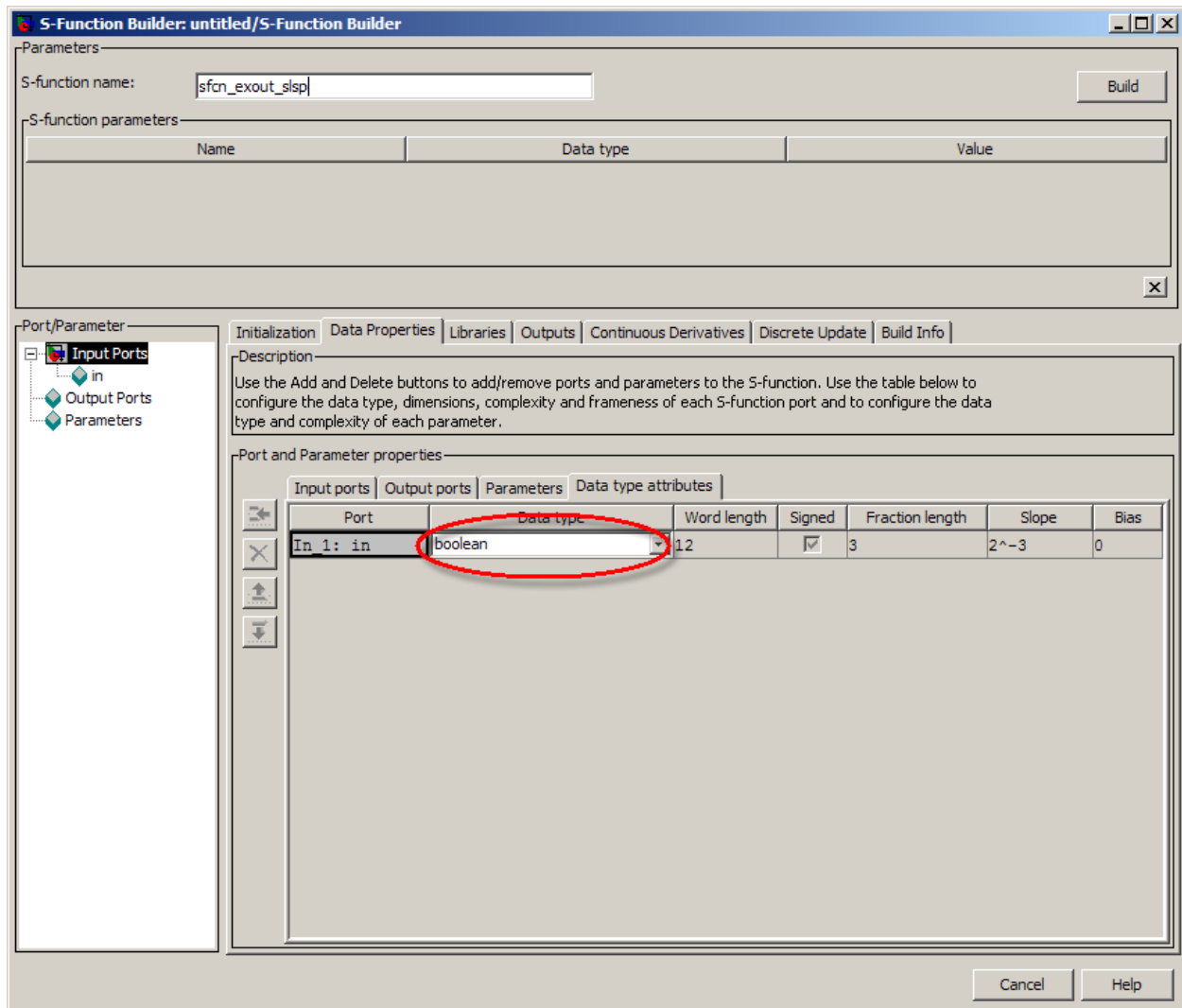


**Figure 6: Data Properties pane: Output ports**

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.

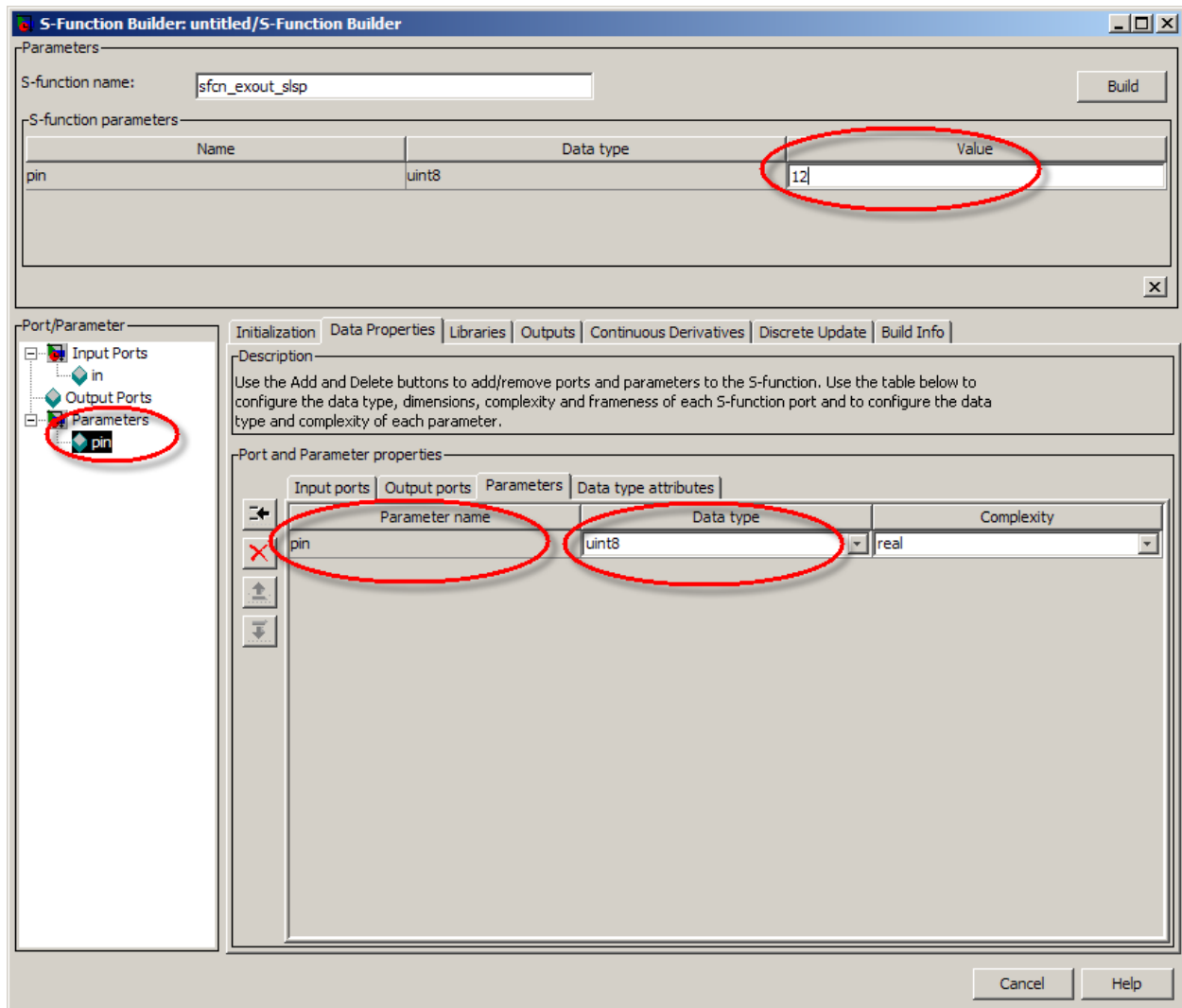
Similarly, by clicking on the “Input ports” subpane we can rename the input port to “in” instead of u0. The value coming from this input port will be later referred to as in[0] in the code.

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.



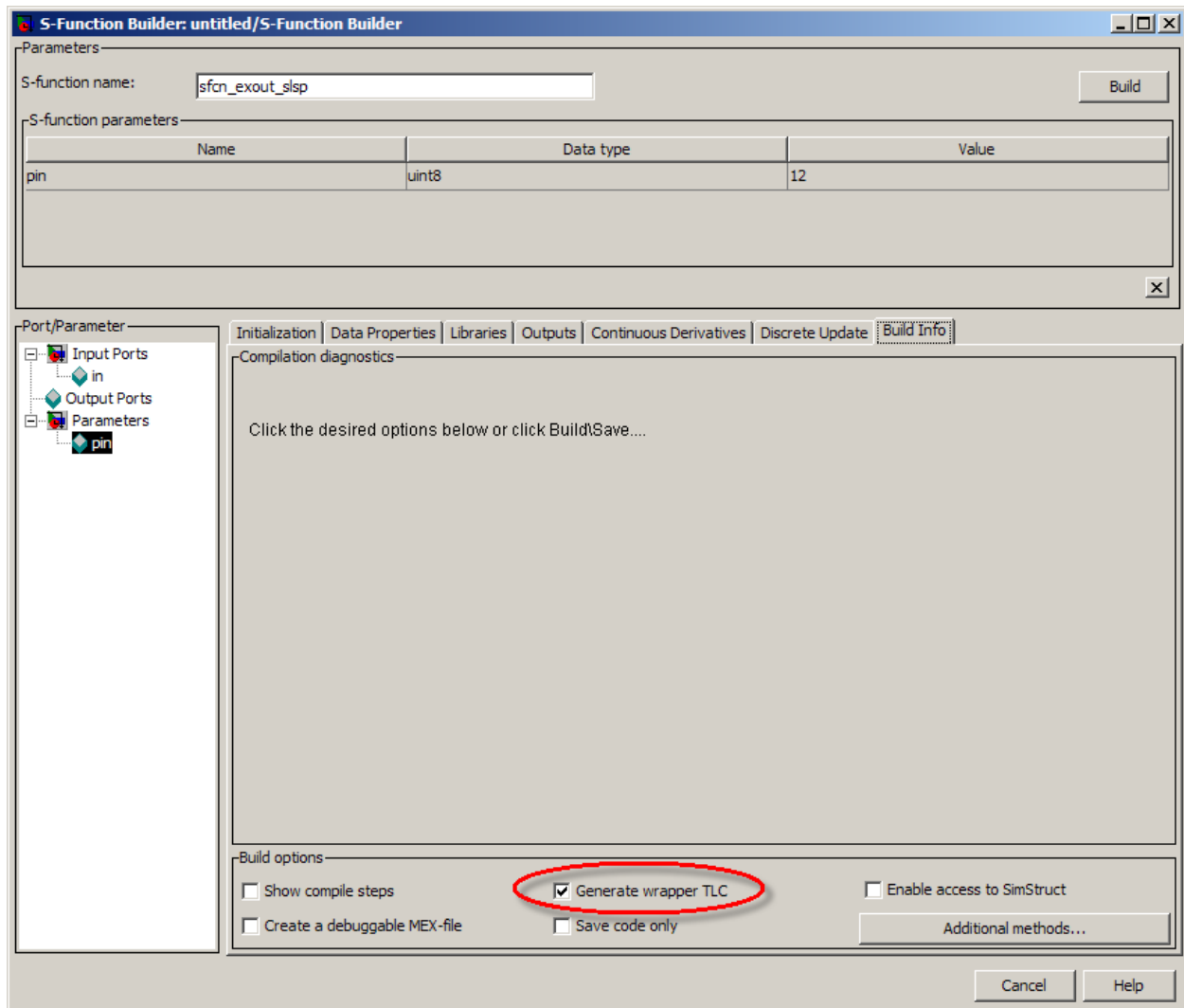
**Figure 8: Data Properties pane: Parameters**

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:

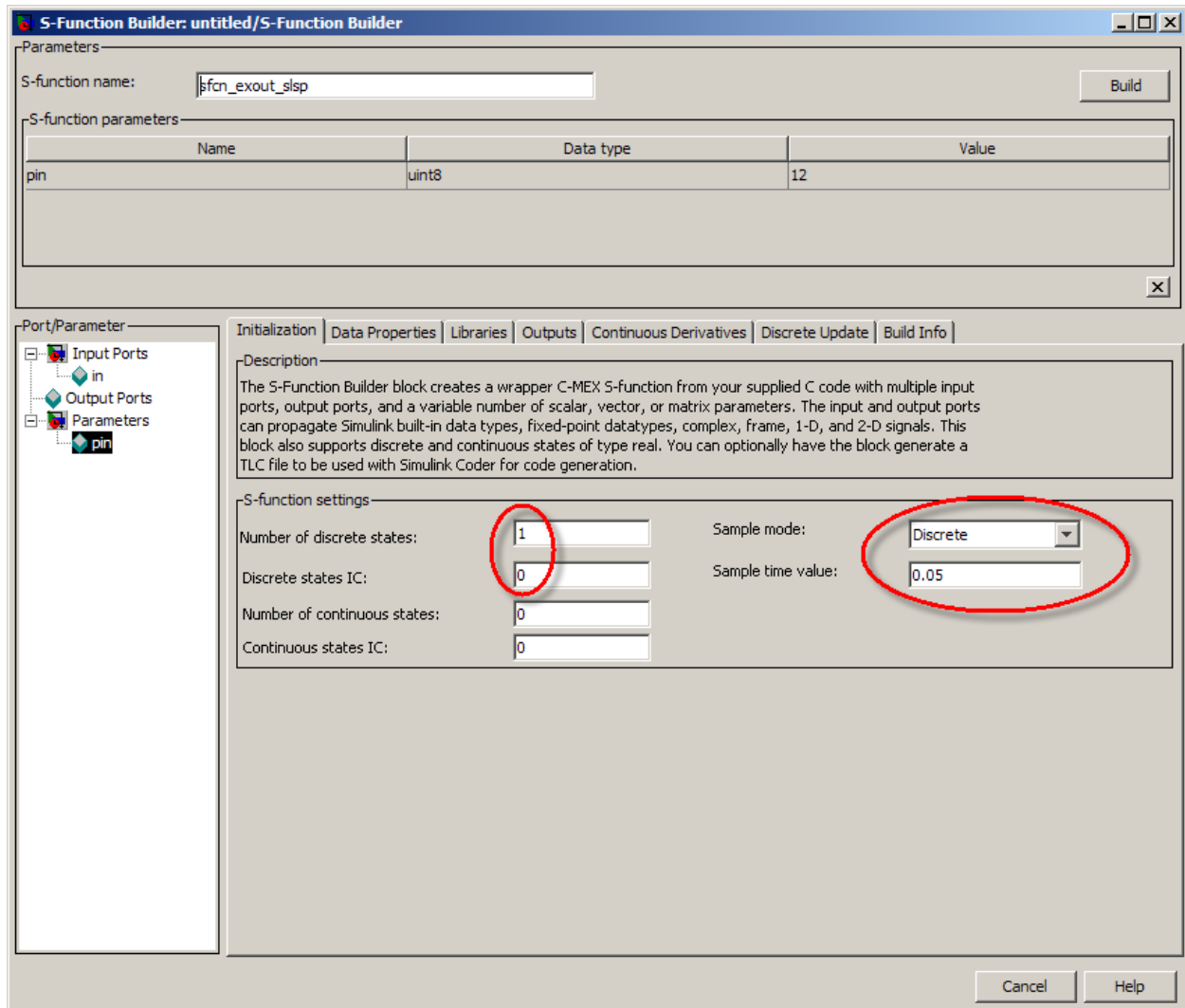


**Figure 9: Build Info pane of the output driver block**

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**

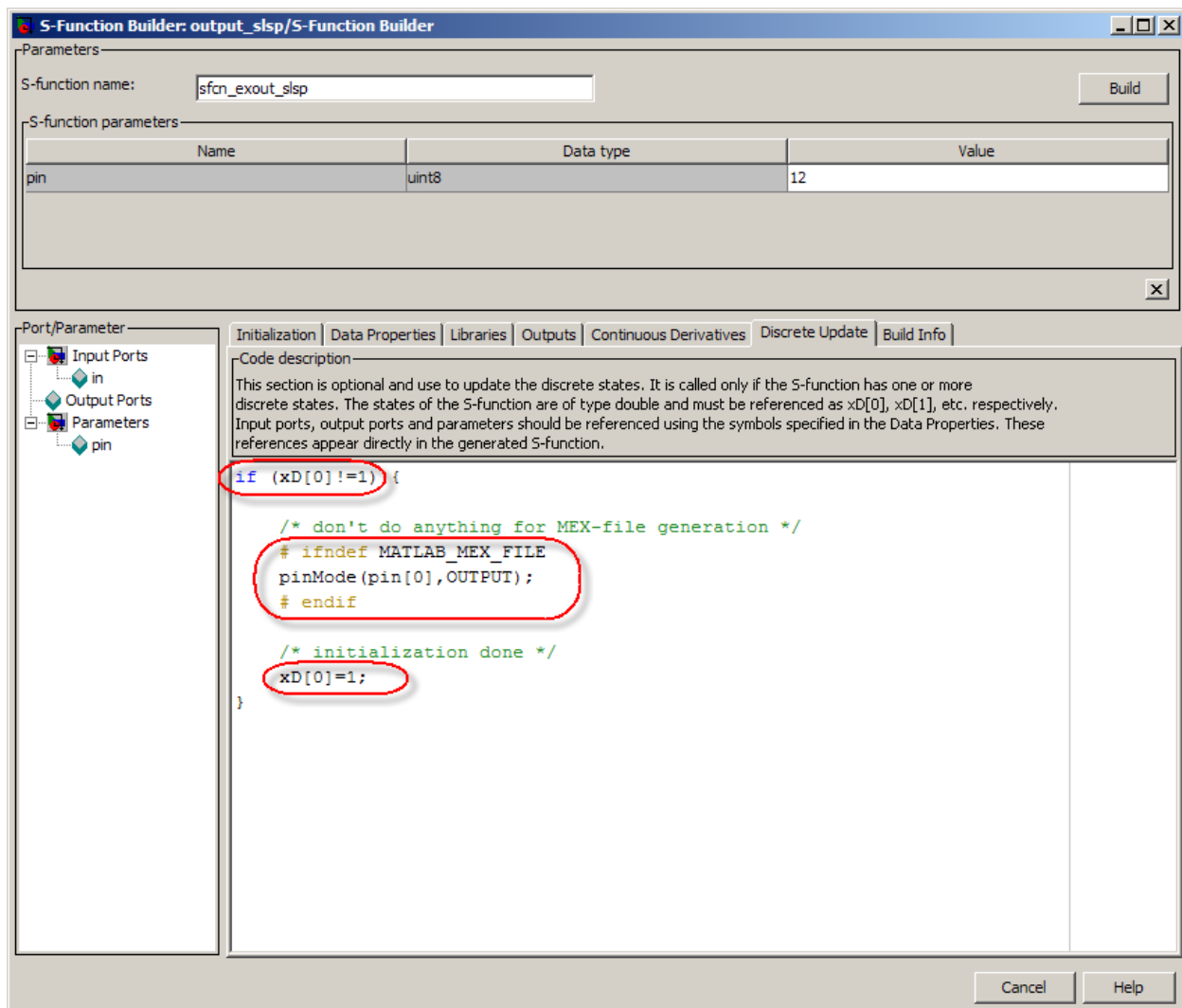
The Initialization pane establishes the block sample time and its number of continuous and discrete states. Normally driver blocks execute in discrete time and have no continuous states. In this case we have chosen to set the sample time to 50ms (see Figure 10) but one could very well select the sample 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  $xD[0]$ ) 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), 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 the 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.

For the Arduino case, the initialization code is the code included between the curly bracket in the “void setup() { ... }” function (make sure you copy and paste only the content of the curly brackets, not the whole setup function). If no initialization code is needed, then this line (but only this line, that is “pinMode ...”) 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.

## Output driver block: Outputs pane

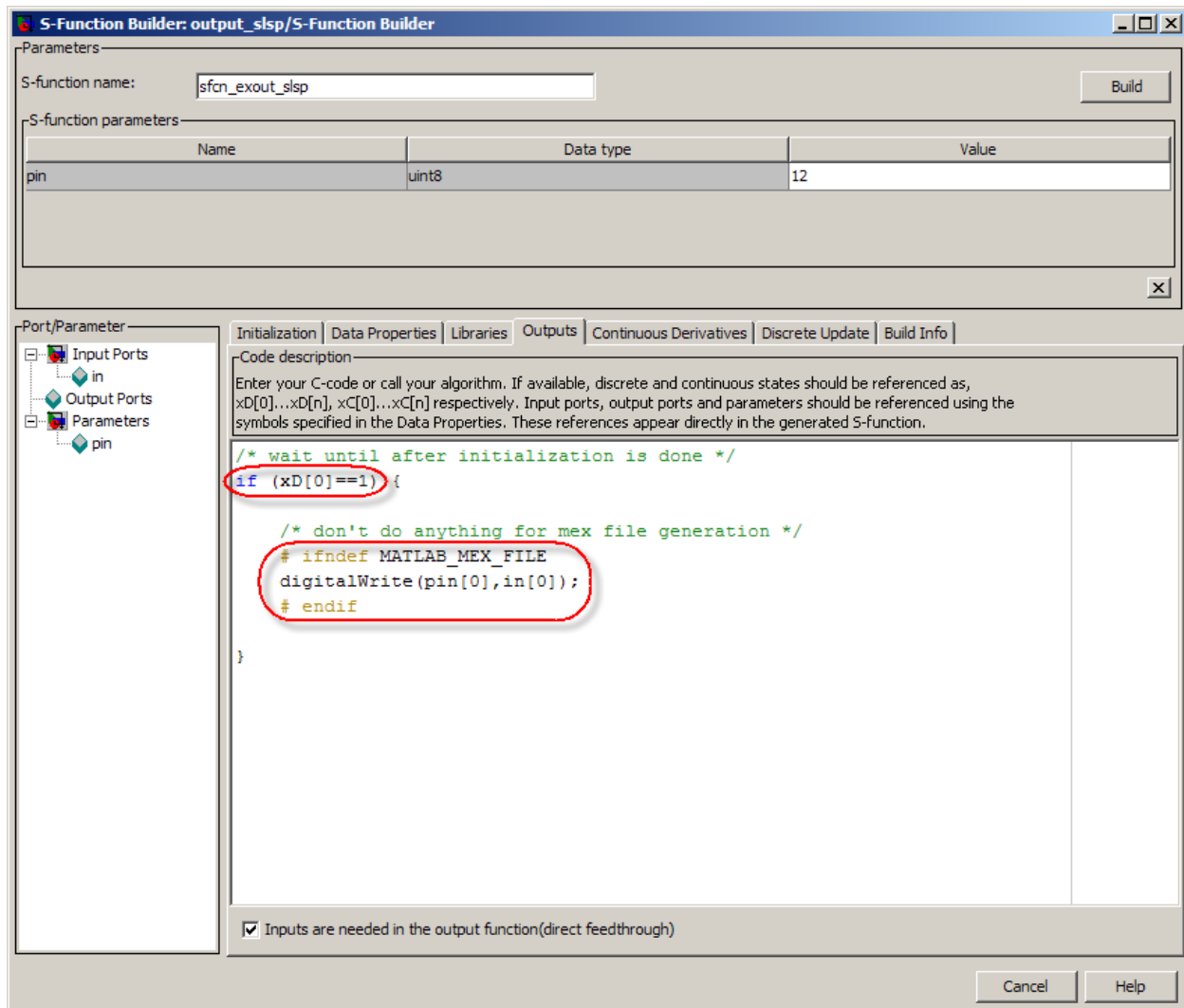
The Outputs update pane defines the actions that the block performs (in general on its outputs), when it is executed. As for the discrete update case, we can type the code directly in the edit field.

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 will not include any output code, and therefore will not do anything. Conversely, the

executable generated for execution on the Arduino will include the digital write line. When this code will be 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.



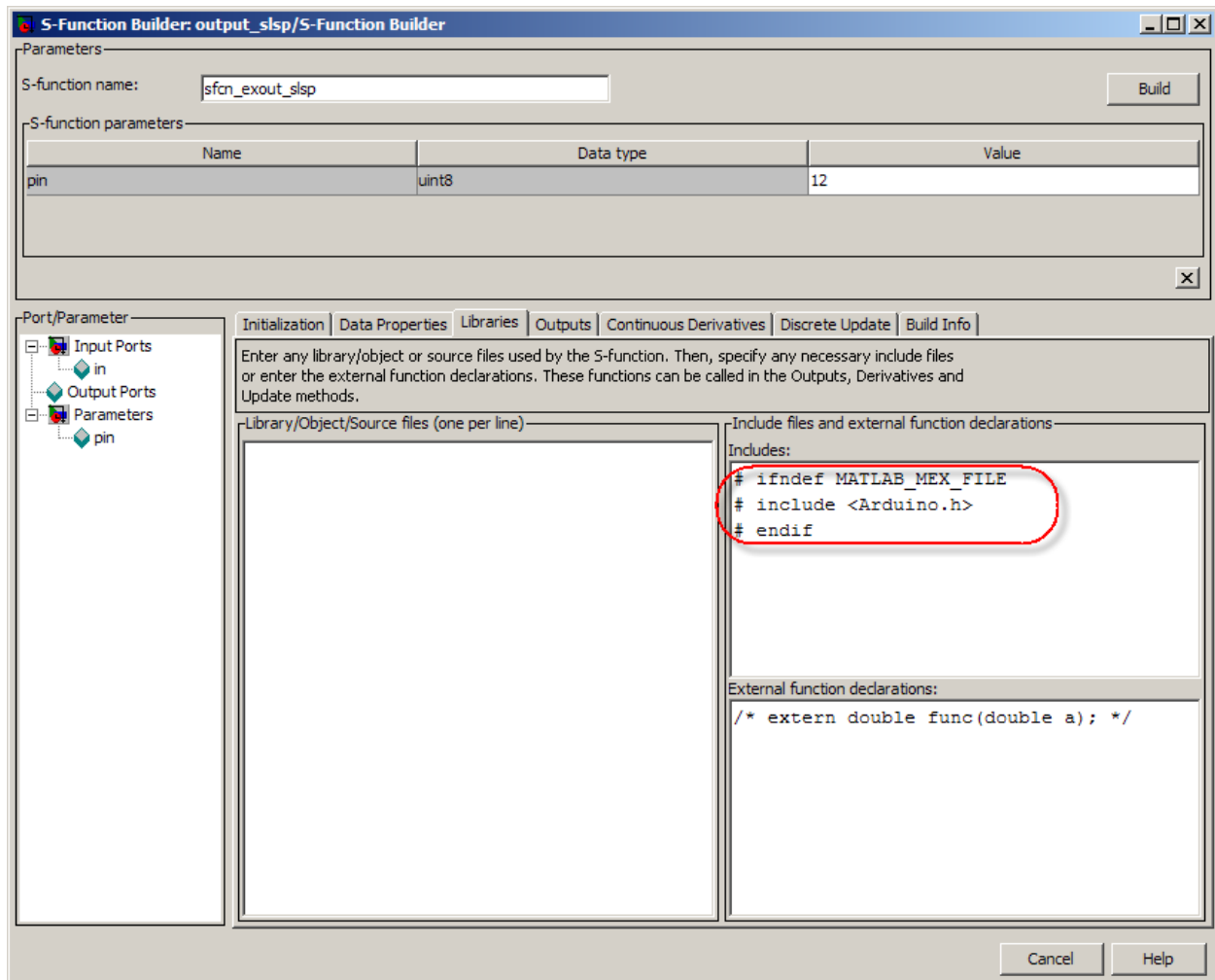
**Figure 12: Outputs pane**

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 (for the Arduino case, this is the code contained between the curly bracket in the “`void loop() { ... }`” function).

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 the code from both the Outputs and Update functions is placed inside a function means, among other things, that any variable defined inside such 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



**Figure 13: 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.

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 in general result in code that is less clear. More importantly, since global variables are *shared within the whole model*, 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 that is not numeric, so can't fit in xD).

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

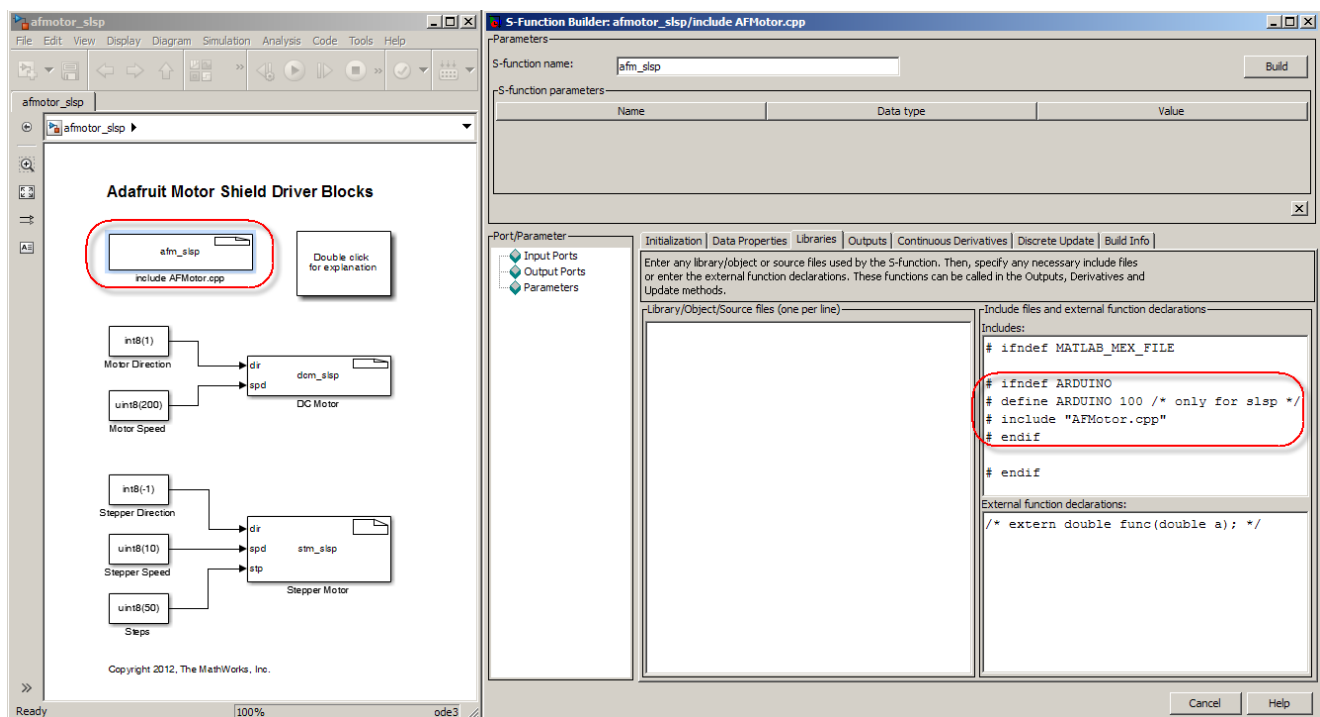
One of the challenges in working with external libraries (that is libraries added later that are not part of the standard distribution of the target) is that the compiler might not know where to find them.

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 when, 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 (better) 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, it 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 might also work).

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.

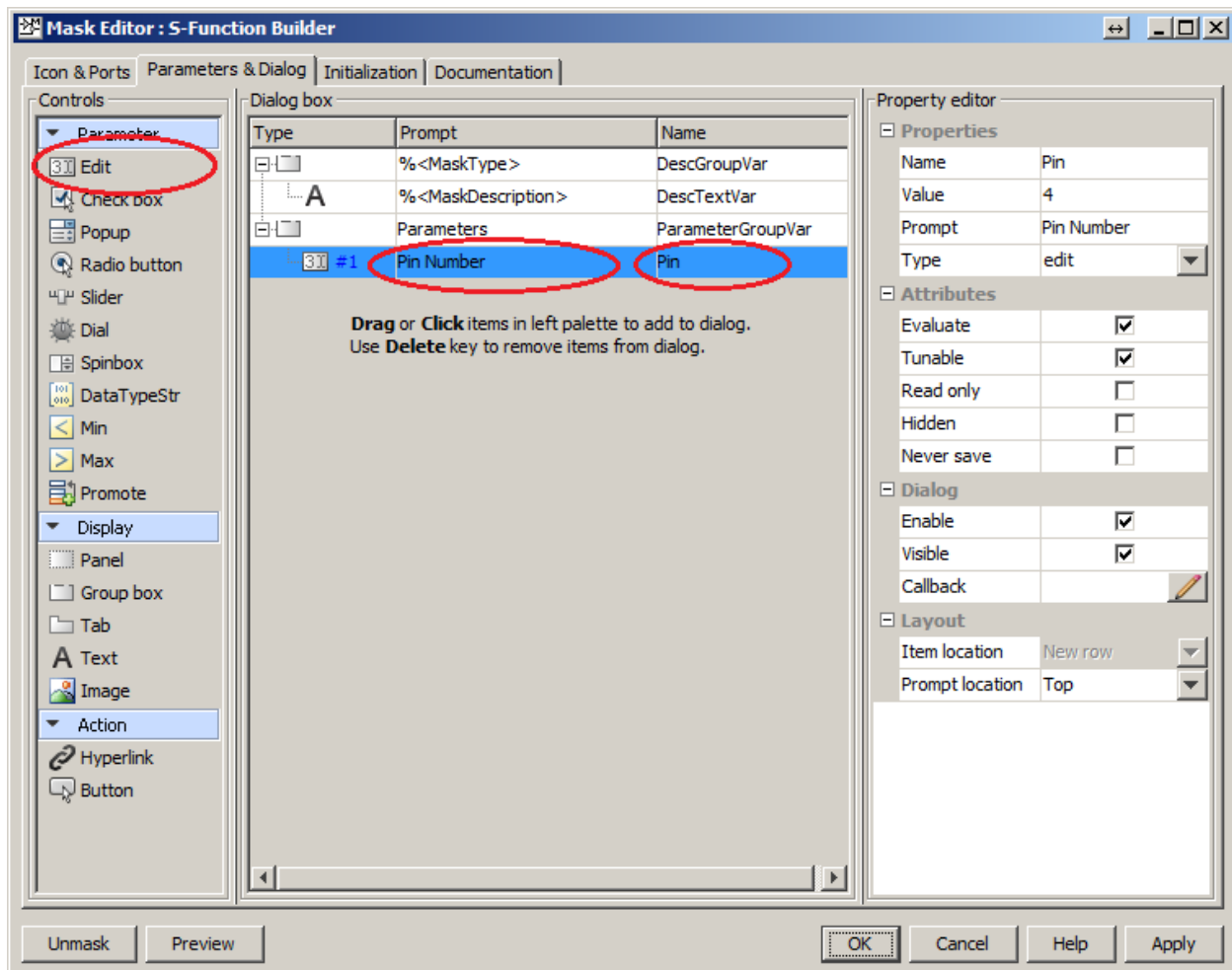
The easiest solution is to define the variable as global, (alternatively you could instead use additional elements of the state vector). Global variables can be defined after all the include directives inside the Includes field of the Libraries pane (see page 16). As an example you might also look at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

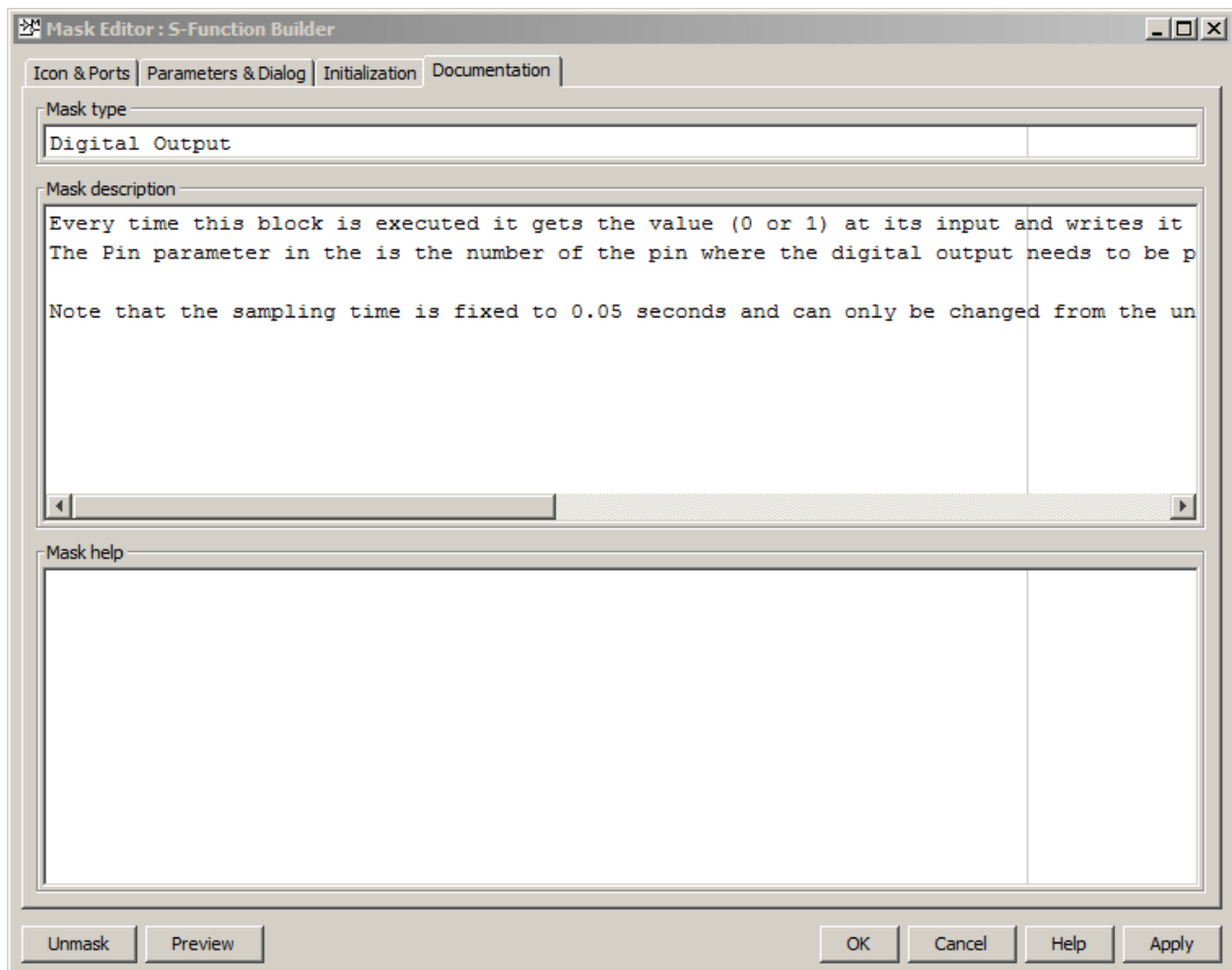
It is possible to [mask](#) an S-Function Builder block as with any other Simulink block. To do so, select the block and either right click and select “Mask” then “Edit Mask” from the menu, or simply use the keyboard shortcut “Ctrl+M”.

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):

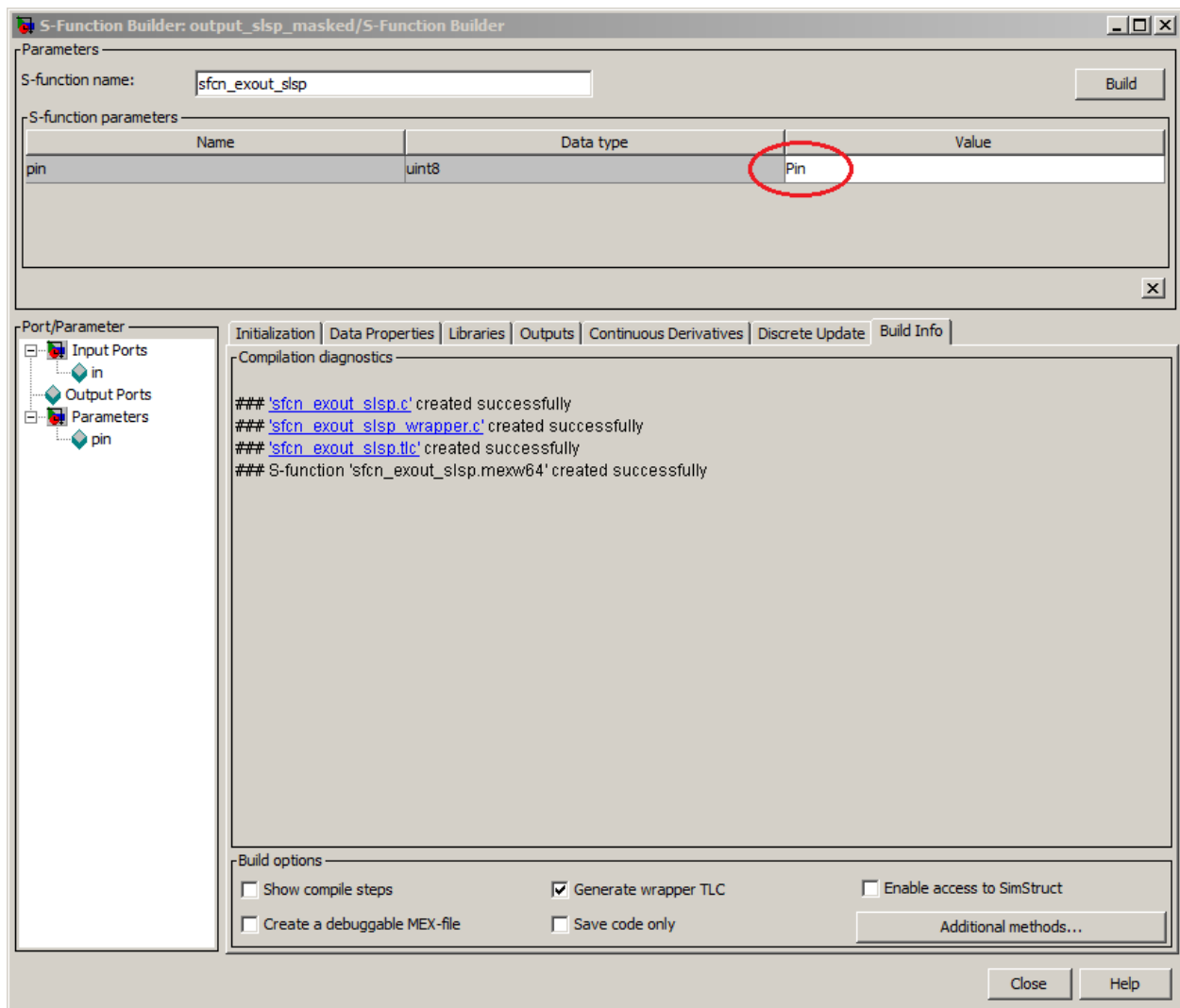


**Figure 16: Documentation pane for the digital output block mask.**

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.

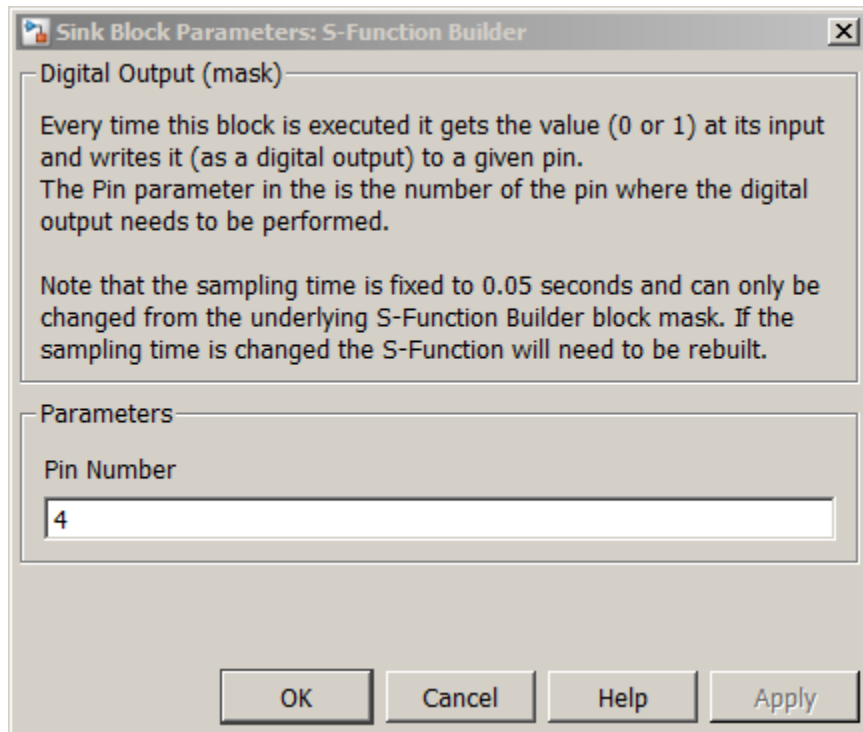


**Figure 17: “Pin” variable passed to the underlying S-Function.**

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 other words, 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 Legacy Code Tool Approach

The [Legacy Code Tool](#) (LCT) is a Simulink tool that allows users to integrate existing C (or C++) functions in a model for simulation and code generation, and that can therefore be very easily used to develop device drivers.

Note that this approach relies on at least two separate files that need to be shipped together. The first is a C (or C++) file that handles the lower level interactions with the target. The other is a MATLAB file needed to define a data structure (which includes information such as the functions to be called and their location, inputs and outputs), and (based on the defined data structure) to build the files needed for both simulation and on-board execution. It is also a good idea to also give to the user a Simulink model containing the actual LCT block which calls the driver executable, although in principle this latter block can also be automatically generated from the previously mentioned MATLAB file.

In the following sections of this guide, we assume that you are familiar with the Legacy Code Tool concepts. Please refer to the documentation and to the various examples shipping with Simulink.

As explained at the beginning of this guide (pages 4-6), when creating device drivers for Simulink in general you have to consider two cases: Running the simulation in Simulink and executing the model on the hardware. In essence your device driver code is compiled for each of the cases separately. So when implementing your device driver you would need to make sure that in the simulation case you do not call functions that are only intended to be executed on the hardware, but can't be executed on your computer where you run Simulink.

There are different ways how you can handle this: One of them would be to make your C-code use the `MATLAB_MEX_FILE` macro (as is done for the S-Function Builder approach). This macro is automatically defined when your code is compiled for the simulation within Simulink, and it is not defined when your code is compiled for the execution on the hardware. However, this approach likely requires you to edit the device driver code snippets. The Legacy Code Tool allows for another (simpler and better) approach, which allows you to use most existing code snippets as they are.

## Example: Creating an Analog Output driver block

Consider this C-code snippet for performing Analog Output operations on an Arduino:

```
#include <Arduino.h>
#include "aout_arduino.h"

extern "C" void aout_init(uint8_T pin) {
    pinMode(pin, OUTPUT);
}

extern "C" void aout_output(uint8_T pin, uint8_T val) {
    analogWrite(pin, val);
}
```

This code would be stored in the file `aout_arduino.cpp`. In order to integrate that code with your Simulink model using the Legacy Code Tool, you would now create a MATLAB script that defines a data structure and performs several operations with it:

```
% 1) Initialization
def = legacy_code('initialize');
def.SFunctionName = 'aout_sfunk';

% 2) Simulation
def.OutputFcnSpec = 'void NO_OP(uint8 p1, uint8 u1)';
def.StartFcnSpec = 'void NO_OP(uint8 p1)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def, '-DNO_OP=//');

% 3) Executing on Hardware
def.SourceFiles =
{fullfile(pwd, '..', 'src', 'aout_arduino.cpp')};
def.HeaderFiles = {'aout_arduino.h'};
def.IncPaths = {fullfile(pwd, '..', 'src')};
def.OutputFcnSpec = 'void aout_output(uint8 p1, uint8 u1)';
def.StartFcnSpec = 'void aout_init(uint8 p1)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
legacy_code('rtwmakecfg_generate', def);

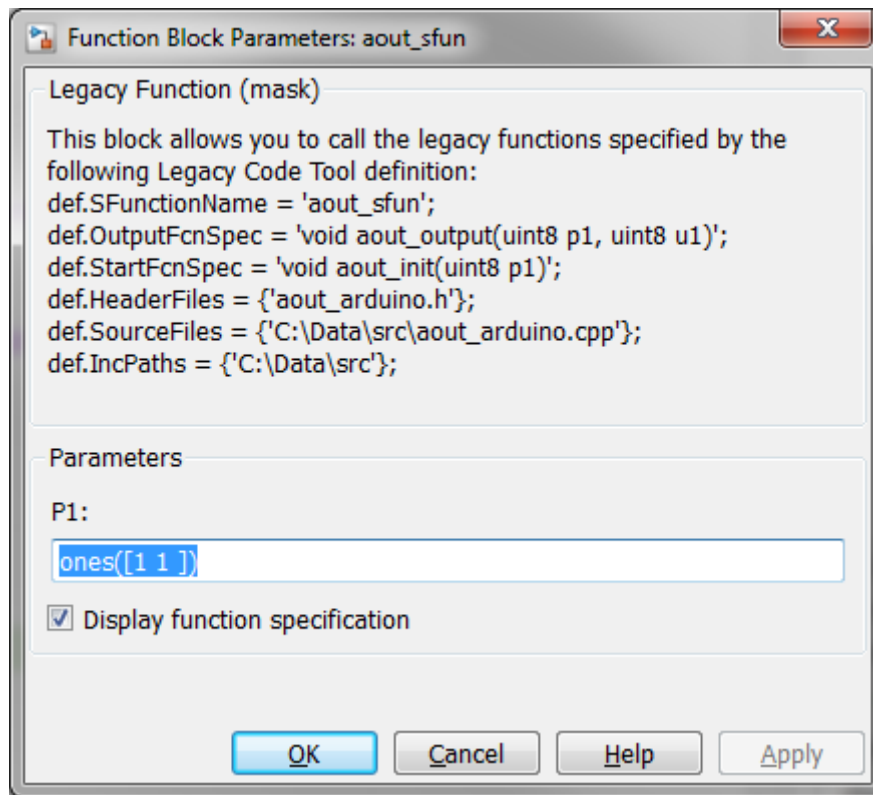
% 4) Create Simulink block
legacy_code('slblock_generate', def);
```



The following paragraphs discuss the four code blocks from the above script in more detail:

- 1) This part initializes the Legacy Code Tool data structure. For all/remaining properties, it would use default values, such as the Sample Time. By default this is set to be “inherited”. Since our script never sets the SampleTime property, eventually our Analog Output block will inherit its sample time from the signal that is driving it. Additionally the first part defines the S-Functions name.
- 2) This second part specifies the details of the function that needs to be integrated, e.g. data types or number of arguments. In this second code section, we use a trick to avoid calling the actual Arduino functions during the simulation of the model in Simulink. Specifically, the functions pointed by the `OutputFcnSpec` and `StartFcnSpec` properties still use the proper data types and number of arguments, but they have the dummy name “NO\_OP”. Then, just before the S-Function for simulation is compiled (in the last line of this code section), the compiler option: `-DNO_OP=//`, (-D stands for “Define”) replaces every occurrence of “NO\_OP” with “//” therefore effectively commenting out both initialization and output functions. As a result, no Arduino code will be called in the simulation.
- 3) The next step is to re-define the `OutputFcnSpec` and `StartFcnSpec`, properties of the data structure in order to call the real function names for interacting with the Arduino. Here we also specify the location of the source and header files that need to be included. Then we re-generate just the S-Function source code, (this is actually optional, and almost never needed). The last few lines of this section are used to generate the Target Language Compiler (TLC) file and the executable that will run on the target.  
**Note:** For `SourceFiles` and `IncPaths` we use the `fullfile()` and `pwd()` functions to specify the absolute path for the sources and headerfiles. This is done to work around an issue in Release 2014a and older. So if you plan to move your driver files to another location, then you need to re-run the script in order update the generated files.

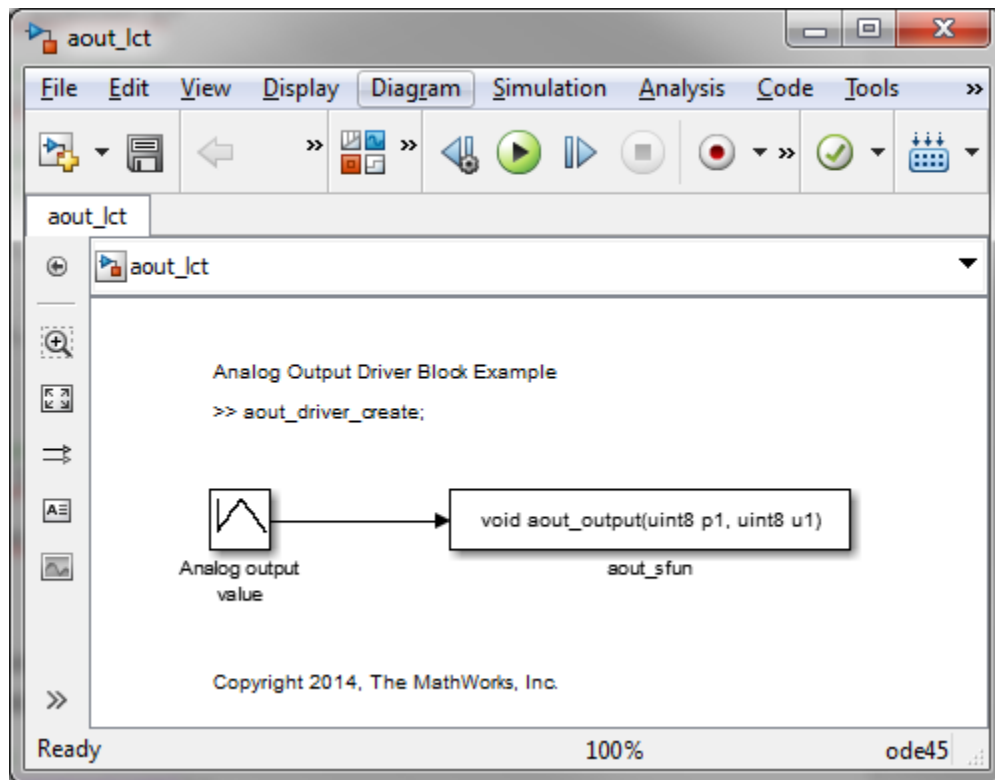
- 4) This code section finally generates an S-Function block, properly annotates and configures it for your integrated function. This block can then be copied into your model. The S-Function block is also automatically masked, so that you can easily provide the parameter. See the Figure below for details of the block mask. Note that the created mask only refers to the parameters as P1, P2, etc. In our example P1 actually is the pin on the Arduino hardware which we want to use for analog output. By default all parameters are set to 1, so make sure you configure your block to use the right pin before running your model on the hardware for the first time



**Figure 19: Mask of the generated Analog Output driver block**

With the generated driver files and block you can then start to create a model around it. The Figure below shows a simple example which also ships as part of this guide. You can run the following MATLAB Command to open the example model:

```
>> aout_lct
```



**Figure 20: Analog Output driver block created by the Legacy Code Tool**

## Example: Creating a Digital Input driver block

The previous example covered an output driver block, i.e. a “Sink” in Simulink terms. This example is about an input driver block, i.e. a “Source” block. In this case there are a few more things to consider, e.g. specifying the SampleTime for the driver block. This is the Arduino code snippet to integrate for Digital Output functionality:

```
#include <Arduino.h>
#include "digitalio_arduino.h"

// Digital I/O initialization
extern "C" void digitalIOSetup(uint8_T pin, boolean_T mode)
{
    // mode = 0: Input
    // mode = 1: Output
    if (mode) {
        pinMode(pin, OUTPUT);
    }
    else {
```

```

        pinMode(pin, INPUT);
    }
}

// Read logical state of a digital pin
extern "C" boolean_T readDigitalPin(uint8_T pin)
{
    return ((boolean_T)digitalRead(pin));
}

```

The above code snippet can be found in the file “digitalio\_arduino.cpp”. Also see pages 54 and 55 in this guide for more explanation on the included file “digitalio\_arduino.h”, and why is a good idea to use it.

As for the previous example, the next step is then to create a MATLAB script that defines and operated the LCT data structure:

```

% 1) Initialization
def = legacy_code('initialize');
def.SFunctionName = 'din_sfunk';
def.SampleTime = 0.05;

% 2) Simulation
def.OutputFcnSpec = 'boolean y1 = NO_OP(uint8 p1)';
def.StartFcnSpec = 'void NO_OP(uint8 p1)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def, '-DNO_OP=0.0;//')

% 3) Executing on Hardware
def.SourceFiles =
{fullfile(pwd, '..', 'src', 'digitalio_arduino.cpp')};
def.HeaderFiles = {'digitalio_arduino.h'};
def.IncPaths = {fullfile(pwd, '..', 'src')};
def.OutputFcnSpec = 'boolean y1 = readDigitalPin(uint8 p1)';
def.StartFcnSpec = 'void DIN_INIT(uint8 p1)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
legacy_code('rtwmakecfg_generate', def);

% 4) Create Simulink Block
legacy_code('slblock_generate', def);

```

The following paragraphs again discuss the details of the four code sections in the script, mainly focusing on the differences from the previous example:

A Source block in Simulink should specify its Sample Time rather than trying to inherit it. So this first code block (Initialization) also specifies the Sample Time for the device driver (specifically with the “`def.SampleTime`” instruction). Note that the Legacy Code Tool does not support making the `SampleTime` a parameter of the S-Function block. In other words, similarly to what happens for the S-Function Builder block, the sampling time of an S-Function Builder block **cannot be passed** as a variable from a mask, and users wishing to change the sampling time must change the numerical value in the initialization code block and re-run the whole script in order to re-generate the executable file. Similarly to the S-Function Builder block case, this is a limitation to keep in mind when creating device drivers that need to be distributed to users.

Another difference with respect to the analog output driver can be found in the second code block (Simulation). In Simulink, a signal needs to have a value defined in every single time step of the simulation. Therefore, a Source block needs to output an actual signal even if a device driver block does not interact with the actual hardware when running a simulation. Typically, input driver blocks output zero during the simulation. Using the trick described in the first example, we use “`-DNO_OP=0.0; //`” to replace every occurrence of “`NO_OP`” with “`0.0; //`” therefore effectively replacing both initialization and output functions with a 0.0 followed by a comment. Again, as a result, during the simulation, no Arduino code will be called, and the value 0.0 will be given as the block output (because, after the replacement of “`NO_OP`” with “`0.0; //`”, the line that will be given to the compiler will be `y1 = 0.0; //(uint8 p1)`).

The third code section (Executing on Hardware) is slightly different from the one in the analog output example because it relies on the function `DIN_INIT(pin)` which is defined in the “`digitalio_arduino.h`” file as following:

```
DIN_INIT(pin) digitalIOSetup(pin, 0)
```

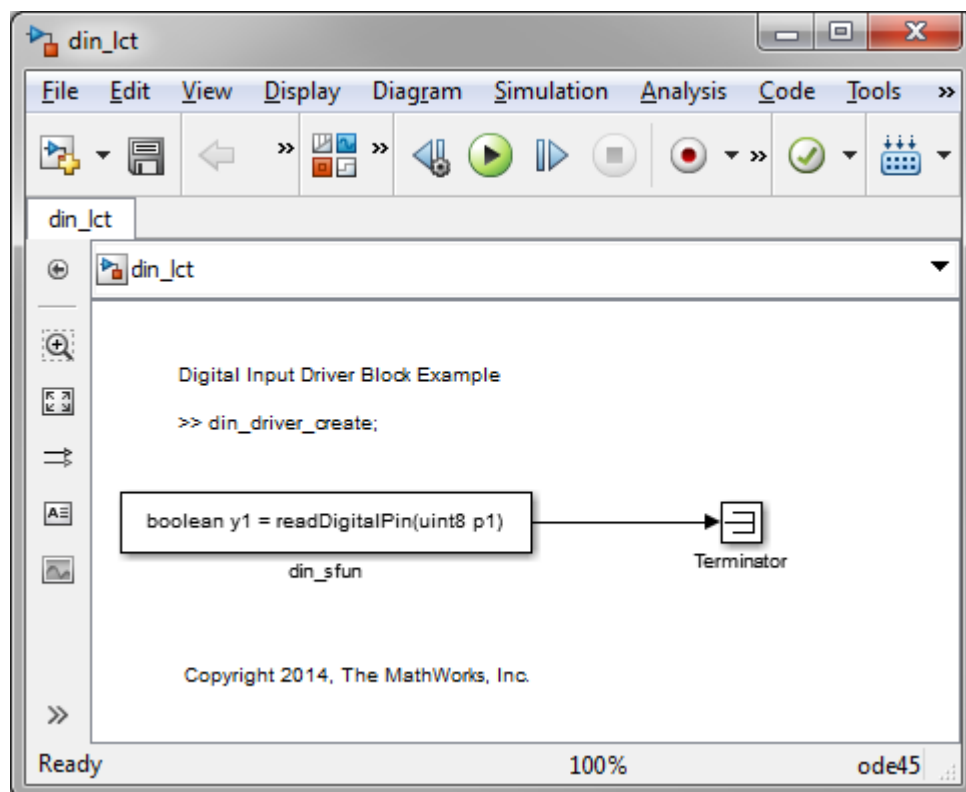
Where the function `digitalIOSetup(pin, 0)` sets up a given pin either as an input (when the second parameter is 0) or as an output (when the second parameter is 1). The reason we had to define the functions `DIN_INIT(pin)` and `DOUT_INIT(pin)` instead of calling `digitalIOSetup` directly is that the LCT syntax for the initialization and output function does not allow for passing

constants directly as input variables. Specifically, it only allows for variables named *u1*, *u2* ... for inputs, *p1*, *p2*, ... for parameters, *work1*, *work2* ... for work vectors and *y1*, *y2* ... for outputs.

As in the analog output example, the last code section (Create Simulink Block) generates the actual S-Function block including a mask that allows you to configure the parameters of this input device driver.

You can now use the generated block and driver files for your model. A simple example ships with this guide, and it is shown in the following Figure. You can open it using the following command in MATLAB:

```
>> din_lct
```

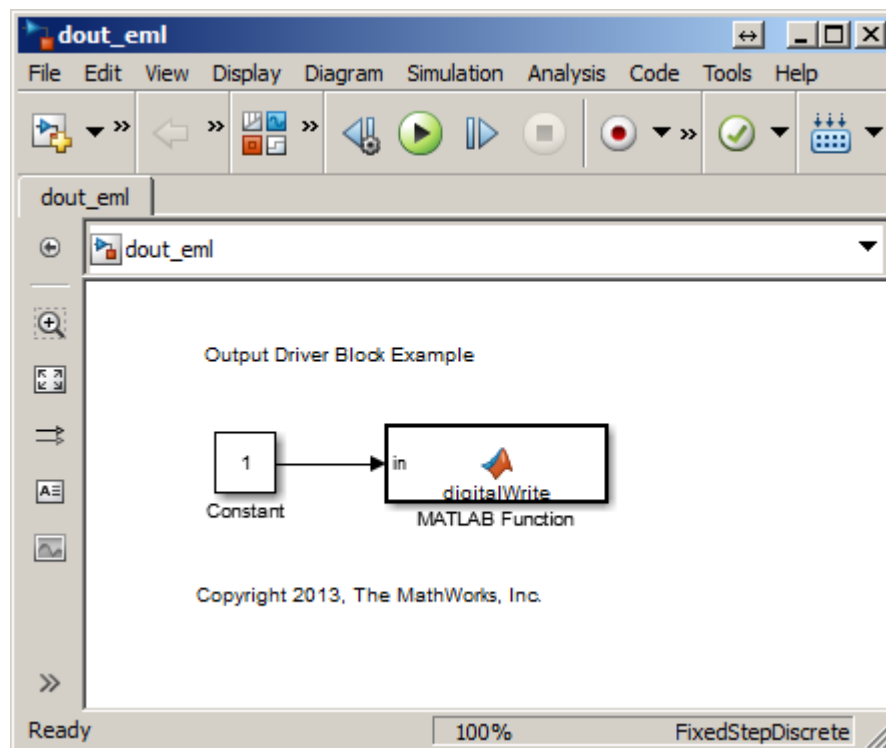


**Figure 21: Model with Digital Input driver block generated by Legacy Code Tool**

## The MATLAB Function Approach

Similarly to the LCT approach, this approach to creating device driver blocks also 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 22), which defines the dimensions and the parameters that can be changed by the user as well as handling the Simulink inputs and outputs.



**Figure 22: 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. Note that this latter source code file can be exactly the same one that works for the LCT (or as we’ll see later for the System Object) approach.

Both the initialization and the output functions defined in the source file are called from the MATLAB code in the MATLAB Function block.

As mentioned at the beginning of this guide, the MATLAB Function approach works better than the S-Function Builder approach for developing driver blocks that are more complex (i.e. they rely on larger source code files), have many parameters that need to be passed to them, and have to be masked and redistributed to a large number of users.

Indeed, 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). Regarding the Legacy Code Tool, while double-clicking on the LCT-generated block does bring up the default mask, often the author of the driver might feel the need for a customized mask, which explains to the user more details about the block’s behavior. In such cases the mask generated by the LCT does not really offer any advantages.

Even more importantly, the fact that the block sampling time can also be passed as a parameter (as opposite to the S-Function Builder and Legacy Code Tool approaches) eliminates any need for the user to look or modify things under the mask, and to ever remember to rebuild (or actually even build in the first place) the S-function. Therefore, this approach be especially useful for end users who might need to change the sampling time relatively often.

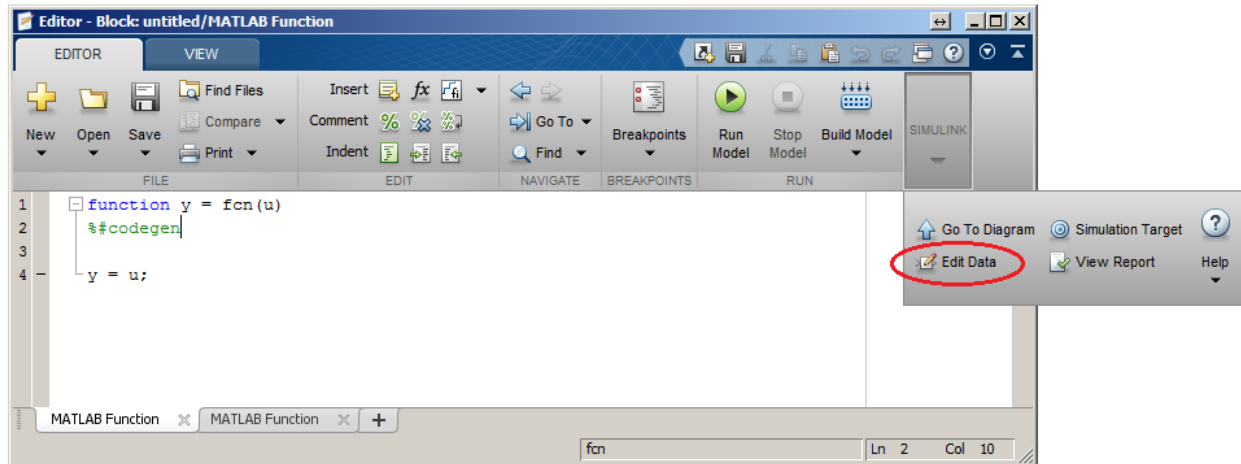
On the other hand, a few additional complexities involved in handling the MATLAB block and interfacing the MATLAB code to both Simulink and C, are some of the disadvantages of the MATLAB Function approach, compared to the previous ones.

Finally, the fact that this approach (as well as the LCT one) relies on two components that need to be managed separately (that is the model and the C/C++ source file) is also something that in some context might be considered as an additional complication for users.



## 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.



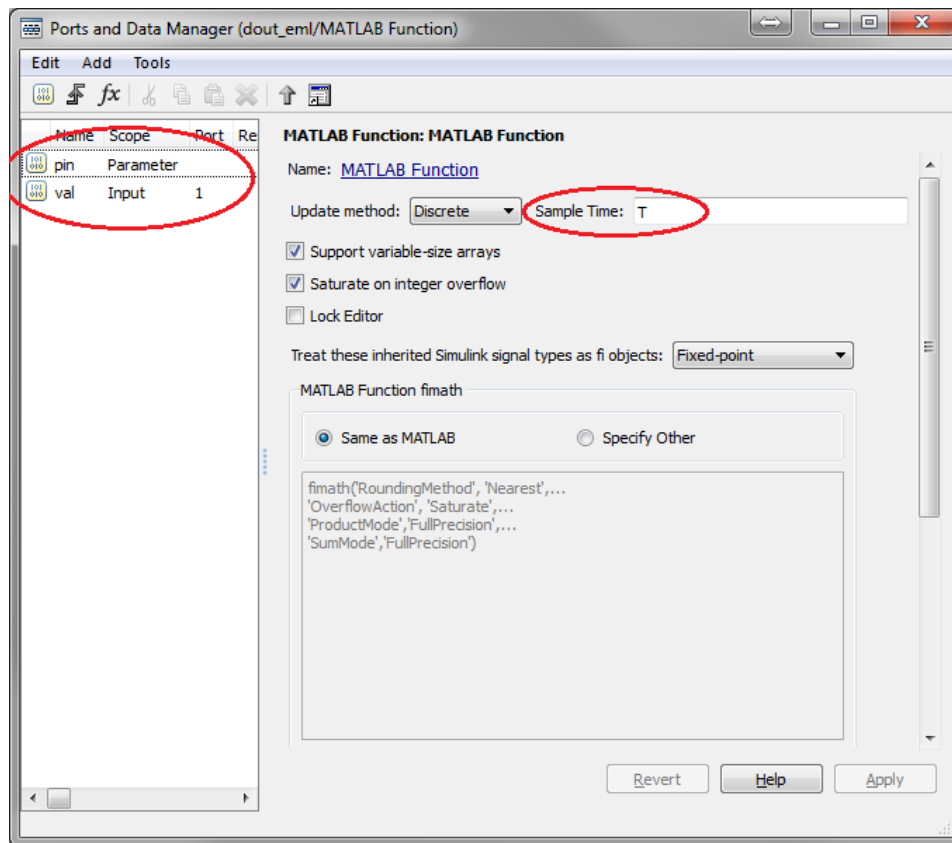
**Figure 23: Editing the MATLAB Function**

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 23) brings up the data manager, which allows you to define inputs, outputs, parameters, and the sample time.

Similarly to the S-Function Builder case (pages 11-13), and to the LCT case (page 32) we can define an input variable (`val`), a pin parameter (`pin`), and the sampling time which is set to a variable `T` meant to be eventually passed by a mask (Figure 24).

To test the driver before the mask is created, `T` can be defined as a normal variable in the MATLAB workspace.



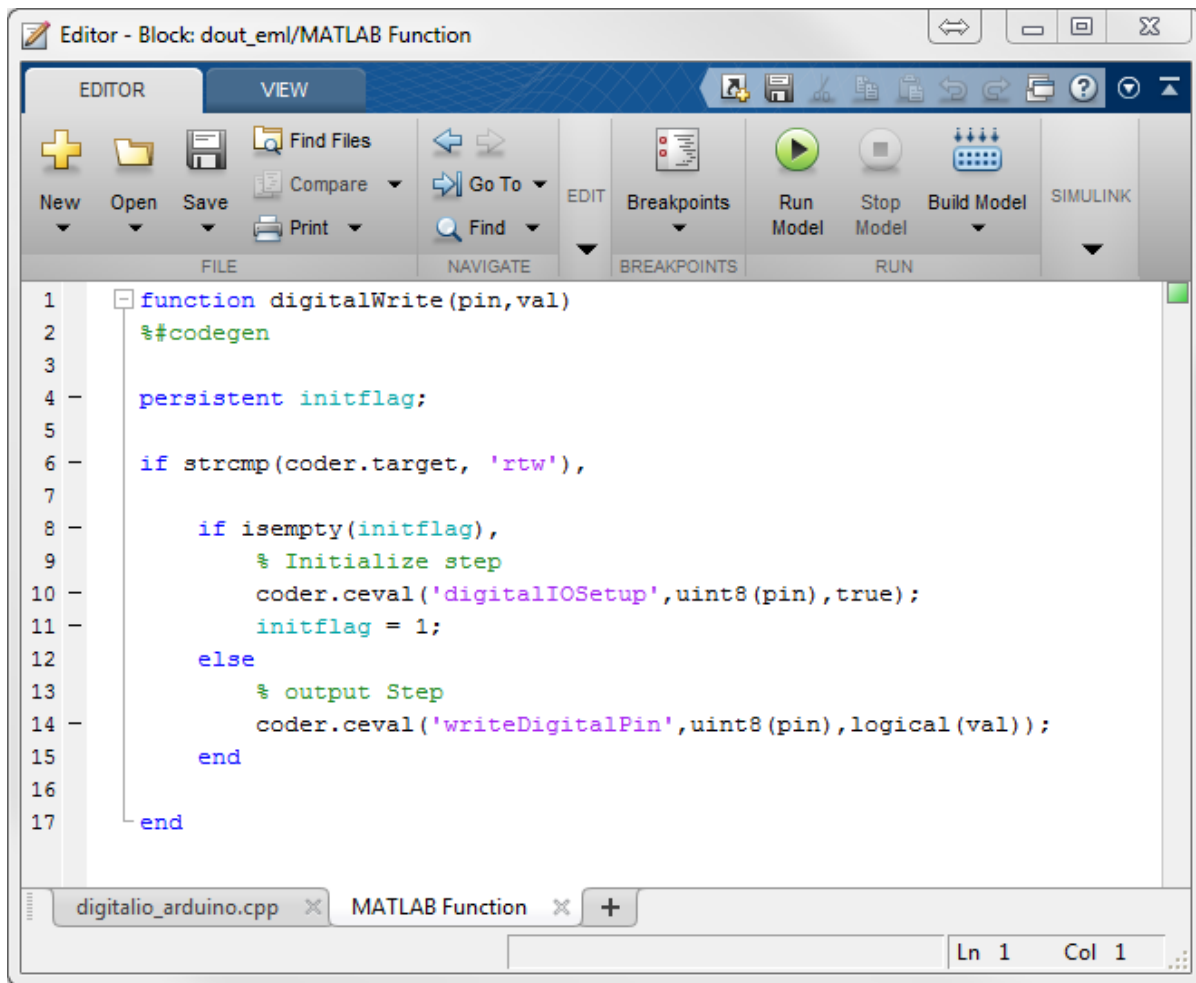
**Figure 24: MATLAB Function input, outputs, and parameters**

Note that when defining both the input and parameter variables, we have the choice of specifying the data type (e.g. `uint8`, `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 `val` variables in the MATLAB code which we can just type in the editor (Figure 23).



**Figure 25: 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 25.

The function `digitalWrite` takes in the parameter (`pin`) and the input (`val`) 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 25) the bulk of the function is skipped and nothing is done. This is what is typically 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`”, so in this case the bulk of the function is executed.

Since `initflag` is empty the first time this function is executed on the target (see line 8) the C **initialization** function `digitalIOSetup(uint8(pin), true)` is called by the `coder.ceval` instruction (line 10) and then `initflag` is then set to 1 (line 11).

From that point on, (since `initflag` remains equal to 1) every successive time the function is executed on the target, the C **output** function `writeDigitalPin(uint8(pin), logical(val))` is instead called by the `coder.ceval` instruction (line 14).

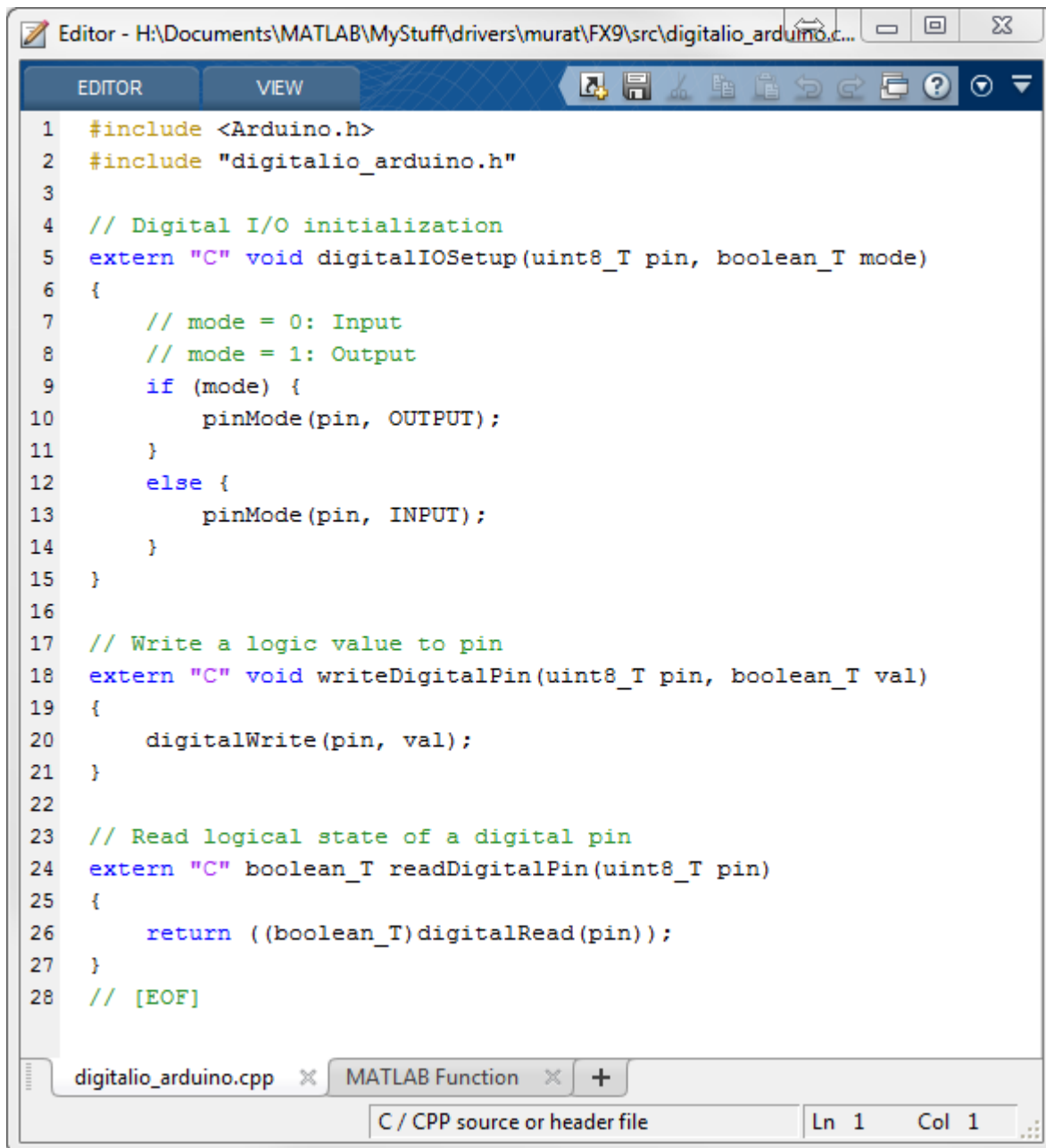
Note that this behavior is perfectly equivalent to the one described for the S-Function Builder block in pages 16-19 with the small difference that, in the S-Function Builder case, the conditional compilation instructions checking on the compilation target must be repeated in both calls (because an S-function must have two separate calls for output and discrete update). On the other hand, the LCT handles the simulation behavior separately, (via the use of a dummy function name that is commented before the MEX file is generated, see pages 32-33). This is actually a slight advantage of the LCT approach vs. the other methods, since with the LCT, the author does not need to explicitly write code (either in MATLAB or C) to manage the two execution modes separately.

## 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 26). This code (found in the file “`digitalio_arduino.cpp`”), handles both the digital input and output for an Arduino board, and consists of a header section where libraries can be included and global variables can be defined, followed by an initialization function and two output functions respectively.

The initialization C++ function `digitalIOSetup` takes `pin` and `mode` as input variables, and uses `pinMode(pin, OUTPUT)` (if `mode` is `TRUE`) or

`pinMode(pin, INPUT)` (if `mode` is `FALSE`) to respectively initialize to `OUTPUT` or `INPUT` a certain pin. The value of the `pin` parameter 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 C++ initialization function. The `mode` parameter is created directly in the MATLAB function (with a value of `TRUE` or `FALSE`, according to the need of the function) and passed down to the C++ initialization function.



```
1  #include <Arduino.h>
2  #include "digitalio_arduino.h"
3
4  // Digital I/O initialization
5  extern "C" void digitalIOSetup(uint8_T pin, boolean_T mode)
6  {
7      // mode = 0: Input
8      // mode = 1: Output
9      if (mode) {
10         pinMode(pin, OUTPUT);
11     }
12     else {
13         pinMode(pin, INPUT);
14     }
15 }
16
17 // Write a logic value to pin
18 extern "C" void writeDigitalPin(uint8_T pin, boolean_T val)
19 {
20     digitalWrite(pin, val);
21 }
22
23 // Read logical state of a digital pin
24 extern "C" boolean_T readDigitalPin(uint8_T pin)
25 {
26     return ((boolean_T) digitalRead(pin));
27 }
28 // [EOF]
```

The screenshot shows a MATLAB Editor window with the title bar "Editor - H:\Documents\MATLAB\MyStuff\drivers\murat\FX9\src\digitalio\_arduino.c...". The window contains a C++ source file named `digitalio_arduino.cpp`. The code defines two external C functions: `digitalIOSetup` for pin mode initialization and `writeDigitalPin` for writing logic values. It also includes a `readDigitalPin` function. The code uses `Arduino.h` and a custom header `digitalio_arduino.h`. The `pinMode` function is used to set pins to `OUTPUT` or `INPUT` mode based on the `mode` parameter. The `digitalWrite` and `digitalRead` functions are used for writing and reading logic values from the pins. The code is formatted with line numbers 1 through 28. The status bar at the bottom indicates "C / CPP source or header file" and "Ln 1 Col 1".

**Figure 26: C driver code containing initialization and output calls**

Similarly, in the output function `writeDigitalPin`, the Arduino specific instruction `digitalWrite(pin, val)` is used to write the content of the variable `val` (coming from the block input) to the pin specified by the parameter `pin`.

Note that since the Arduino functions `pinMode` and `digitalWrite` accept variables of type `uint8_T` and `boolean_T` for the variables `pin` and `val`, the MATLAB function also takes care of casting these variables into the appropriate type (lines 10 and 14) before passing them to the underlying C++ file.

The above code snippet can be found in the file “`digitalio_arduino.cpp`”. Also see pages 54 and 55 in this guide for more explanation on the included file “`digitalio_arduino.h`”, and why is a good idea to use it.

## 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 25).

As already mentioned in pages 24-25, this can be achieved using the `CustomSource` model parameter. Specifically, executing the following instruction from the MATLAB command line:

```
>> set_param('dout_eml', 'CustomSource',  
    fullfile('..', 'src', 'digitalio_arduino.cpp')) ;
```

will set the `CustomSource` parameter for the Simulink model “`dout_eml`” to the string “`fullfile('..', 'src', 'digitalio_arduino.cpp')`”.

The MATLAB instruction “`fullfile`” is used to build a platform-dependent full file specification from the folders (in this case “`..\src`”) and file name (in this case “`digitalio_arduino.cpp`”) specified. The result is a string pointing to the source file in Figure 26, which resides in the “`src`” subfolder of the Drivers Guide main installation folder.

Therefore, if the file “`digitalio_arduino.cpp`” 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 `“fullfile('..','src','digitalio_arduino.cpp’)”`, 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).

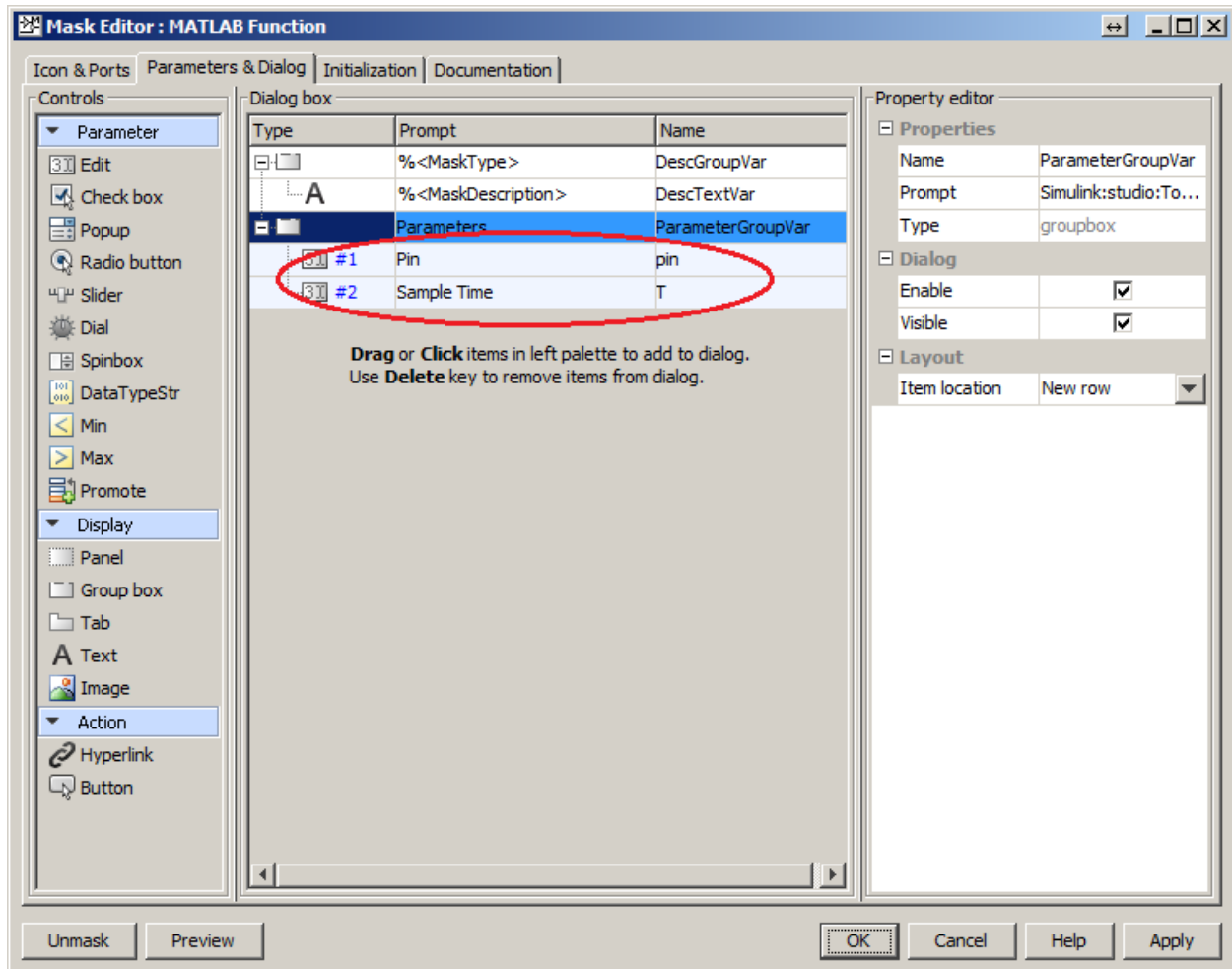
This (the necessary MATLAB command to be executed) can be written in the mask documentation (or even in a text box in the Simulink model), so that users are reminded to properly set the `CustomSource` parameter.

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 (and should) be masked following pretty much the same procedure described in the previous pages (27-29) for the S-Function Builder block.

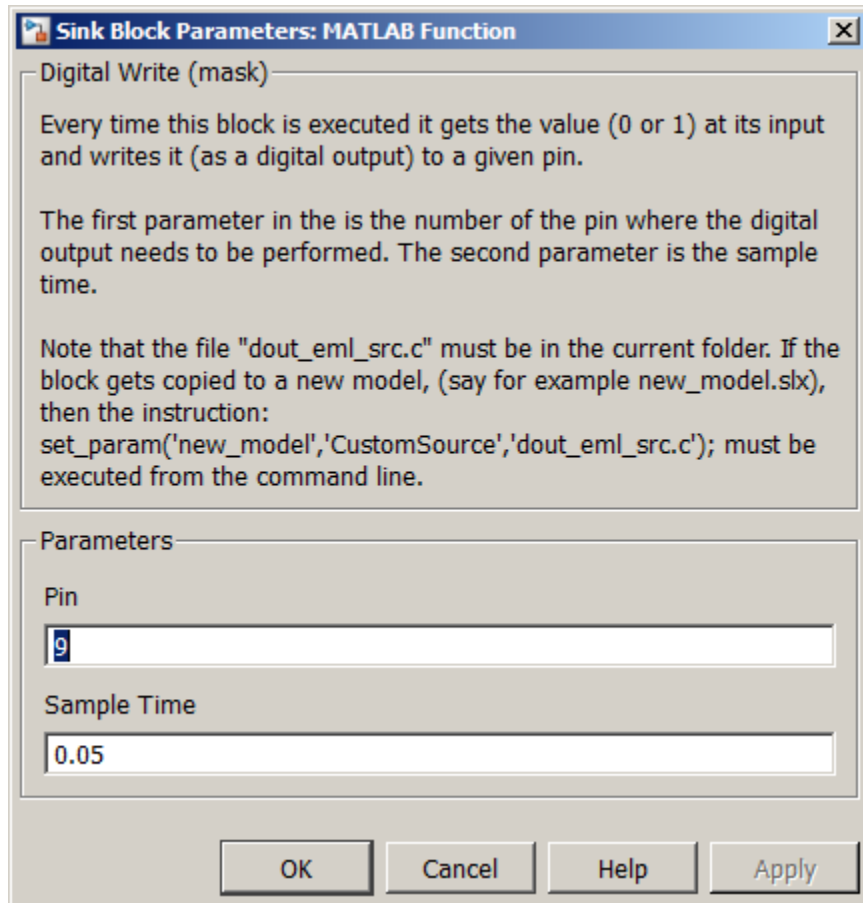
As shown in Figure 27, both a Pin (`pin`) and a Sample Time (`T`) parameter can be added to the mask so that their values can be selected by the user and passed to the underlying MATLAB Function block. Some pertinent documentation can be also added in the Documentation pane.



**Figure 27: Masking the MATLAB Function block**

Differently from the case of the S-Function Builder block, the user can access the mask (Figure 28) just by double clicking on the block.





**Figure 28: 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 (unmasked) S-Function Builder approaches, 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.

If the S-Function Builder block is masked, the S-function needs to be rebuilt only when the Sample Time is changed (similarly, to what happens for the LCT approach).

## The MATLAB System Block Approach

MATLAB System block is an interface to System Objects. System Objects are MATLAB classes inheriting from `matlab.System` and have a set of API's that make them well suited for representing device drivers. A System Object has methods called `setup`, `step` and `release` that can be used to define initialization, output and termination code for a device driver. For example, if you are developing a Digital Read function for an Arduino board, you configure a digital pin as input at model initialization, read the pin state in model execution or `step` and release resources used by the Digital Read in model termination.

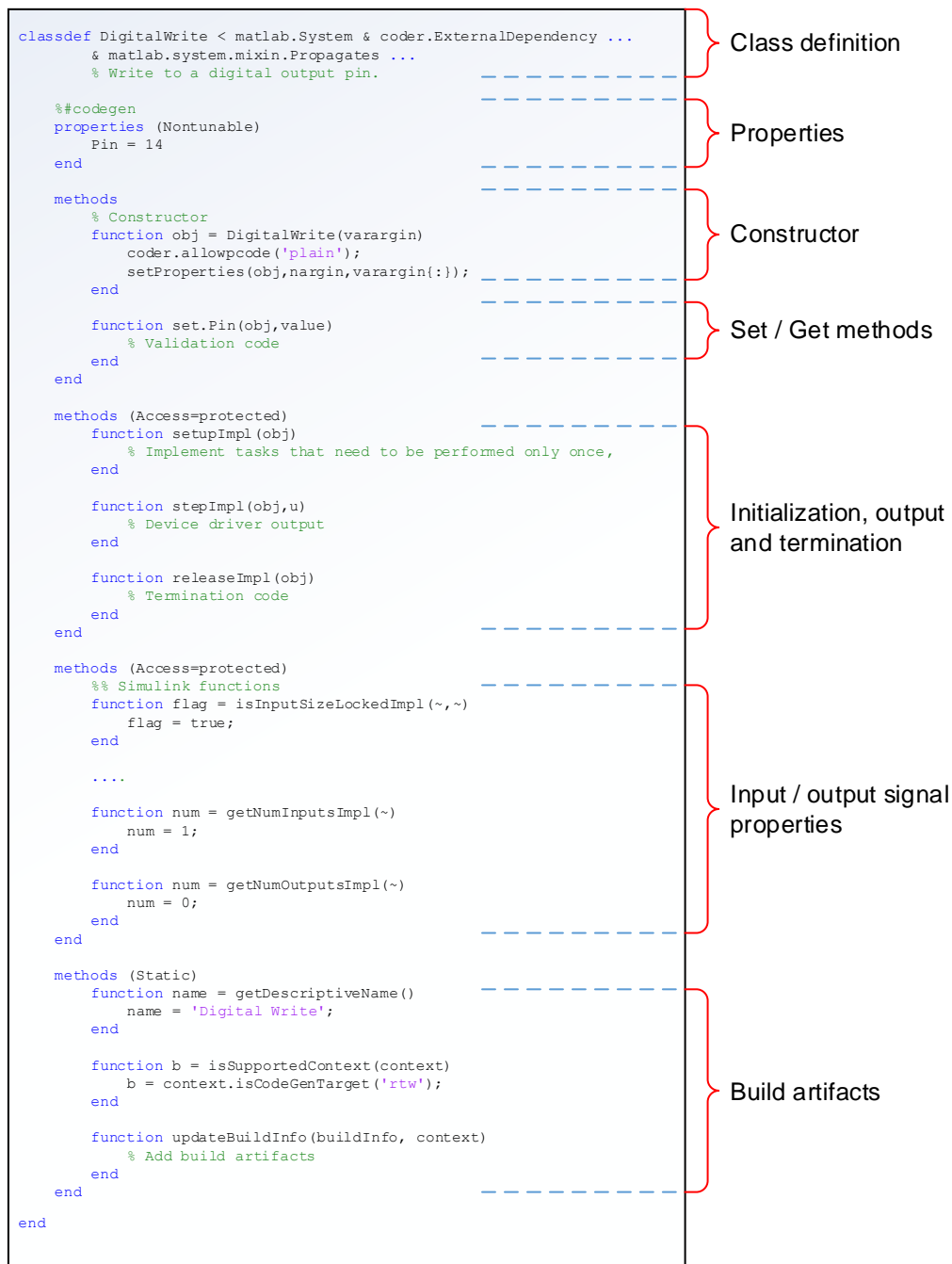
In addition to modeling what happens during code generation, System Objects also allow you to define simulation behavior for a device driver using familiar MATLAB language. As also explained in the previous sections of this guide, in simulation mode, the entire model runs on the host computer, and you do not have access to hardware resources. Consequently, the input or output of a device driver block has to be simulated. Continuing with the Digital Read example the block could, for example, simulate the outputs by reading them from a file. This allows you to create test vectors that provide meaningful simulation.

System Objects also provide services for adding build artifacts (i.e. source files, include paths, shared libraries, pre-processor defines, etc.) to Simulink generated code, defining input or output port properties of a block and automatically generating a Simulink block mask.

**Note** The System Object approach described here is only compatible with MATLAB releases R2015a or higher.

## Anatomy of a System Object -based Device Driver

A device driver System Object contains logically grouped sections of code as shown in the following diagram:



**Figure 29: A typical System Object representing a device driver.**

Here is a brief overview of each code section:

- **Class definition:** At the top of the System Object code, you define the name of your System Object and the classes it inherits from. All System Objects must inherit from `matlab.System`. In addition, device driver System Objects inherit from `coder.ExternalDependency` that provides API's to add build artifacts to generated code. The `matlab.system.mixin.Propagates` class provides API's to define the output size, data type, and complexity of a System Object. You can inherit from other classes, e.g. `matlab.system.mixin.CustomIcon`, which lets you specify the name and icon used by a MATLAB System Object block.
- **Constructor:** In our examples, we use a standard class constructor.
- **Set / Get methods:** Set and get methods allow you to execute code before assigning a property or perform actions before returning a property value. You typically use set methods for properties that require range, type and dimension checking. A concrete example of using a set method is to check that the pin number for a Digital Write block is within the range defined by hardware.
- **Initialization, output and termination:** the `-impl` methods in this section allow you to define what happens at initialization, output and termination. You use `setupImpl` to initialize hardware peripheral, `stepImpl` to read from or write to the hardware peripheral and `releaseImpl` to release hardware resources used. These three methods are the backbone of defining the behavior of a device driver block.
- **Input / output signal properties:** In this code section, you define the number of inputs or outputs of a block and the data types and sizes. For example, for a source block you set the number of inputs to zero in `getNumInputsImpl` method and set the number of outputs in `getNumOutputsImpl` method.
- **Build artifacts:** In this section of the code, you define the source files, include paths, shared libraries, library search paths and pre-processor defines required to compile the device driver code. The `getDescriptiveName` method allows you to define an identification string to the System Object for use by code generation engine to report errors. The `isSupportedContext` lets you specify the code generation context. We only care about real-time workshop (rtw) code generation context so this function will always specify 'rtw'. The `updateBuildInfo` method allows you to specify source and header files, include paths, libraries and defines required to build the System Object.

Rather than a clinical explanation of each method, we will dive directly into authoring a System Object implementing Digital Write functionality for Arduino hardware. We will start from a standard *Sink* template System Object and customize it to write to a digital output pin on Arduino hardware.

## Creating a Digital Write Block Using System Objects

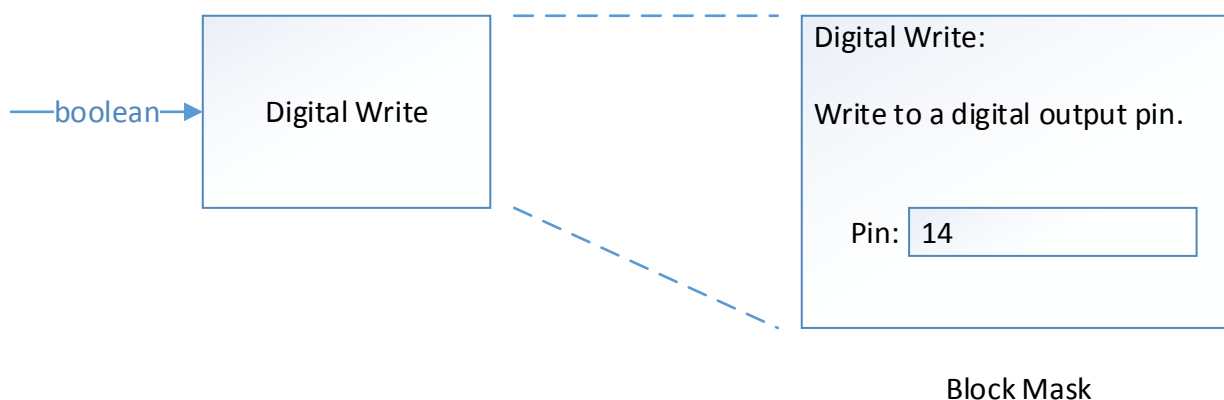
Before starting with the implementation, perform a rough functional design to determine how the block should look like. In particular, try to determine the following:

- Number of inputs / outputs of the block
- Parameters to be exposed on the block mask
- Range, data type, data size checking needed for the parameters
- Input or output port properties including data type, size and complexity

In this particular example, we will develop a Digital Write block according to the following functional design:

- The block will have a single input and no output
- A single parameter called **Pin** will be exposed on the block mask. This parameter will determine which digital output pin on the Arduino hardware is used
- The **Pin** will be a numeric scalar integer. It will assume values from 0 to 53
- The input port will accept scalar boolean signals
- The block will write the input port value to the digital output pin

According to this functional design, the block should look like the following:



**Figure 30: Functional design for a Digital Write block.**

## Developing the Wrapper Functions Implementing Digital Write

In most cases, you need to write a wrapper function around the API's provided by the hardware vendor to integrate device driver code as a Simulink block. Let's develop the C++ code required to implement the digital write functionality:

```
#include <Arduino.h>
#include "digitalio_arduino.h"

extern "C" void digitalIOSetup(uint8_T pin, boolean_T mode)
{
    // mode = 0: Input
    // mode = 1: Output
    if (mode) {
        pinMode(pin, OUTPUT);
    }
    else {
        pinMode(pin, INPUT);
    }
}

// Write a logic value to pin
extern "C" void writeDigitalPin(uint8_T pin, boolean_T val)
{
    digitalWrite(pin, val);
}
```

Let's assume that we saved this code to `digitalio_arduino.cpp` file. We also need a `digitalio_arduino.h` header file defining the C prototypes of the functions implemented in the CPP file:

```
#ifndef _DIGITALIO_ARDUINO_H_
#define _DIGITALIO_ARDUINO_H_
#include "rtwtypes.h"

#ifdef __cplusplus
extern "C" {
#endif

void digitalIOSetup(uint8_T pin, boolean_T mode);
void writeDigitalPin(uint8_T pin, boolean_T val);
boolean_T readDigitalPin(uint8_T pin);

#ifdef __cplusplus
}
#endif
#endif // _DIGITALIO_ARDUINO_H_
```

The Simulink Support Package for Arduino Hardware uses a C-compiler called `avr-gcc`. The `digitalWrite_arduino.cpp` contains C++ functions. In order to compile and link these C++ functions with a C-compiler, we need `extern "C"` identifier in each function declaration to tell the compiler not to mangle function names so that they can be used with the C linker. In the `digitalio_arduino.cpp` function we include `<Arduino.h>` file that defines the `pinMode` and `digitalWrite` functions. Note that we are using Simulink data types for `pin` and `val` variables. For this reason, we include “`rtwtypes.h`” file in `digitalio_arduino.h`. You must include this file whenever you reference to Simulink data types. Since `pin` is a number between 0 and 53 we use the `uint8_T` data type to represent this variable. The `in` variable is the value to be written to the digital output pin and is represented by `boolean_T` data type.

## Create a System Object

Create a new System Object called `DigitalWrite` starting from the `Sink.m` System Object template:

```
classdef DigitalWrite < matlab.System ...
    & coder.ExternalDependency ...
    & matlab.system.mixin.Propagates ...
    & matlab.system.mixin.CustomIcon
%
% Set the logical state of a digital output pin.
%
%#codegen
%#ok<*EMCA>

...

methods
    % Constructor
    function obj = DigitalWrite(varargin)
        coder.allowpcode('plain');
        setProperties(obj,nargin,varargin{:});
    end
    ...
end
...
end
```

Name the System Object DigitalWrite. Modify the highlighted constructor name to match the System Object name. After modifying template as indicated above, save it to a file named 'DigitalWrite.m'.

## Define Properties

In the functional design we determined that the block will have a single parameter called **Pin**. The **Pin** is a number indicating the hardware pin number in an Arduino board. Since the **Pin** is exposed on the block mask, we define it as a public property.

```
properties (Nontunable)
    % Public, non-tunable properties.
    Pin = 3
end
```

The property **Pin** is declared nontunable indicating that it does not change once the simulation starts.

There are many different types of Arduino boards that have different number of digital I/O pins. Arduino Uno has 14 pins that can be used as digital outputs while Arduino Mega has 54. You can add a range check to the value assigned to the property **Pin** by adding set method for **Pin**. Take a look at the DigitalWrite\_arduino.m file to see how this can be done.

## Define Inputs

In the functional design stage of the Digital Write block, we determined that the input to the block is a Boolean representing HIGH and LOW logic states of a digital output pin. We also determined that the block has a single numeric scalar input and no outputs. Putting these all together, here is how we implement the *Input / output signal properties* code section:

```
methods (Access=protected)
    ...
    function num = getNumInputsImpl(~)
        num = 1;
    end

    function num = getNumOutputsImpl(~)
        num = 0;
    end
end
```



```

function validateInputsImpl(~, u)
    if isempty(coder.target)
        % Run this always in Simulation
        validateattributes(u,{'logical','double'},
            {'scalar','binary'},'', 'u');
    end
end
end
end

```

In `getNumInputsImpl` and `getNumOutputsImpl` methods, we declare that the System Object has one input and no outputs.

We use the `validateInputsImpl` method to ensure that the input signal is a 'logical' or 'double' representing a binary (i.e. boolean) scalar.

## Define Initialization, Output and Termination Functions

In Section 3.2 we created the C wrapper functions implementing the initialization and output of digital write operation. We use `setupImpl` and `stepImpl` to hook these C-functions to the System Object. The initialization of a digital pin as output needs to be done only once at model initialization. Hence we call `digitalIOSetup` function in `setupImpl`. To update the logic state of the digital output pin, a call to `writeDigitalPin` is made from `stepImpl` method. There is nothing to be done at termination. As a result, the *Initialization, output and termination* code section of the DigitalWrite System Object looks like the following:

```

methods (Access=protected)
    function setupImpl(obj)
        if coder.target('Rtw')
            coder.cinclude('digitalio_arduino.h');
            coder.ceval('digitalIOSetup', obj.Pin, 1);
        end
    end
end

function stepImpl(obj,u)
    if coder.target('Rtw')
        coder.ceval('writeDigitalPin', obj.Pin, u);
    end
end

function releaseImpl(obj) %#ok<MANU>
end

```

```
end
```

To execute calls to C wrapper functions created earlier, `coder.ceval` is used. The C-function calls to the wrapper functions are only executed when generating code. In normal simulation mode, this block does nothing. To incorporate simulation behavior, you place an else condition to `if coder.target('Rtw')` code block and place the simulation code on the else branch. For example, you may want to print the input value on the MATLAB command line using a `printf` statement.

Finally, the `coder.cinclude` call in the `setupImpl` method includes the header file containing the function prototypes.

## Define Build Artifacts

The `coder.ExternalDependency` class provides an interface for instructing the code generation engine to include external C code and header files while generating code and building the model. Since we created a `*.cpp` file and a `*.h` file, we need to tell the code generation engine to add these files:

```
methods (Static)
    function name = getDescriptiveName()
        name = 'Digital Write';
    end

    function b = isSupportedContext(context)
        b = context.isCodeGenTarget('rtw');
    end

    function updateBuildInfo(buildInfo, context)
        if context.isCodeGenTarget('rtw')
            % Update buildInfo
            rootDir = fullfile(fileparts(mfilename('fullpath')), 'src');
            buildInfo.addIncludePaths(rootDir);
            buildInfo.addIncludeFiles('digitalio_arduino.h');
            buildInfo.addSourceFiles('digitalio_arduino.cpp', rootDir);
        end
    end
end
```

The `getDescriptiveName` and `isSupportedContext` methods were explained earlier. In most System Objects you write, you will use the standard template shown above for these methods. The `updateBuildInfo` method is where you define the source and header files needed to build the System Object. In

addition you need to specify where to find these files. In this example, we chose to place the source files in a directory called *src* under the main Drivers Guide folder.

The `buildInfo` API also lets you specify shared libraries or global defines used for compilation should you need it. You can see an example of linking a shared library in `DigitalWrite_raspi.m` System Object.

Note that this is a specific advantage with respect to other methods of creating device drivers, where the inclusion of shared libraries had often to be defined using the `CustomSource` or `CustomLibrary` model-wide parameters.

## Testing the System Object

The `DigitalWrite` System Object is created to be used with a MATLAB System block. Before we import the System Object to Simulink we do a sanity check on the MATLAB command line:

```
>> dw = DigitalWrite('Pin', 5)
dw =
  System: DigitalWrite
  Properties:
    Pin: 5
>> dw.step(1)
>> clear dw
```

The calls above exercises the `setupImpl`, `stepImpl` and `releaseImpl` methods. You can also test the `set` method for `Pin` by assigning an invalid value. It is a good idea to exercise the System Object as much as possible on the MATLAB command line before moving on to creating a Simulink block.

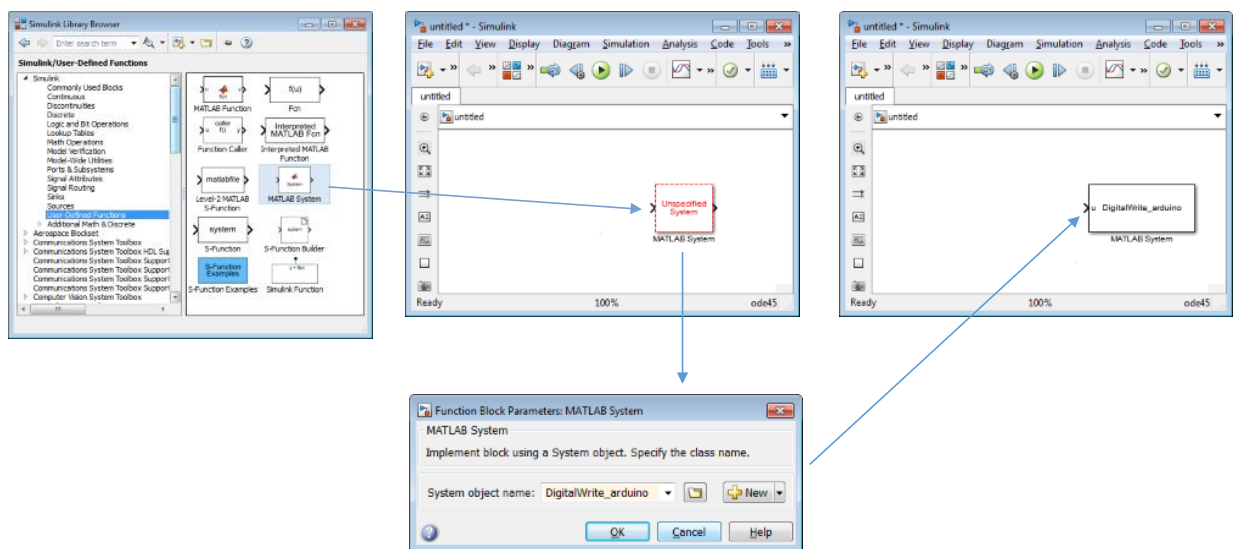
The possibility of calling the methods from the MATLAB command line is another advantage of this approach, with respect to the previous ones.

## Using System Objects in Simulink with the MATLAB System Block

To bring the DigitalWrite\_arduino System Object to Simulink, follow the steps below:

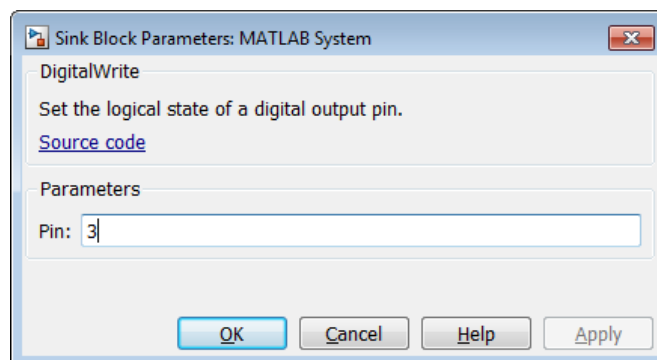
1. Drag and drop a MATLAB System block to a Simulink model
2. Open the MATLAB System block dialog and enter the name of the System Object.
3. Click OK on the MATLAB System block.

A pictorial representation of the steps above is shown in Figure 31.



**Figure 31: Steps for importing a System Object to Simulink.**

Double click on the MATLAB System block after completing the steps above to see the block mask:

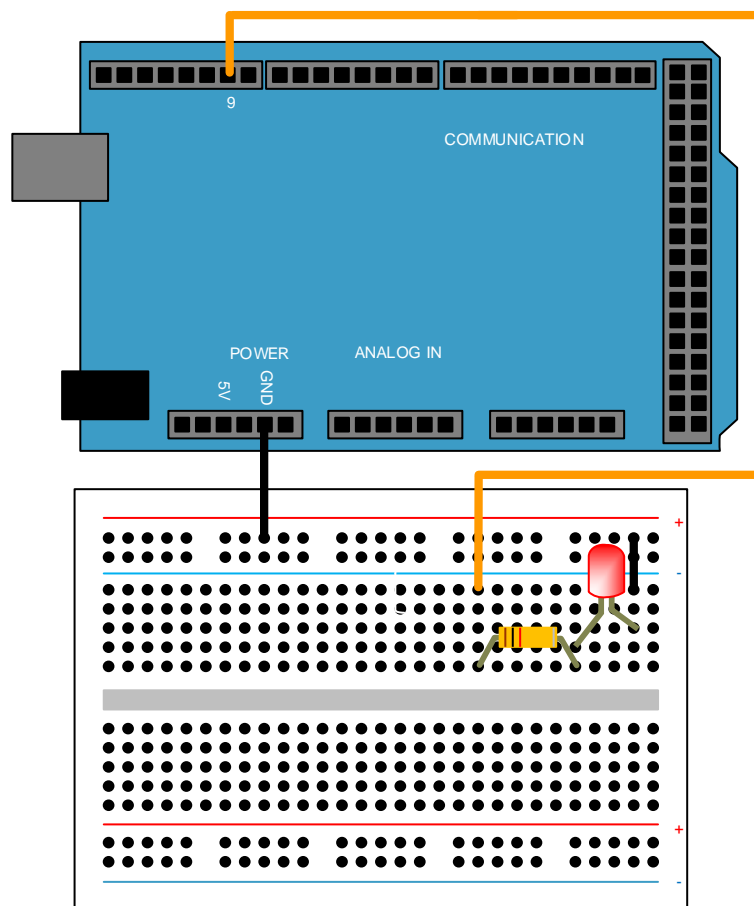


**Figure 32: The block mask for DigitalWrite System Object.**

The public property **Pin** defined in the DigitalWrite System Object is exposed on the block mask as an editable parameter.

## Run on Target Hardware

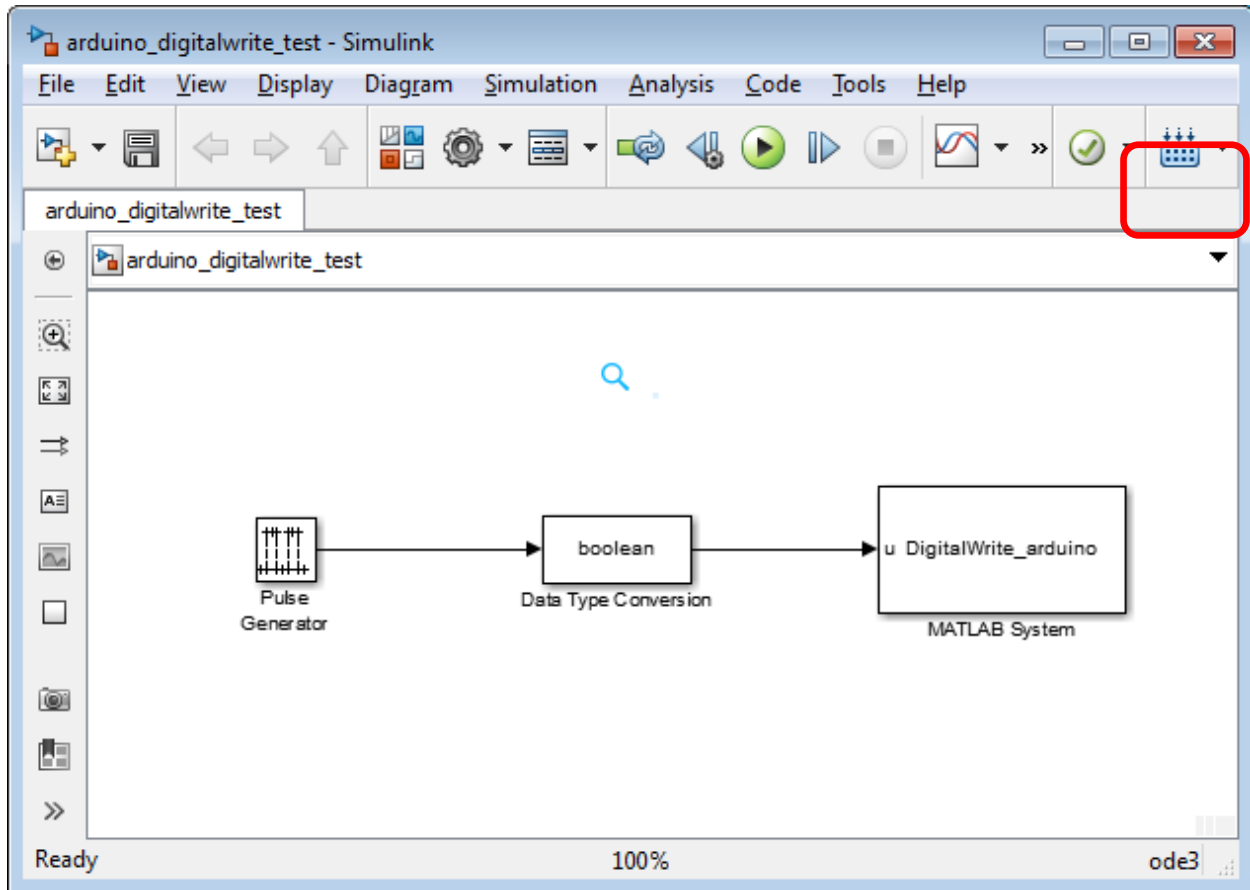
The next step is to test the DigitalWrite block on Arduino hardware. We will blink a red LED connected to a digital output pin with a 1 KOhm resistor in series as seen in Figure 30. The connections are illustrated for an Arduino Mega board though you can use any Arduino compatible hardware with a similar circuit.



**Figure 33: Hardware connections for testing Digital Write block. This example is for Arduino Mega.**

The Simulink model for blinking the LED employs a Pulse Generator connected to the input port of the DigitalWrite block as seen in Figure 31. DigitalWrite block only accepts boolean values at its input port. A Data Type Conversion block is

inserted between the Pulse Generator and the DigitalWrite block to convert the output of the Pulse Generator to a boolean value expected by the DigitalWrite System Object.



**Figure 34: Simulink model created to blink an LED connected to a digital output pin.**

The model shown in Figure 34 is included in the Drivers Guide for convenience. Open the 'digitalwrite\_arduino\_test' model. Follow the steps below to run the Simulink model on the Arduino hardware:

1. Double click on the 'MATLAB System' block mask and set the **Pin** to the output pin you attached the LED
2. Click on Tools > Run on Target Hardware > Options. Set the 'Target Hardware' to the Arduino compatible board you are using
3. Click on the 'Deploy to Hardware' button indicated with a red rectangle in Figure 6 to build and run the model on your hardware

After executing the steps 1 – 3 above, you should see the LED blinking with a period of 1 second.

## Additional System Object Examples

We have also included the following System Object examples for Arduino and Raspberry Pi hardware:

- Digital Read
- Digital Write
- Encoder
- Analog Output

Digital Read and Encoder are developed starting from the *Source.m* System Object template while Digital Write and Analog Output are developed using the *Sink.m* System Object template.

The Raspberry Pi ‘Digital Read’, ‘Digital Write’ and ‘Encoder’ System Objects (tagged with *\*\_raspi.m*) utilize [Wiring Pi](#) library to read from and write to the GPIO pins. These blocks link in the wiringPi shared library which comes pre-installed in the MathWorks Raspbian Linux image. The Analog Output block uses an add-on software called [ServoBlaster](#), which provides an interface for generating PWM waveforms on the GPIO pins. The instructions for installing and using ServoBlaster is included in the help for AnalogOutput\_raspi System Object (type in ‘help AnalogOutput\_raspi’ to see the System Object help) and also available from [this link](#).