Use Authorization

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission to download and/or print my thesis for scholarly purposes may be granted by the Dean of the Graduate School, Dean of my academic division, or by the University Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

DEVELOPMENT OF VOICE CONTROL APPLICABLE TO PROSTHETIC ARMS

By

DURGASAMANTH PIDIKITI

A thesis submitted in partial fulfillment of the

Requirements for the degree of

MASTER OF SCIENCE

IN

MEASUREMENT AND CONTROL ENGINEERING

IDAHO STATE UNIVERSITY

DECEMBER 2014

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of DURGASAMANTH PIDIKITI find it satisfactory and recommend that it be accepted.

_____

Dr. Marco P. Schoen, Ph.D, Major Advisor

_____

Dr. Alba Perez, **Ph.D, Member**

_____

Dr. Alex Urfer, PT, Ph.D, GFR

ACKNOWLEDGEMENTS

I would like to extend a special thanks to my major advisor, Dr. Marco P.Schoen for his assistance and guidance in completing this thesis. Without his knowledge, motivation, and support this research would not have been possible.

I would also like to thank my family members. Without their support and unlimited patience this research would not have been possible.

TABLE OF CONTENTS

NOMENCLATURE

EMG	Electromyography
GUI	Graphical User Interface
EEPROM	Electrically Erasable Programmable Read Only Memory
LCD	Liquid Crystal Display
HDL	Hardware Description Language
DSP	Digital Signal Processing
SD	Secure Digital

LIST OF FIGURES

ABSTRACT

This thesis presents a new approach of controlling a prosthetic arm using voice commands. This voice controlled technique can also be applied to other applications such as security systems, electronic device control and so on. The goal of this prosthetic project is to reduce the hardware and software dependencies in controlling the prosthetic arm. The user of the arm first needs to record the set of voice commands in order for the system to respond. The system responds only to the actual user as a safety mechanism. The code is integrated in to the hardware board once the recording is done. The robotic arm is then able to follow the specific instructions. The user can add new commands to the existing set. As the size of the board and data processing is very small the amount of energy consumed will be very small.

Simulink™ blocks compatible with Arduino™ hardware are created as part of this thesis work. These blocks will help develop Simulink™ designs without prior knowledge of electronic components. The main contribution of this project is the creation of Simulink™ blocks and the development of voice controlled code for a prosthetic arm compatible with Arduino™ hardware.

CHAPTER 1.0

INTRODUCTION

Voice-based control applications are gaining popularity with every passing year as they reduce the human effort. They also reduce the time for processing, analyzing, computing and executing various functions. The voice-based control is being used in gadgets such as smartphones [1], speech to text applications [2] and so on.

A great deal of research work is [3a][3b][3c] going on to improve the prosthetic arms which would replicate the movement of an actual hand. One of the most important tasks that have to be performed to replicate the natural arm movement is the processing of EMG signals resulting from muscle activities. At present, research [4] is being done on this key aspect of data processing. Here data processing means collection of control pulse originated in the brain and generated by the muscles responsible for the human motion. Collection of these EMG signals using electrodes is a difficult task. The location of EMG signals generation from muscles varies from person to person. These EMG signals serve as input control data for the prosthetic arm control. The signal strength will be low and the signal to noise ratio will be low for the EMG signals captured using surface electrodes [5]. Therefore, the signal strength has to be improved and noise has to be reduced before feeding it to control unit of prosthetic arm.

The requirement of signal processing will lead to extra circuitry [5]. Added circuitry leads to power consumption. Personalized therapy or training is required before starting to use the device. Based on the above facts the author has decided to proceed

with voice controlled prosthetic arm. The types of voice control systems available for use and advantages of each system are explained below.

Voice control can be broadly classified in to two types as limited word based and unlimited word based or sentence handlers. Limited word control based systems can handle a few words ranging from processing tens of words to a maximum of a few hundred words. Unlimited word based or sentence handlers typically can handle entire sentences. Compared to word handling software, the sentence handling software requires more complex algorithms to collect given commands, speech to text conversion, search and convert text to speech. The entire process requires an internet connection [6].

The Internet connection is not required for limited word handler [7]. A minimal set of hardware and software packages are required for implementing the limited word based controller. Wrist movement control requires only few commands. All of the above points are the major factors considered while implementing the limited word based controller.

The limited word based controller shown in Fig.24 is taken and first programmed with the words or users choice. The procedure for uploading commands and downloading them in to the Arduino™ board is explained in detail in Appendix A1.

The goals undertaken by the author of this work are explained in Section 1.2.

1.1 Problem Statement

Classical control mechanism utilized to control a prosthetic arm movement involves complex steps such as EMG signal capture, signal strength enhancement [5], signal processing, etc.

EMG signal capturing also involves detection of motor points on the human arm for placement of the electrodes. Surface electrodes are used to collect the EMG pulse generated from the muscles. Placement of the electrodes varies from person to person as each individual is unique. The signal strength of the EMG pulses captured will decrease as it passes through the muscles tissue and skin at the time it reaches electrodes [5]. Therefore the signal strength has to be increased or amplified [5]. The signal processing circuitry then processes the sensed EMG data and feeds it to the algorithm [8] which controls the prosthetic device movement.

The proposed method uses voice commands to control the prosthetic arm which eliminates the step of EMG signal collection from muscles. Therefore the requirement of the use of electrodes and signal processing circuitry is eliminated.

1.2 Thesis Goals

In the following, a list of specific thesis goals is presented:

- Develop Simulink™ blocks of basic electronic components which can be downloaded to an Arduino™ microcontroller board.
- Develop voice controlled code to control the movement of a prosthetic arm.
- Carry out voice control hardware simulations on electronic components.

CHAPTER 2

Simulink™, Arduino™ and how to acquire an Arduino™ support package for Simulink™ are explained in this chapter.

2. 1 Introduction To Simulink™

Simulink™, developed by MathWorks, is a data flow graphical programming language tool for modeling, simulating and analyzing multidomain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries [9]. Simulink™ tab in Matlab™window is shown in Fig.1. Simulink™ window is shown in Fig.2.

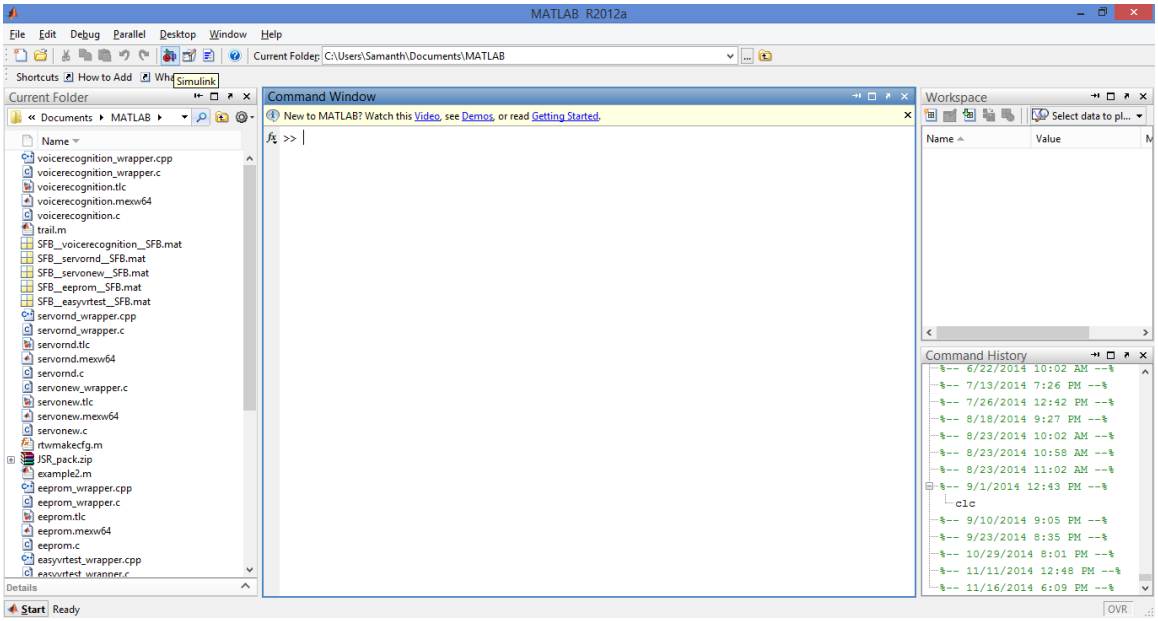


Fig.1 Opening Simulink™ from Matlab™ window.

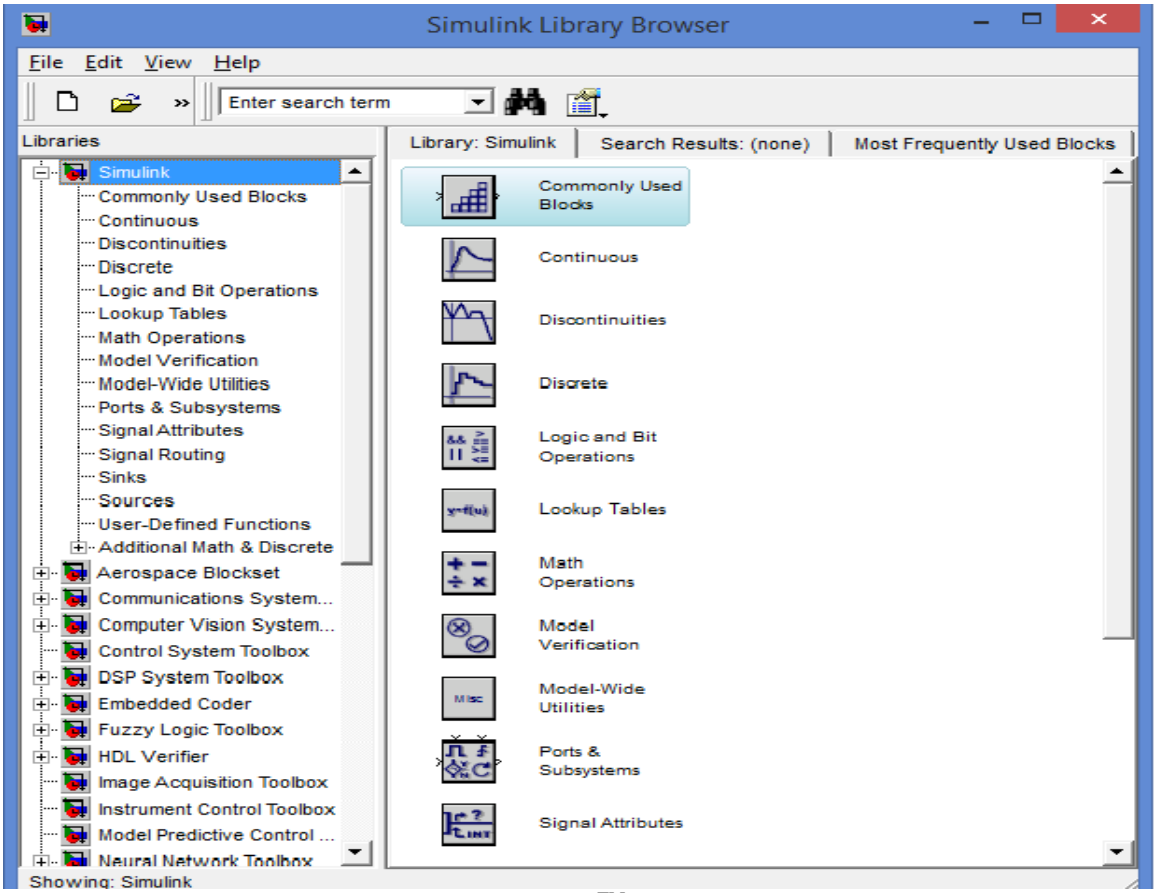


Fig.2 Simulink™ window.

A number of third party hardware and software products are compatible with Simulink™. Some of the examples of libraries available in Simulin™, are Simulink™, Aerospace, Communication system, Computer vision system, DSP system, Embedded coder, Fuzzy logic, HDL verifier, Image acquisition, Neural network, State flow, Vehicle network and so on, shown in Fig.3.

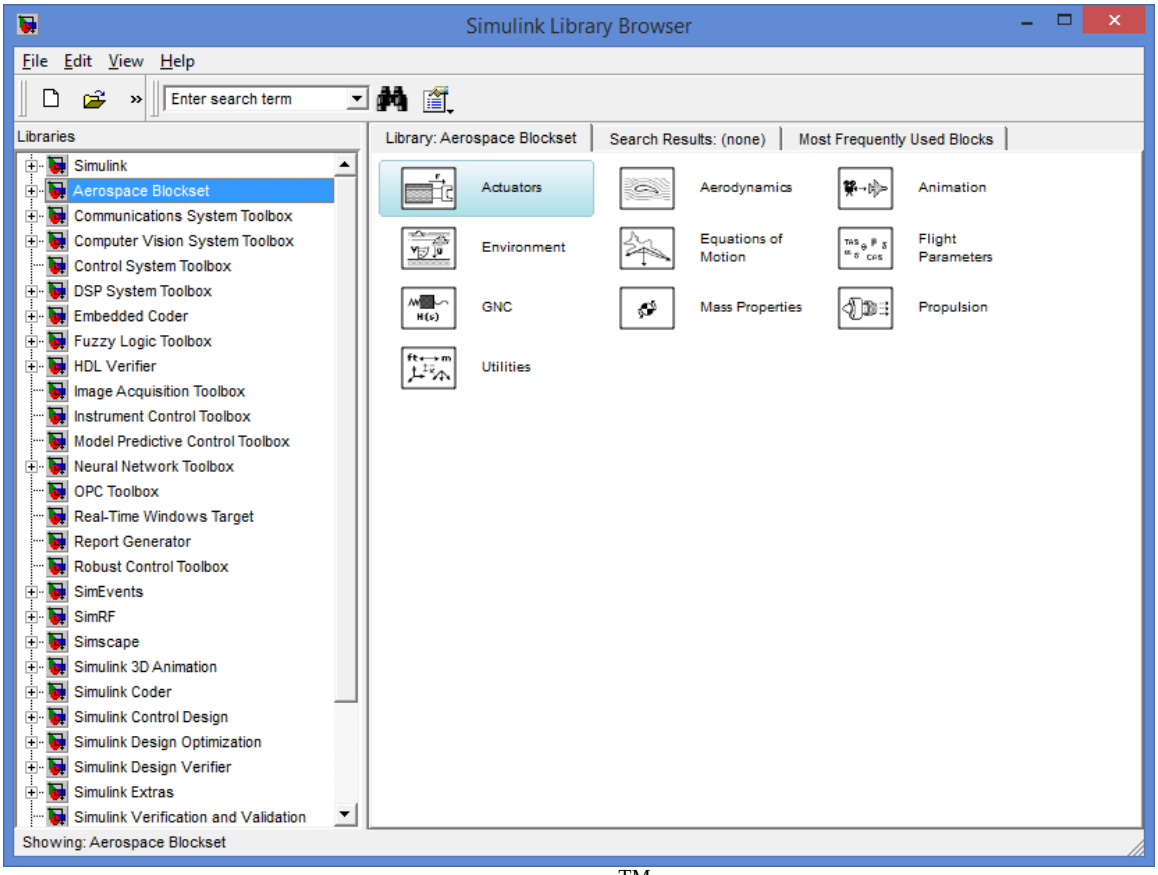


Fig.3 Libraries in Simulink™ window.

Each library consists of a number of basic blocks required for creating a Simulink™ design. The Communication System library consists of blocks shown in Fig.4. Creation of customized blocks is explained in detail in topic 3.

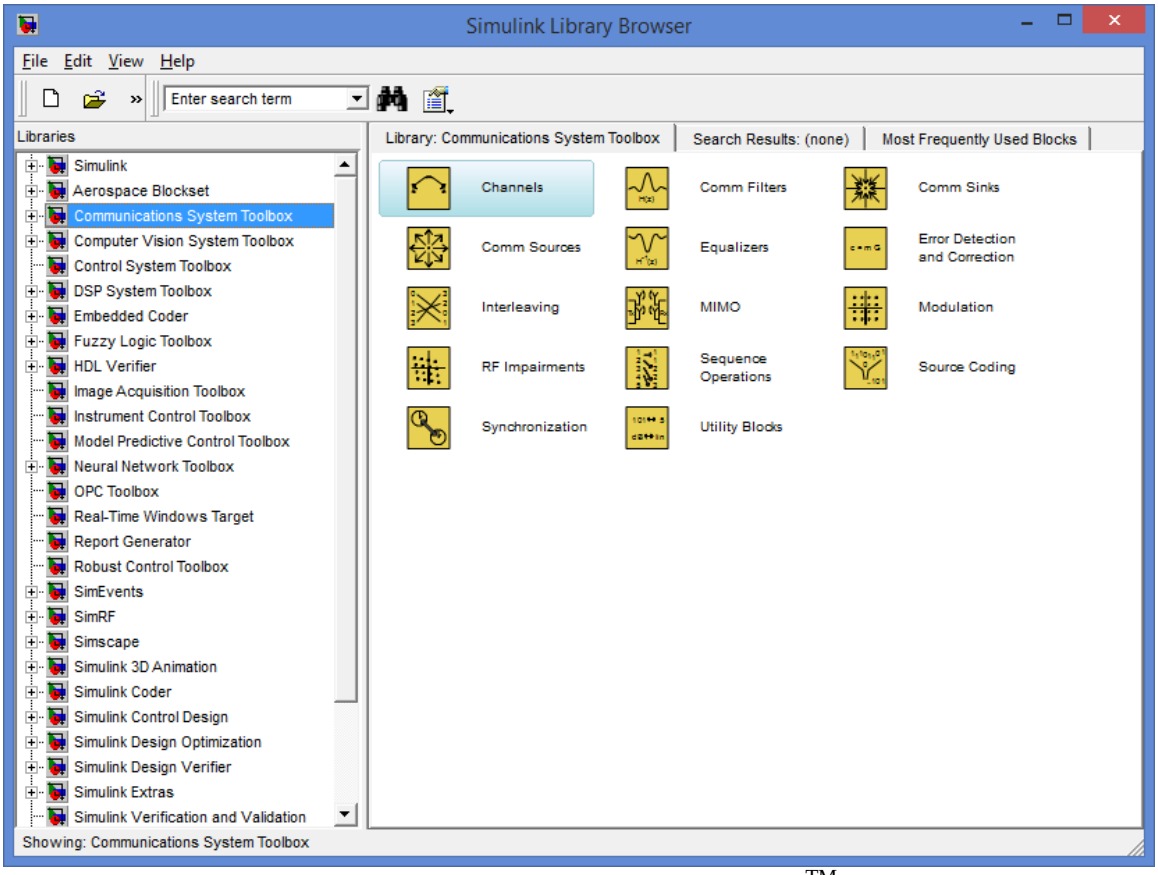


Fig.4 Blocks of Communication System Library in Simulink™ window.

2.2. Introduction To Arduino™

Arduino™ is an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board [10]. Arduino™ is an inexpensive, cross platform (i.e. operating system independent), simple programming environment, open source hardware and software. The open source software environment has led to the development of a pool of libraries by developers all over the world.

Arduino™ GUI is shown in Fig.5. The list of examples which are available in Arduino™ is shown in Fig.6. A basic example of analog read in Arduino™ programming language is shown in Fig.7. Arduino™ programming language consists of

two main functions namely setup() and loop(). The setup() is used for initialization of data. The loop() is used for data manipulation to achieve the desired output.

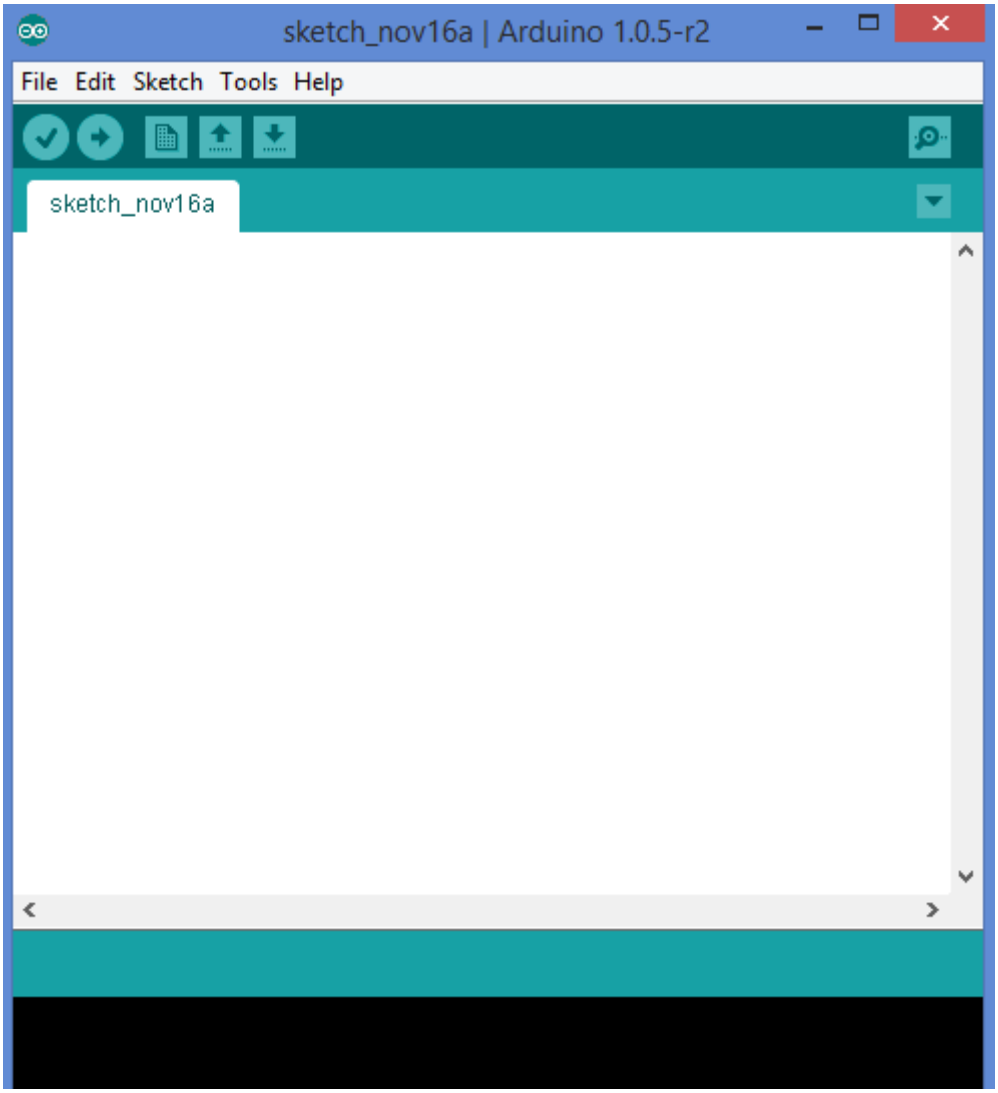


Fig.5 GUI of Arduino™ Software.

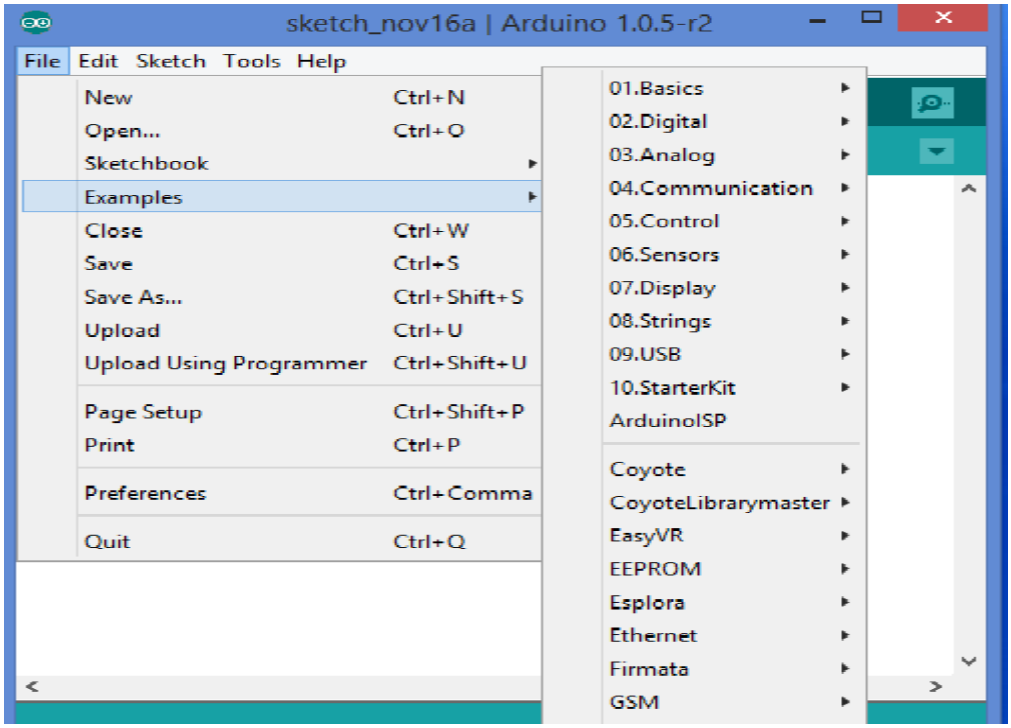


Fig.6 List of examples in Arduino™ Software.

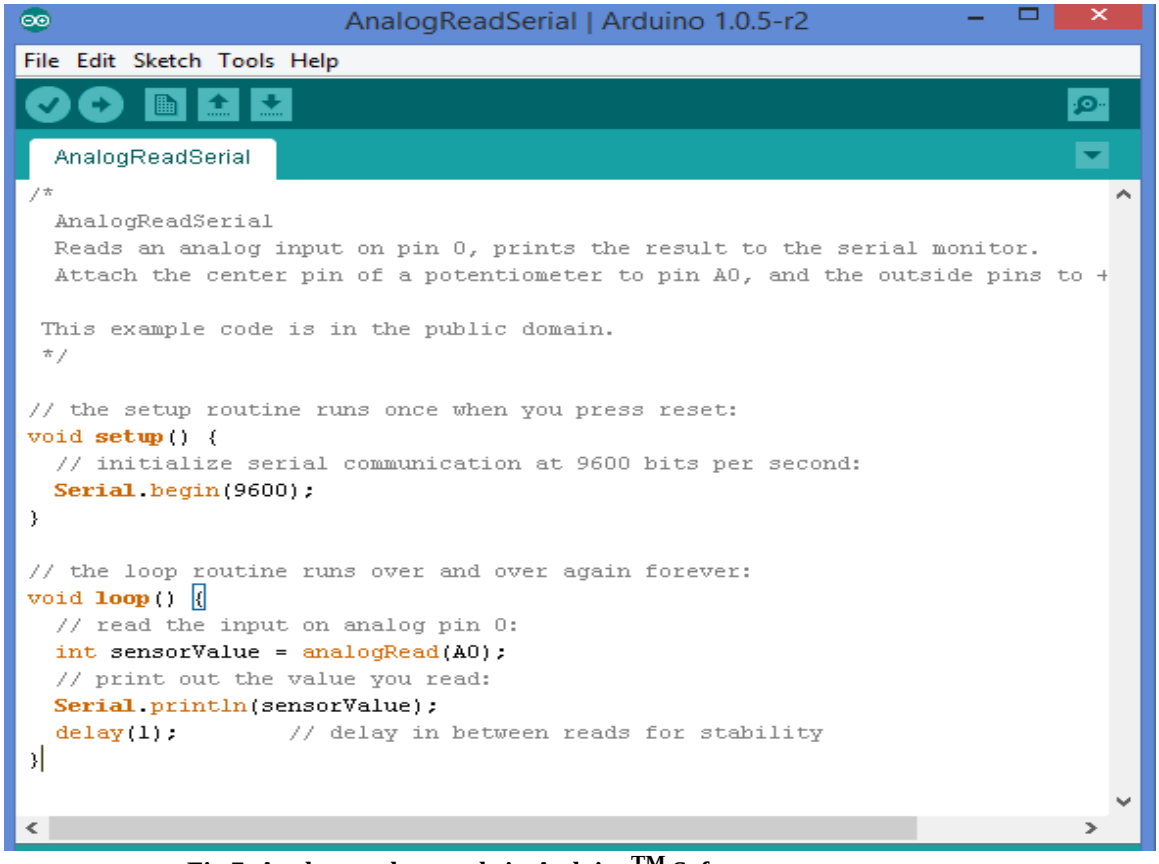


Fig.7 Analog read example in Arduino™ Software.

2.3. Introduction To Arduino™ compatibility in Simulink™

The user has to make sure to install Matlab™software in the system which consists of Simulink™ package before going into this step. The steps of installing Arduino™ support package are given below:

i. Open the Matlab™software window and give the command "targetinstaller".

ii. A new GUI will pop up shown in Fig.8 with options of installing target using Internet and Folder.

iii. Recommended option is to select Internet and click on next.

iv. A list of all the hardware support packages will be displayed as shown in Fig.9.

v. Desired targets have to be selected and click on next.

vi. In the next window click on Install to complete the installation procedure.

vii. User can also download a support package from website [11].

viii. If the support package is downloaded the option of Folder has to be selected in point ii and follow the instructions.

After completing 2.3 users can create customized blocks explained in detail in Section 3.

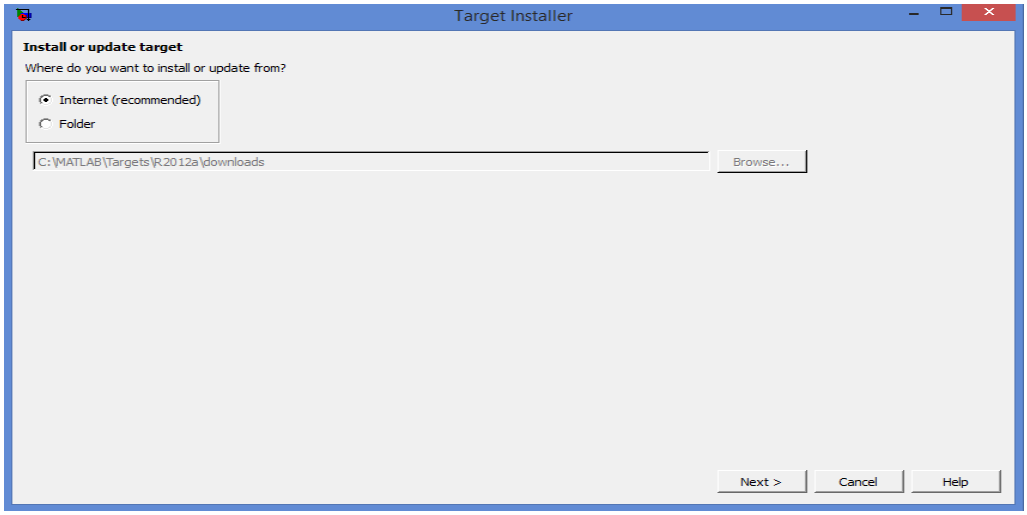


Fig.8 Target-Installer window.

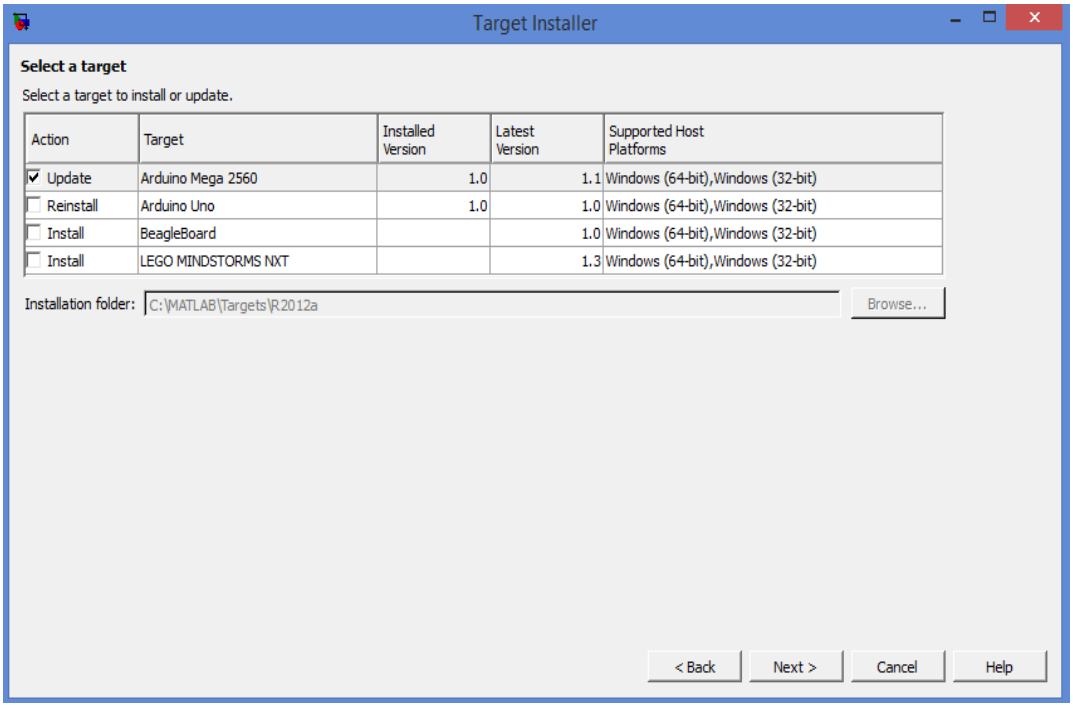


Fig.9 List of support packages supported by Matlab.

CHAPTER 3

In this chapter the author explains about basic requirements of the creation of a Simulink™ block and few customized blocks. The introductory part is common to all the blocks.

3.1 Introduction To Creating Blocks In Simulink™

The advantage of creating serial data transmission, EEPROM, LCD display and servo motor blocks is to create Simulink™ designs without prior knowledge of electronic components. The following is a detailed step-by-step procedure on how to develop a Simulink™ block.

Step1: Open the Simulink™ in Matlab™ and select “New model” in Simulink™ from File tab's drop down menu as shown in Fig.10. Keyboard shortcut "ctrl + n" will also create a new model.

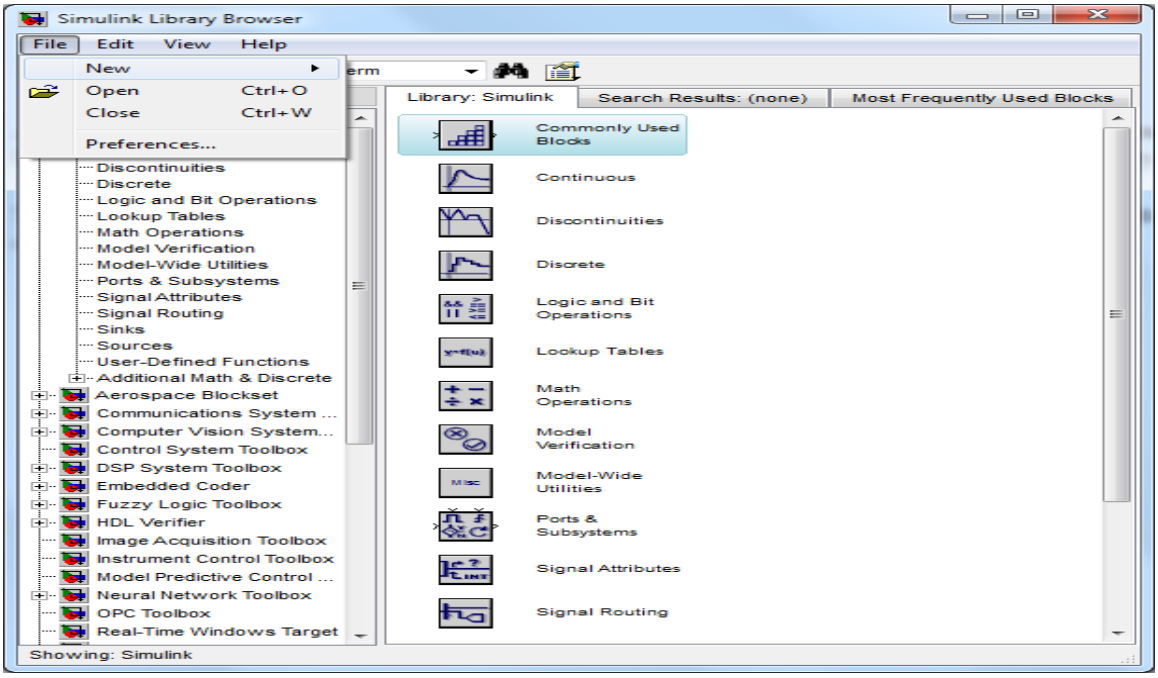


Fig.10 New model selection window.

Step 2: Once a new model is selected the screen will appear as shown in Fig.11

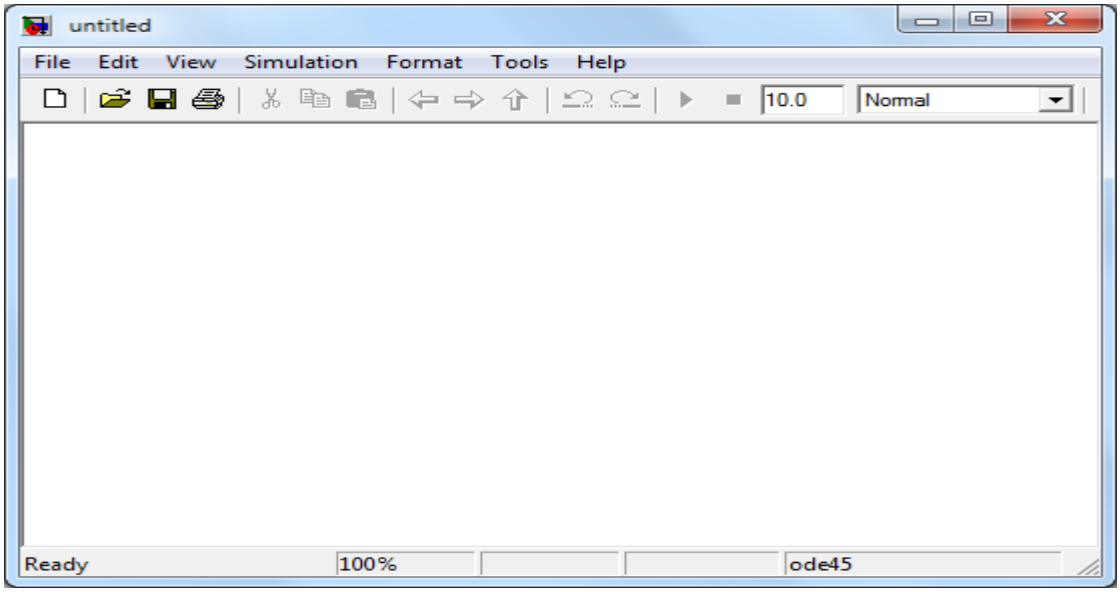


Fig.11 New untitled window for block creation.

One should drag the S-Function builder block from the user defined functions tab in the Simulink™ library as shown in Fig.12 and drop it in the new window.

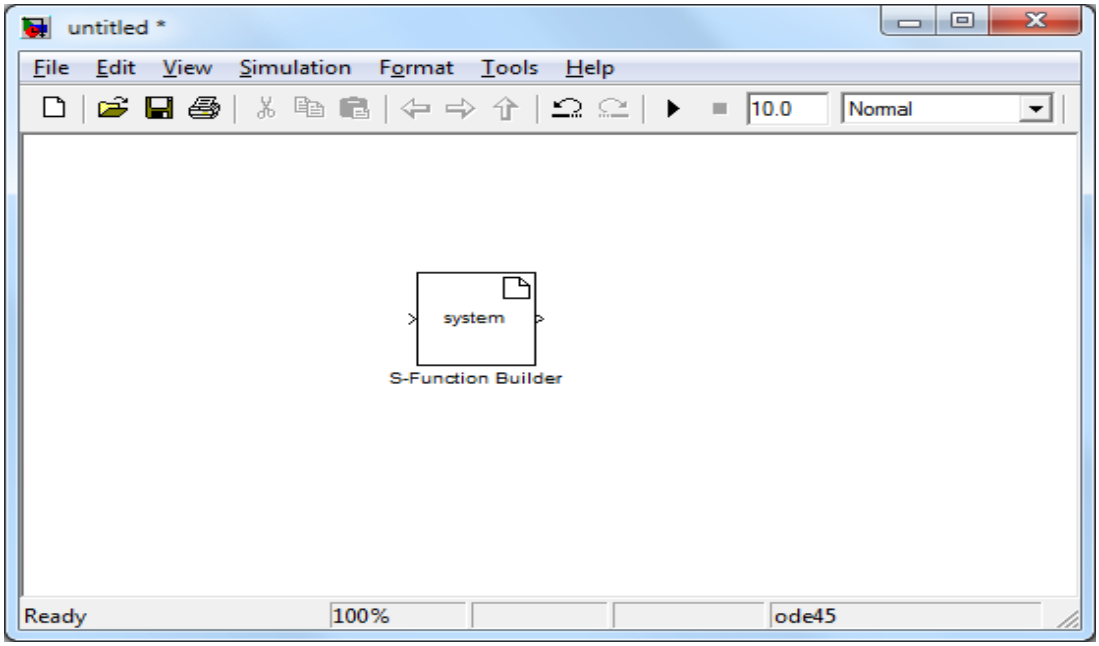


Fig.12 New window with S-Function Builder in it.

Step 3: One important thing one should check is whether the Matlab™ under use consists of a predefined/inbuilt compiler or not, for compiling the C/C++ custom code. The command to check the compiler availability in the Matlab™ command window is

mex -setup

It's a simple way to know the availability of all the compilers in the system. Type this command and press enter, a list of all the compilers present in the system will be displayed. If there are inbuilt compilers available then select the desirable compiler for compilation of the custom code by selecting the number corresponding to the list of compilers and press enter. If there are no inbuilt compilers available, the compiler should be first downloaded. This compiler performs the building of the wrapper file and other supporting files for running the block on the hardware.

After typing mex -setup in the Matlab™ command window the message shown in Fig.13 will be displayed (Note: A space should be present between the mex and -setup).

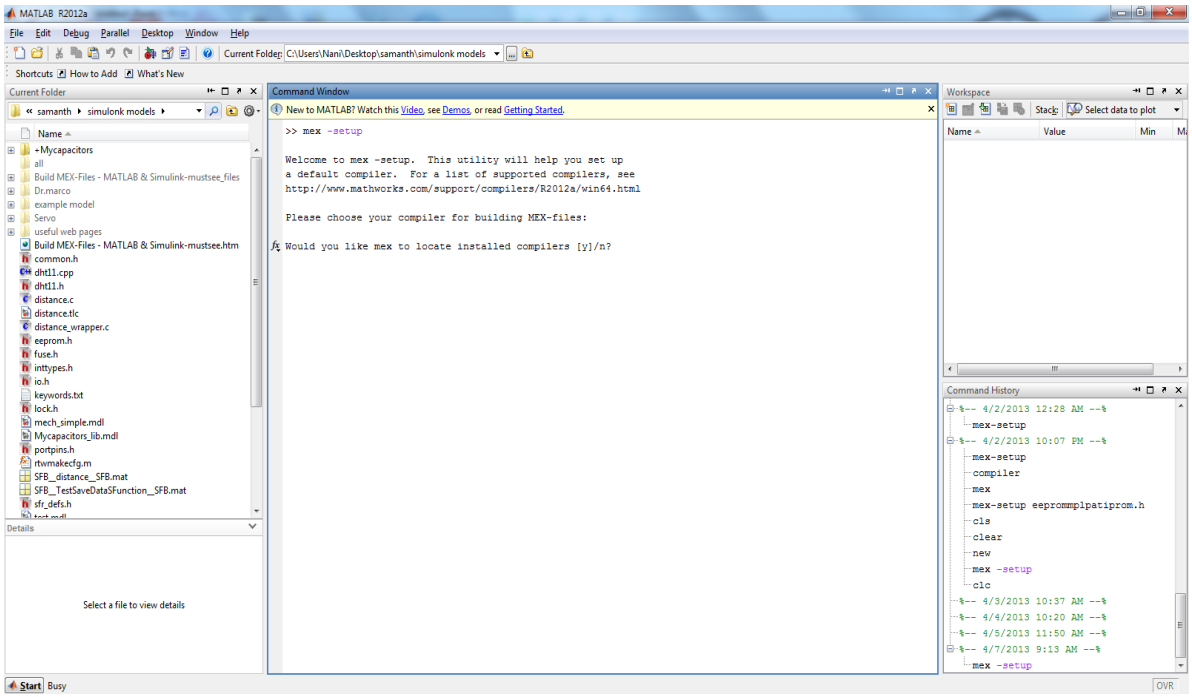


Fig.13 mex –setup display message

One should press y and enter, to get a list of all the compilers available in the system as shown in Fig.14.

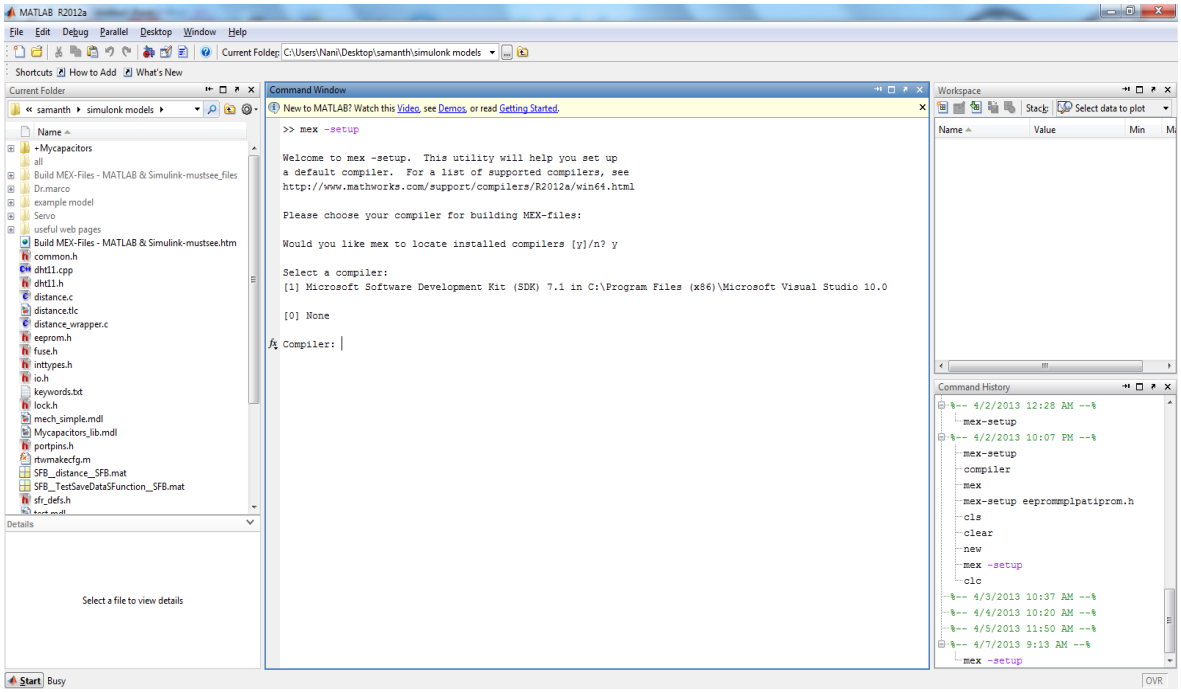


Fig.14 List of all the compilers present in the system.

As there is only one compiler in the demonstration system, one available option in the compiler's list is displayed. If there are multiple compilers in a system then all of them will be displayed. The number of the desirable compiler has to be given for compilation of the custom code and press enter. This will make the compiler, selected as a default compiler for compiling the custom codes. The compiler can be changed by selecting a different compiler number.

Step 4: By double clicking the S-Function Builder icon as shown in the Fig.12, a new pop up screen appears as shown in Fig.15.

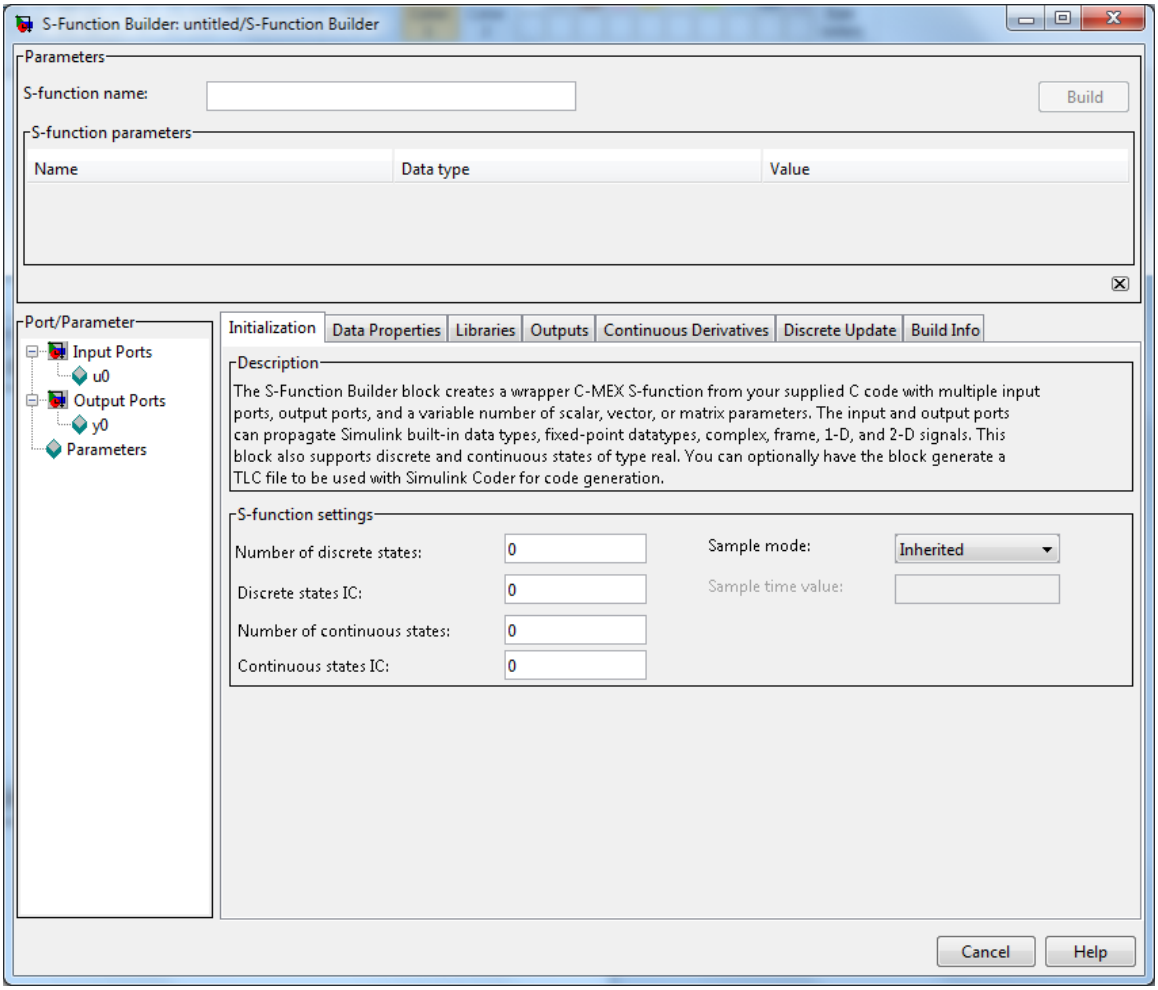


Fig.15 S-Function Builder code and parameter filling window.

Fig.15 shows the window with multiple panes of control options where one should enter information needed for defining the S-Function Builder block in order to build a customized S-function. The information on the panes and the controls that they contain is explained in detail including examples in the link [12]. For convenience this document [12] is given in Appendix A2.

Step 5: A small overview of what should be placed under each tab shown in Fig.15 is explained below.

Libraries Tab:

Under the Libraries tab in the right side corner one should place all the .h files that are useful for the block creation. Note: All the .h files placed here should be placed in the Simulink™ path of the working directory to avoid recognition errors.

An example of how the .h files should be include is shown below

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

Discrete Update tab:

The following code is placed in the Discrete update pane. The detailed explanation of each of these lines is explained after the code.

```
if(xD[0]!=1)
{
    #ifndef MATLAB_MEX_FILE
        pinMode(pin[0],OUTPUT);
    #endif
xD[0]=1;
}
```

The initial condition for the discrete state is 0 (this is set up by the initialization pane). The first time this Discrete Update function is called xD[0] is equals to 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.

The three central lines inside the brackets are:

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

The "MATLAB_MEX_FILE" will be defined at compilation time by the MEX file. The MEX file is generated from the S-Function Block. 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 does 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 (shown in code above) which will set the mode of the pin specified by the parameter pin[0] (12 in this case) to "OUTPUT". Pin[0] is set in the parameter pane. Detailed information about the pinMode is explained in the reference link [13].

When writing the output block, it is a good idea to start with this block and replace the line "pinMode(pin[0],OUTPUT);" with any initialization code one might need. Initialization code here meant the code which assigns a particular pin to specific purpose. Here pin[0] was assigned as an output pin. If no initialization code is needed, then only pinMode(pin[0],OUTPUT) 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).

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.

Outputs Tab:

The following code is placed inside the Outputs tab. Detailed explanation of each line of this code and its purpose is explained in detail after the code.

```
if(xD[0]==1)
{
  /*don't do anything for MEX-file generation*/
  #ifndef MATLAB_MEX_FILE
```

```
    /* Whatever the functional code snippet or actual run code that one want to run on the
target an example is placed below*/
        digitalWrite(pin[0],in[0]); //Writing the data in in[0] to the pin[0]
   #endif
}
```

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, one can type the code directly in the edit field.

The first thing to notice 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. Therefore the code in the brackets is executed only after the initialization code has already been executed.

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

When using this block as a starting point to create a driver (driver here meant code section which performs a desired task), the Arduino™ specific instruction "digitalWrite(pin[0],in[0]);" (here data from in[0] is written to pin[0]) should be replaced with a custom target specific code. 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, 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.

These are the three main tabs which should be carefully watched. The rest of all the tabs are explained in detail in the link provided, see [14]. For convenience this document [14] is given in Appendix A2.

Step 6: Once the above steps are successfully completed then one should click on the Build tab. If everything goes well and one has followed all the steps stated here, then build success message will be displayed and one can see the built files in the Build Info pane.

3.2 Serial Data Transmission Block

The idea behind the creation of this block is to use it as a single point source to read and write all the serial data required or produced by other components in a complex design. The following section explains the details on how this block should be created. Step 1 to step 3 of Section 3 are the common steps in creation of this block. Step 4 is a very important step in the creation of this block. In this step a sample Arduino™ serial code was taken as a base reference [16]. The idea behind taking Arduino™ code is to get rid of any compatibility issues.

The data that has to be entered in each tab of the S-Function Builder is as follows

a. Initialization tab: Number of discrete states = 1, Discrete states IC = 0, Number of Continuous states = 0, Continuous states IC = 0, Sample mode = discrete and sample time value = 0.05.

b. Data Properties: Input ports, Output ports, Parameters and Data type attributes have to assigned depending on the number of input and output connections to this block.

c. Libraries: There are 3 major sections in this tab

   i. Library/Object/Source files: This is the place where one should place the path for reference libraries that have to be defined. This is an optional field. If all the reference libraries of a particular block are present in current working directory then this field can be left alone.

      Example:

      The Library/Object/Source files for the serial data transmission block are

      C:\Users\Samanth\Desktop\Dr.Marco\aurdinofile\arduino-1.0.3\libraries\SoftwareSerial\examples\SoftwareSerialExample

   ii. Includes: All the include files have to be defined in here. One important thing that has to be taken into consideration is all the include files must be defined in between #ifndef MATLAB_MEX_FILE and #endif.

      The contents of this particular block are

      ```
      # ifndef MATLAB_MEX_FILE
      #include "SoftwareSerial.cpp"
      extern SoftwareSerial mySerial(10, 11); // RX, TX
      # endif
      ```

   iii. External function declaration: All the external function declarations have to be defined in here.

d. Outputs: This is one section where part of the actual working code has to be placed. The code which is placed under this section will execute once the code in the Discrete Update section executes and sets the xD[0] = 1. The code under this section will be the main executable code. The entire code has to be placed in between if (xD[0]==1) {} and the main working code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif.

The code for this particular block is

```
if (xD[0]==1)
 {
   /* don't do anything for mex file generation */
  # ifndef MATLAB_MEX_FILE
    if (mySerial.available())
      Serial.write(mySerial.read());
    if (Serial.available())
      mySerial.write(Serial.read());
  # endif
 }//end of if (xD[0]==1)
```

e. Continuous Derivatives: This is an optional section and is used to calculate derivatives. This section was left alone for this block.

f. Discrete Update: This is the other section where the initialization part of the code is present. The code execution first enters this block and performs all the initializations. The entire code has to be placed in between if (xD[0]!=1){} and the main initialization code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif.

The code for this particular block is

```
if (xD[0]!=1)
{
      /* don't do anything for MEX-file generation */
     # ifndef MATLAB_MEX_FILE
```

```
        // Open serial communications and wait for port to open:
          Serial.begin(57600);
        // Printing the Hello world
          Serial.println("Hello world!");
       // set the data rate for the SoftwareSerial port
         mySerial.begin(4800);
         mySerial.println("Good morning");
      # endif
     /* initialization done */
     xD[0]=1;
}
```

3.3 EEPROM Block

The idea behind the creation of this block is to use it as a single point source to read and write data to a SD card (SD card is an EEPROM device) by other components in a complex design. The following section explains the details on how this block should be created. Step 1 to step 3 of Section 3 are the common steps in the creation of this block. Step 4 is a very important step in the creation of this block. In this step, a sample Arduino™ EEPROM read/write code was taken as a base reference [16]. The idea behind taking Arduino™ code is to get rid of any compatibility issues.

The data that has to be entered in each tab of the S-Function Builder is as follows

a. Initialization tab: Number of discrete states = 1, Discrete states IC = 0, Number of Continuous states = 0, Continuous states IC = 0, Sample mode = discrete and sample time value = 0.05.

b. Data Properties: Input ports, Output ports, Parameters and Data type attributes have to assigned depending on the number of input and output connections to this block. By default Input ports and output ports are not assigned. Parameters tab consists of one pin declaration with a data type unit8 and real complexity.

c. Libraries: There are 3 major sections in this tab

i. Library/Object/Source files: This is the place where one should place the path for reference libraries that have to be defined. This is an optional area as far as all the files are placed in the current working directory. For this block this section was left alone.

ii. Includes: All the include files have to be defined in here. Include files must be defined in between #ifndef MATLAB_MEX_FILE and #endif. The contents of this particular block based on the Author's example are:

```
# ifndef MATLAB_MEX_FILE
    # include "Arduino.h"//Include the .h file
    #include "EEPROM.cpp" //Include the CPP file
    #include "SoftwareSerial.cpp"//Include the CPP file
    uint8_t addr=0; //Global variable declaration
    uint8_t val=4;
# endif
```

iii. External function declaration: All the external function declarations have to be defined in here. This section was left alone for this block.

d. Outputs: This is one section where part of the actual working code has to be placed. The code which is placed under this section will execute once the code in the Discrete Update section executes and sets the xD[0] = 1. The code under this section will be the main executable code. Two important things under this section are the entire code has to be placed in between if (xD[0]==1) {} and the main working code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif.

The contents of this particular block are

```
/* wait until after initialization is done */
if (xD[0]==1)
{
```

```
/* don't do anything for mex file generation */
# ifndef MATLAB_MEX_FILE
        begin:  //Switch to case statement
        EEPROM.write(addr,val); //write function writing data
    "val" to address location "addr"
        delay(100); //Delay of 100 milliseconds
        Serial.print(addr); //Printing the Address location
        Serial.print("\t"); //Printing a tab space
        Serial.print(val); // Printing the value
        val = EEPROM.read(addr); // Read function reading a
    value from address location "addr"
        delay(100);
        Serial.print(addr);
        Serial.print("\t");
        Serial.print(val);
        addr = addr+1; //Incrementing the address value by 1
        if(addr==512) //Checking for address value = 512 and if
    a match is found then resetting the address to zero again
        {
          addr = 0;
        }
        else
        {
          goto begin; //If the address hasn't reached the 512
    value then continue the process of writing and reading
        }
# endif

}
```

e. Continuous Derivatives: This is an optional section and is used to calculate derivatives. This section was left alone for this block.

f. Discrete Update: This is the other section where the initialization part of the code was present. The execution of the code first enters in to this block and performs all the initializations. One important thing under this section the entire code has to be placed in between if (xD[0]!=1){} and the main initialization code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif.

The contents of this particular block are

```
if (xD[0]!=1) {
      # ifndef MATLAB_MEX_FILE
        Serial.begin(9600); //Setting the baudrate for serial
communication
      # endif
   xD[0]=1;
}
```

One special case of the EEPROM module is the use of an SD-card module. This module can be used with generated block in the same fashion. The SD card module has 8 pins which have to be connected to the UNO board. The connections should be done using a bread board. The connections are following

SD card -------> UNO
5v 5v
gnd gnd
MOSI 11 pin
SS 10 pin
SCK 13 pin
MISO 12 pin

The SD card module pin configuration is shown in Fig.16.

Fig.16 SD card module

For purchasing this board and spec details of this board use this reference [15].

3.4 LCD display Block

The idea behind the creation of this block is to use it as a single point source to display the data produced by other components in a Simulink™ model. This was also created as a starting point for developing the touch interface for users, similar to a smartphone. One major difference between other blocks and this particular block is it required an Arduino™ Mega board. The mounting of the LCD interface on the Arduino™ Mega board should be done as shown in Fig [19].

The following section explains the details on how this block should be created. Step 1 to step 3 of Section 3 are the common steps in creation of this block. Step 4 is a very important step in the creation of this block. In this step a sample Arduino™ LCD code was taken as a base reference[16]. The idea behind taking Arduino™ code is to get rid of any compatibility issues.

The data that has to be entered in each tab of the S-Function Builder is as follows

a. Initialization tab: Number of discrete states = 1, Discrete states IC = 0, Number of Continuous states = 0, Continuous states IC = 0, Sample mode = discrete and sample time value = 0.05.
b. Data Properties: Input ports, Output ports, Parameters and Data type attributes have to be assigned depending on the number of input and output connections to this block. For the demo block the inputs and outputs are left alone. The Parameters tab consists of one pin assigned with data type unit8 and real complexity.
c. Libraries: There are 3 major sections in this tab

i. Library/Object/Source files: This is the place where one should place the path for reference libraries have to be defined. This is an optional field. This can be left blank if all the files of library are present in the working directory of Matlab.

Example:

The path for this particular block in the demo system of author is C:\Users\Samanth\Desktop\Professors\Dr.Marco\aurdino file\arduino-1.0.3\libraries\UTFT

ii. Includes: All the include files have to be defined in here. One important item that has to be taken in to consideration is that all the include files must be defined in between #ifndef MATLAB_MEX_FILE and #endif.

The contents of this particular example demo block of author are

```
# ifndef MATLAB_MEX_FILE
    #include "UTFT.h" //Including the header and CPP files
    #include "UTFT.cpp"
    #include "DefaultFonts.c"
    #include "UTouch.h"
    #include "UTouch.cpp"
    extern uint8_t SmallFont[]; //Declaring the global variables
    extern uint8_t BigFont[];
    extern uint8_t SevenSegNumFont[];
    UTFT myGLCD(ITDB32S,38,39,40,41); //1st parameter=LCD model name; 2nd = ReadSelect pin, Write Pin, 3rd=Chip Select pin;4th=Reset Pin; 5th=Serial pin
# endif
```

iii. External function declaration: All the external function declarations have to be defined in here. This section was left alone for this block.

d. Outputs: This is one section where part of the actual working code has to be placed. The code which is placed under this section will execute once the code in

the Discrete Update section executes once and sets the xD[0] = 1. The code under this section will be the main executable code. Placement of code in between if (xD[0]==1) {} is very important.

The contents of this example demo block of author are

```
if (xD[0]==1)
{
      # ifndef MATLAB_MEX_FILE
                  int x, y;   //Local variable declaration; these variables can be used only in this section of code
                  char stCurrent[20]=""; //Declaring a null character array of size 20
                  int stCurrentLen=0;
                  char stLast[20]="";
                  void updateStr(int val) //String update function
                  {
                   if (stCurrentLen<20) //Max size of string to enter this loop is 20
                   {
                     stCurrent[stCurrentLen]=val;
                     stCurrent[stCurrentLen+1]='\0';
                     stCurrentLen++;
                     myGLCD.setColor(0, 255, 0); //1st parameter = red color, 2nd parameter = green color and 3rd parameter = blue color
                     myGLCD.print(stCurrent, LEFT, 224); //printing the current value of string on left side of screen with a color code of 224
                   }
                   else //If the string length is greater than 20 enter this loop
                   {
                     myGLCD.setColor(255, 0, 0); //setting the text color on screen
                     myGLCD.print("BUFFER FULL!", CENTER, 192); //As the string size has crossed the limit of 20 warning message is displayed
                     delay(500); //Waiting for 500 milliseconds
                     myGLCD.print("           ", CENTER, 192); //Clearing the screen
                     delay(500);
                     myGLCD.print("BUFFER FULL!", CENTER, 192); //Again printing the warning message to create a visual flash effect
                     delay(500);
                     myGLCD.print("           ", CENTER, 192);
                     myGLCD.setColor(0, 255, 0);
```

```
  } //end of else
} //end of void updateStr(int val)

   if(pin[0]==1) //If the input pin value = 1 then enter loop
    {
      myGLCD.setColor(0, 255, 0);
      myGLCD.setBackColor(0, 0, 0); //Intial setting of
background color on LCD scrren
      myGLCD.fillScr(255,160,255); //Fill the entire screen with
selected color combination passed as arguments
      delay(1500);
      myGLCD.fillScr(VGA_RED); //Filling the screen with Red
color
      delay(2500);
      myGLCD.setXY(0,0,240,320); //Setting the XY coordinates
of the screen. Coordinates x1=0,Y1=0,X2=240,Y2=320
      delay(2500);
          myGLCD.fillScr(VGA_BLACK);
          delay(2500);
          myGLCD.setFont(BigFont); //Setting the Big Font format
          myGLCD.print(" SAMANTH", CENTER, 0);
         //Displaying the string SAMANTH at the center of the
         LCD
          delay(3000);
          myGLCD.fillScr(VGA_GREEN); //Filling the screen with
         Green color
          delay(2500);
          myGLCD.print("DR.MARCO", CENTER, 16);
         //Displaying the string  DR.MARCO at the center of the
         LCD
          delay(2500);
          myGLCD.fillScr(VGA_BLUE); //Filling the screen with
         Blue color
          delay(2500);
          myGLCD.print("@ISU", CENTER, 32); //Displaying the
         string  @ISU
          delay(2500);
          myGLCD.fillScr(VGA_AQUA); //Filling the screen with
         AQUA color
          delay(2500);
```

```
                myGLCD.print("THANKYOU DR", CENTER, 64);
                delay(2500);
                myGLCD.print("FOR BELIEVING ME", CENTER, 80);
                delay(2500);
                myGLCD.fillScr(VGA_OLIVE);
                delay(1500);
                myGLCD.printNumI(11, 30, 30, 2, 0);
                delay(1500);
                myGLCD.setFont(BigFont);
                myGLCD.print(" !\"#$%&'()*+,-./", CENTER, 0); //All
              the print commands will print the string passed as first
              argument at the second argument (i.e.CENTER here)
              location; 3rd parameter=degrees
                myGLCD.print("0123456789:;<=>?", CENTER, 16);
                myGLCD.print("@ABCDEFGHIJKLMNO", CENTER,
              32);
                myGLCD.print("PQRSTUVWXYZ[\\]^_", CENTER,
              48);
                myGLCD.print("`abcdefghijklmno", CENTER, 64);
                myGLCD.print("pqrstuvwxyz{|}~ ", CENTER, 80);

                myGLCD.setFont(SmallFont);
                myGLCD.print(" !\"#$%&'()*+,-./0123456789:;<=>?",
              CENTER, 120);

              myGLCD.print("@ABCDEFGHIJKLMNOPQRSTUVWX
              YZ[\\]^_", CENTER, 132);
                myGLCD.print("`abcdefghijklmnopqrstuvwxyz{|}~ ",
              CENTER, 144);

                myGLCD.setFont(SevenSegNumFont);
                myGLCD.print("0123456789", CENTER, 190);
                delay(2500);
                myGLCD.clrScr();
                myGLCD.printNumF(1988,2,CENTER,0);
                delay(3500);

            } //end of if(pin[0]==1)
              //digitalWrite(pin[0],in[0]);
# endif
```

```
}//end of if (xD[0]==1)
```

e. Continuous Derivatives: This is an optional section and is used to calculate derivatives. This section was left alone for this block.

f. Discrete Update: This is the other section where the initialization part of the code was present. The execution of the code first enters in to this block and performs all the initializations. At the end of this block the value of xD[0] is made equal to 1. The main initialization code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif. The drawButtons() contents in this block are required for future work of creating the buttons on the touch screen to make it an interactive user interface. Therefore for this demo purpose the user can remove the entire drawButtons() code in between the #ifndef and #endif statements.

The contents of this demo example block of author are

```
if (xD[0]!=1) {
        /* don't do anything for MEX-file generation */
    # ifndef MATLAB_MEX_FILE
        myGLCD.InitLCD(); //Initializing LCD screen
        myGLCD.clrScr(); //Clearing screen to obtain blank screen
    # endif
        /* initialization done */
        xD[0]=1;
      }
```

LCD consists of 40 pins. It is connected to Arduino™ Mega board using a special shield. The LCD cannot be directly placed on top of Mega board. LCD shield board is connected to the Mega board. On top of the shield board there is a slot of 40 pins and the LCD has to be connected to it. The numbering on the shield should match the numbering

on the LCD. The way in which the LCD board was connected to the Arduino™ Mega board is shown pictorially in the Fig.17, Fig.18, Fig.19.

Fig.17 LCD Module top view with 40 pins on left side

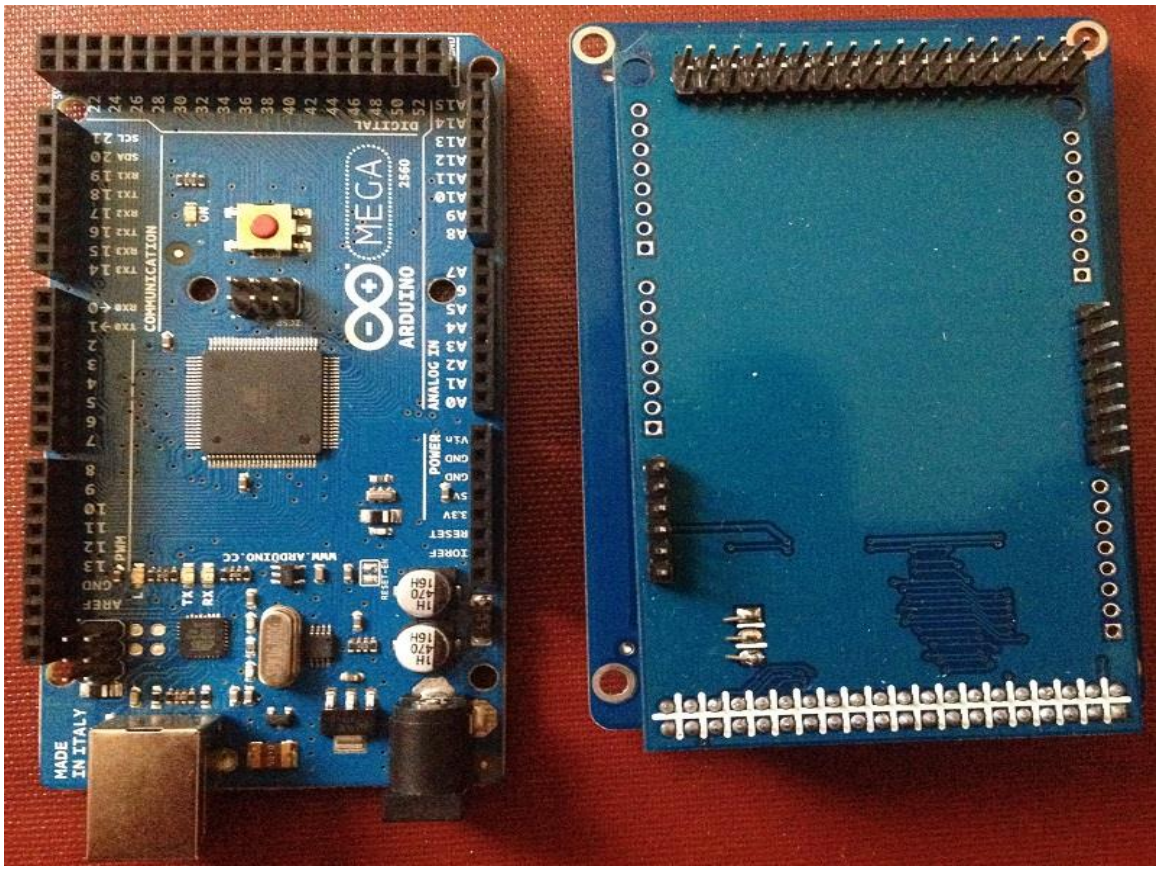


Fig.18 LCD shield module on right and Mega board on left

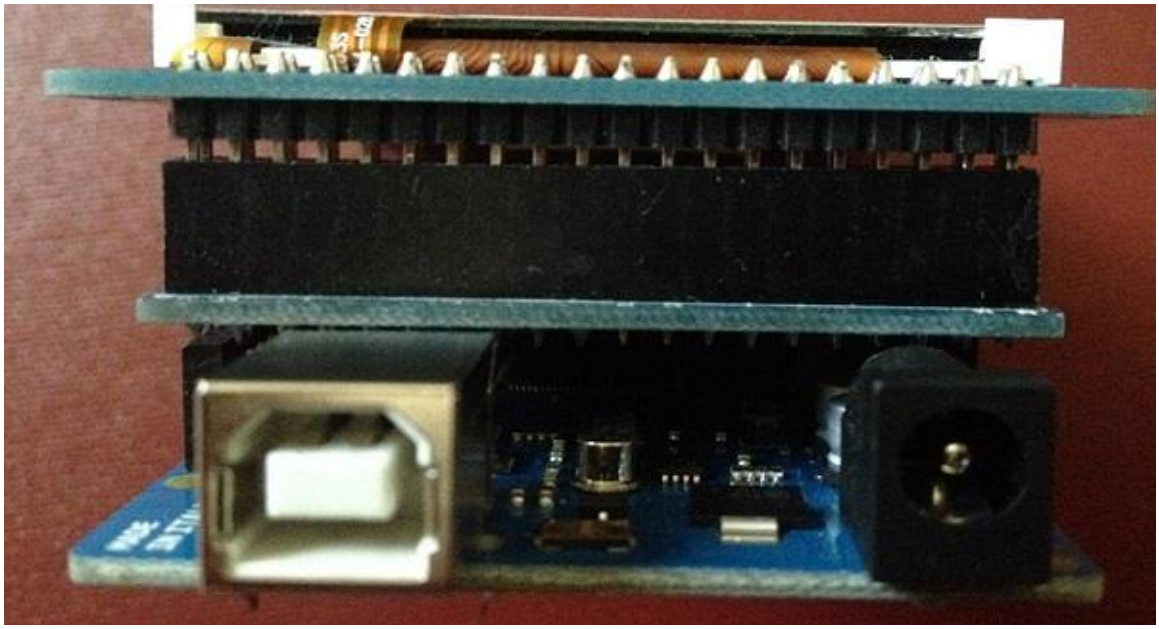


Fig.19 LCD module connected to shield and shield connected to Mega board

3.5 SERVO Motor Block

The idea behind the creation of this block is to use it as a single point source to use a servo motor by other components in a Simulink™ design. The following section explains the details on how this block should be created. Step 1 to step 3 of Section 3 are the common steps in creation of this block. Step 4 is very important step in creation of block. In this step a sample Arduino™ Servo sweep code was taken as a base reference [16]. The idea behind taking Arduino™ code is to get rid of any compatibility issues.

The data that has to be entered in each tab of the S-Function Builder is as follows

a. Initialization tab: Number of discrete states = 1, Discrete states IC = 0, Number of Continuous states = 0, Continuous states IC = 0, Sample mode = discrete and Sample time value = 0.01.

b. Data Properties: Input ports, Output ports, Parameters and Data type attributes have to assigned depending on the number of input and output connections to this block. An input port named “pos” was assigned to this block to serve as an input source. Whatever may be the value of this pos it will be copied in to a localpos variable. Parameters tab consists of one pin declaration with a data type unit8 and real complexity.

c. Libraries: There are 3 major sections in this tab

    i. Library/Object/Source files: This is the place where one should place the path for reference libraries have to be defined. This is an optional area as far as all the files are placed in the current working directory. This is left alone for this block.

ii. Includes: All the include files have to be defined in here. One important thing that has to be taken in to consideration is all the include files must be defined in between #ifndef MATLAB_MEX_FILE and #endif.

The contents of this demo example block of author are

```
#ifndef MATLAB_MEX_FILE
        #include <Servo.cpp>
        #include <Servo.h>
        Servo myservo; // object creation for class Servo
#endif
```

iii. External function declaration: All the external function declarations have to be defined in here. This section was left alone for this block.

d. Outputs: This is one section where part of the actual working code has to be placed. The code which is placed under this section will execute once the code in the Discrete Update section executes once and sets the xD[0] = 1. The code under this section will be the main executable code. There are couple of important things under this section the entire code has to be placed in between if (xD[0]==1) {} and the main working code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif.

The contents of this particular section of this author's example block are

```
if(xD[0]==1)
{
   /*don't do anything for MEX-file generation*/
   #ifndef MATLAB_MEX_FILE
  int localpos =pos[0]; //Assigning the value of pos[0] into localpos

  myservo.attach(9);
  for(localpos = 0; localpos < 180; localpos += 1)  // goes from 0
degrees to 180 degrees
  {                              // in steps of 1 degree
```

```
    myservo.write(localpos);              // tell servo to go to position in
variable 'pos'
    delay(15);                    // waits 15ms for the servo to reach the
position
  }
  for(localpos = 180; localpos >=1; localpos -=1)     // goes from
180 degrees to 0 degrees
  {
    myservo.write(localpos);              // tell servo to go to position in
variable 'pos'
    delay(15);                    // waits 15ms for the servo to reach the
position
  }

#endif
}
```

e. Continuous Derivatives: This is an optional section and is used to calculate derivatives. This section was left alone for this block.

f. Discrete Update: This is the other section where the initialization part of the code was present. The execution of the code first enters in to this block and performs all the initializations. There are couple of important things under this section the entire code has to be placed in between if (xD[0]!=1){} and the main initialization code has to be placed in between #ifndef MATLAB_MEX_FILE and #endif.

   The contents of this particular section for this block are

```
if(xD[0]!=1)
{
      #ifndef MATLAB_MEX_FILE
      #endif
xD[0]=1;
}
```

The servo motor consists of three wires of colors brown, red, and orange. The red wire has to be connected to the 5V supply of the UNO board. The Brown wire has to be

connected to the GND pin of UNO board. The Orange wire has to be connected to pin 9 of UNO board. The orange wire connection can vary depending on user on which pin one wants the output and rest of two are common. The connection of the servo motor and UNO board are shown in the Fig.20.

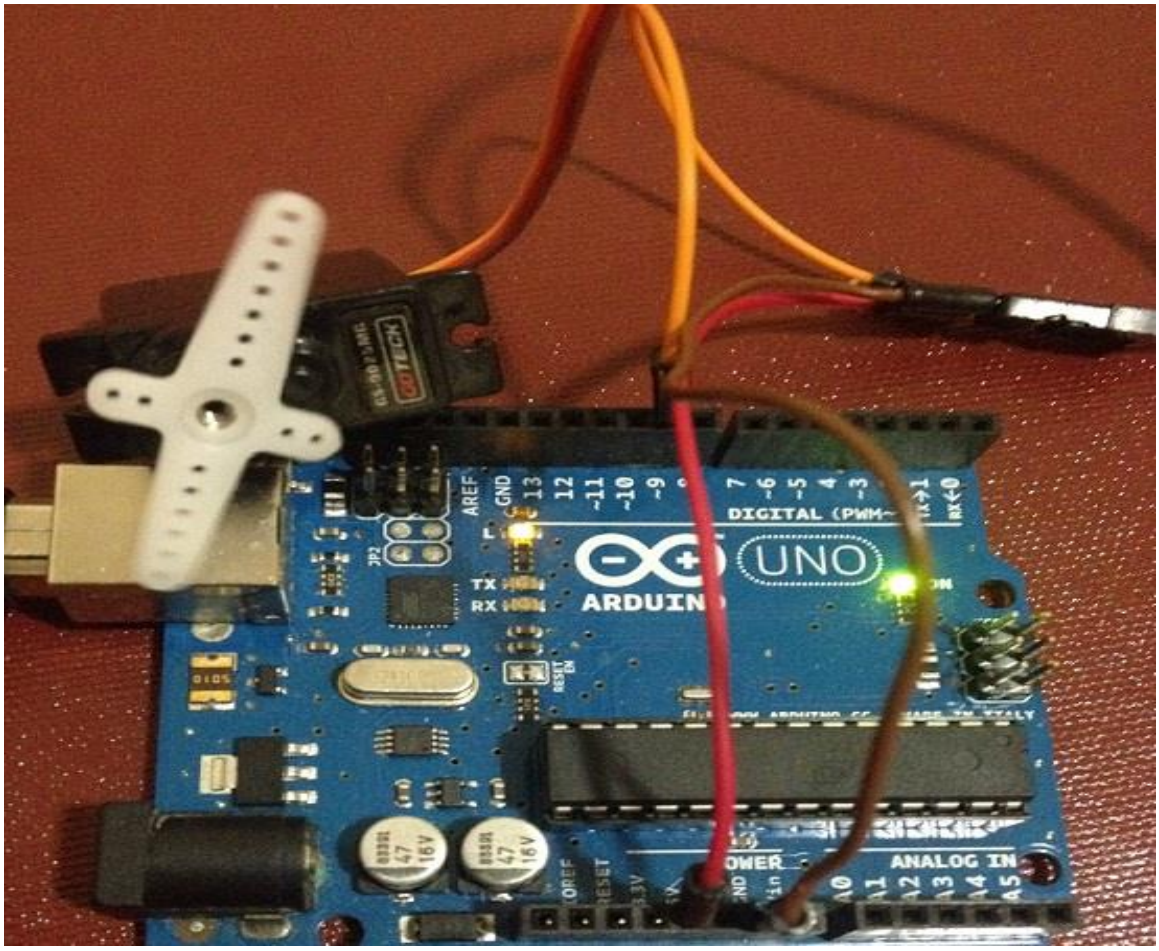


Fig.20 Servo motor connection to UNO board

3.6 VOICE CONTROLLED BLOCK

This block was created to control a servomotor using voice commands. The idea behind the creation of this block is to showcase the voice control capability of the motors in a typical prosthetic arm. The code in the block can be expanded to add more components. Step 1 through step 3 of Section 3 is the common steps in creation of this

block. The rest of the steps as stated in Section 3 vary in creation of this block. The EasyVR access demo code in the Arduino™ example set [16] is taken as a base reference in the creation of this block. The idea behind taking Arduino™ code is to get rid of any compatibility issues.

The data that has to be entered into each tab of the S-Function Builder pane is as follows

a. Initialization tab: This tab consists of values of S-Function settings. Number of discrete states equal to 1, Discrete states IC equal to 0, Number of continuous states equal to 0, Continuous states IC equal to 0, Sample mode equal to Discrete, Sample time value equal to 0.05. In this section almost all the parameters are constant only value of interest is Sample time value it can be increased or decreased depending on speed one wants. With the increase in the Sample time value the frequency decreases and vice versa.
b. Data Properties tab: This tab consists of four sub tabs Input ports, Output ports, Parameters and Data type attributes. Depending on the input and output blocks that are connected to this block one should add the number of inputs or outputs. The contents for this block are shown in Fig.21, Fig.22, Fig.23.

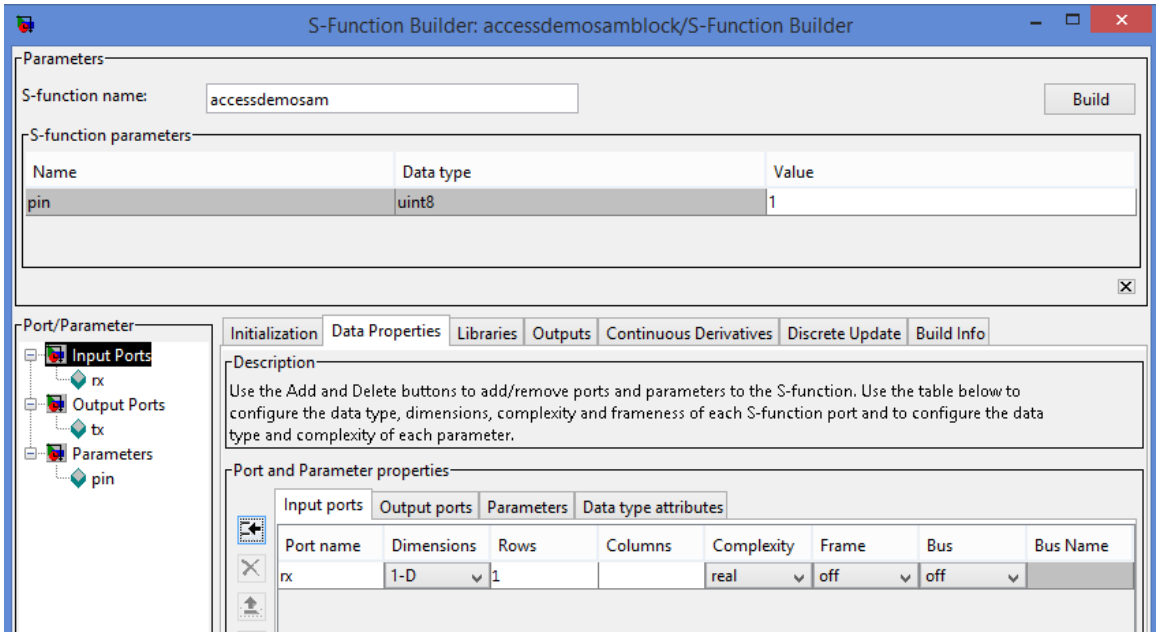


Fig.21 Input ports tab contents

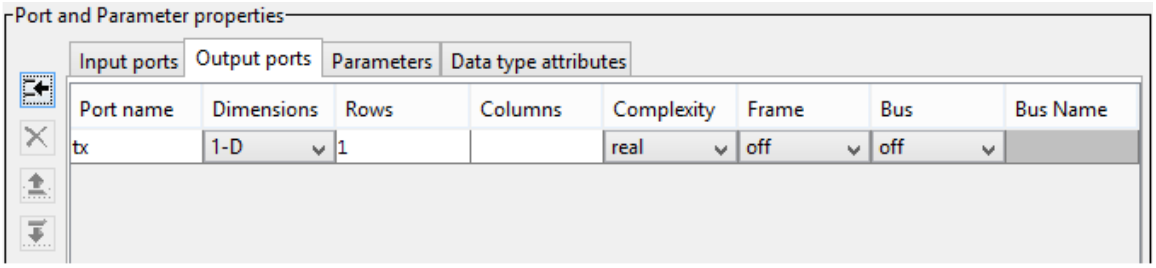


Fig.22 Output ports tab contents

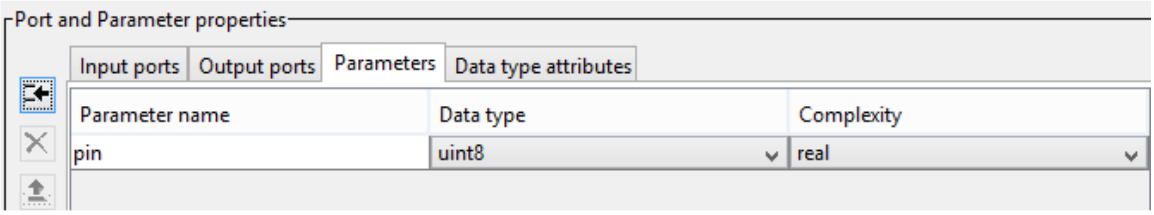


Fig.23 Parameters tab contents

c. Libraries tab: This tab plays a vital role. A lot of errors will arise while building the block due to misplacement of contents required for proper working of the block. This tab consists of 3 sub sections namely Library/Object/Source files, External Function declarations and Includes. Includes section is a mandatory section the rest of the two sections are optional and can be left alone. The contents of the include section of the example block demonstrated by author consist of

```
#include "EasyVRBridge.cpp" //including the C++ file
#include "Servo.cpp" //including the C++ file
SoftwareSerial port(12, 13); // Function declaration with RX, TX as
inputs
EasyVR easyvr(port);       // Function declaration with port
variable as input
bool checkMonitorInput(); // Function declaration
int8_t set = 0;           // Global variable
int8_t group = 0; // Global variable used to define command group
number
uint32_t mask = 0;  // Global variable used in discrete update &
output blocks
uint8_t train = 0;  // Global variable to hold value of number times
command is trained
char name[32]; // character array
int pos = 0;    // variable to store the servo position
bool useCommands = true;
EasyVRBridge bridge; //creating a class object to access
EasyVRBridge class elements
Servo myservo; //creating a class object to access Servo class
elements
#define SND_Access_denied            1 //defining the variables used
by the board to say back these voice commands to user
#define SND_Access_granted           2
#define SND_Hello                    3
#define SND_Please_repeat            4
#define SND_Please_say_your_password 5
#define SND_Please_talk_louder       6
# endif
```

d. Outputs Tab: The code segment which is responsible for data manipulation to produce the intended result/ output is placed inside this tab. The variables which are used in this tab along with Discrete Update tab should be declared as global variables. The function definitions should also be defined as global definitions if they are used by both Outputs Tab and Discrete Update tab. The contents of this tab should be placed inside the

```
if(xD[0]==1)
{
#ifndef MATLAB_MEX_FILE
// Actual working code should be placed here.
#endif
}
```

The example block explained by author consists of the following code segment in the Outputs Tab.

```
/* wait until after initialization is done */
if (xD[0]==1)
{
    /* don't do anything for mex file generation */
    # ifndef MATLAB_MEX_FILE
      int idx_cmd;
      int idx_pwd;
      easyvr.setPinOutput(EasyVR::IO1, HIGH); // LED on (listening)
      Serial.println("Say a name in Group 1");
      easyvr.recognizeCommand(1); // recognise command in group 1
      while (!easyvr.hasFinished()); // wait for user name
      easyvr.setPinOutput(EasyVR::IO1, LOW); // LED off
      idx_cmd = easyvr.getCommand(); // get recognised user name
      if (idx_cmd >= 0)
      {
       Serial.print("Name: ");
       if (easyvr.dumpCommand(1, idx_cmd, name, train))
         Serial.println(name); //this dumps the trained data in to serial
terminal for display.
       else
       Serial.println();
       // ask for password in the next command
       easyvr.playSound(SND_Please_say_your_password , EasyVR::VOL_FULL);
       easyvr.setPinOutput(EasyVR::IO1, HIGH); // LED on (listening)
       Serial.println("Say the password");
       easyvr.recognizeCommand(EasyVR::PASSWORD); // set group 16
       while (!easyvr.hasFinished()); // wait for password
       easyvr.setPinOutput(EasyVR::IO1, LOW); // LED off
       idx_pwd = easyvr.getCommand(); // get recognised password
       if (idx_pwd >= 0)
       {
        Serial.print("Password: "); //this dumps the trained password in to
system

        if (easyvr.dumpCommand(EasyVR::PASSWORD, idx_pwd, name, train))
          {
            Serial.print(" = ");
            Serial.println(name);
          }
        else
        Serial.println();
        if ( idx_pwd == idx_cmd) // index of username and password are the
same, access granted
      {
        Serial.println("Access granted");
        easyvr.playSound(SND_Access_granted , EasyVR::VOL_FULL);
        for(pos = 0; pos < 180; pos += 1)  // goes from 0 degrees to 180
degrees
          {                                  // in steps of 1 degree
              myservo.write(pos);              // tell servo to go to
position in variable 'pos'
             delay(15);                       // waits 15ms for the servo
to reach the position
```

```
          }
         for(pos = 180; pos>=1; pos-=1)     // goes from 180 degrees to 0
degrees
         {
           myservo.write(pos);              // tell servo to go to position
in variable 'pos'
           delay(15);                       // waits 15ms for the servo to
reach the position
         }

      }
      else // index of username and password differ, access is denied
      {
        Serial.println("Access denied");
        easyvr.playSound(SND_Access_denied , EasyVR::VOL_FULL);
      }
     }
    int16_t err = easyvr.getError();
    if (easyvr.isTimeout() || (err >= 0)) // if password timeout or access
is denied go in to this loop
    {
      Serial.println("Error, try again...");
      easyvr.playSound(SND_Access_denied , EasyVR::VOL_FULL);
    }
  }
  else
  {
    if (easyvr.isTimeout())
         Serial.println("Timed out, try again...");
        int16_t err = easyvr.getError();
    if (err >= 0)
    {
      Serial.print("Error ");
      Serial.println(err, HEX);
    }
  }
# endif
}//end of if (xD[0]==1)
```

e. Continuous Derivatives tab: This is an optional tab. If the user intended to perform derivative calculations this tab should be used. For this example case author is leaving this tab blank.

f. Discrete Update tab: This tab consists of data initialization code. All the variables are initialized here. The code execution enters in to the code defined in this tab. At the end of the code in this block the value of xD[0] is made equal to one from zero. The contents of this block should be placed inside the

```
 if(xD[0]!=1)
{
#ifndef MATLAB_MEX_FILE
// Actual initalization code should be placed here.
#endif
xD[0]=1;
}
```

The example block explained by author consists of the following code segment in the Discrete Update Tab.

```
if (xD[0]!=1)
{
    # ifndef MATLAB_MEX_FILE
      // bridge mode?
  if (bridge.check()) //checking whether the bridging between the
UNO and EasyVR board is formed properly
  {
    cli();
    bridge.loop(0, 1, 12, 13);
  }
   Serial.begin(9600); //setting the serial port baudrate
   port.begin(9600); //beginning the serial port with 9600 baud
   myservo.attach(9); //Servvo motor was attached to pin 9 on UNO
board
   if (!easyvr.detect()) //Searching for the EasyVR board
    {
     Serial.println("EasyVR not detected!");
     for (;;); //Waiting in a continuous loop
    }
   easyvr.setPinOutput(EasyVR::IO1, LOW); //initializing the
output to zero
   Serial.println("EasyVR detected!"); //Print command on
successful EasyVR detection
   easyvr.setTimeout(5);
   easyvr.setLanguage(EasyVR::ENGLISH); //Selecting the language
as English
   int16_t count = 0;
   if (easyvr.getGroupMask(mask)) // get trained user names and
passwords
    {
    uint32_t msk = mask;
    for(group = 0; group <= EasyVR::PASSWORD; ++group, msk >>= 1)
    {
      if (!(msk & 1)) continue; //This for making sure msk
variable is not equal to zero; only when the msk variable is not
equal to zero then the rest of the body in for loop executes.
```

```
      if (group == EasyVR::TRIGGER) //Checking if the group value
equal to the trigger word or username
        Serial.print("Trigger: ");
      else if (group == EasyVR::PASSWORD) //Checking if the group
value equal to password
        Serial.print("Password: ");
      else //This loop print “Group groupnumber:” Ex: Group1:
      {
        Serial.print("Group ");
        Serial.print(group);
        Serial.print(": ");
      }
      count = easyvr.getCommandCount(group); //Getting the number
of commands in the group. Ex: Number of commands in group 1
      Serial.println(count); //Printing the command count
      for (int8_t idx = 0; idx < count; ++idx) //This loop helps
in looping through all the elements in the group
      {
        if (easyvr.dumpCommand(group, idx, name, train)) //This
loop is used for printing the list of all the commands in a group
and list of trained commands
        {
          Serial.print(idx); //Index value
          Serial.print(" = ");
          Serial.print(name); //Name of the command
          Serial.print(", Trained ");
          Serial.print(train, DEC); //Printing the trained
commands
          if (!easyvr.isConflict()) //checking for repeatability
of the same word or command
            Serial.println(" times, OK");
          else
          {
            int8_t confl = easyvr.getWord();
            if (confl >= 0)
              Serial.print(" times, Similar to Word ");
            else
            {
              confl = easyvr.getCommand();
              Serial.print(" times, Similar to Command ");
            }
            Serial.println(confl);
          }
        }
      }
    }
  }
  easyvr.setLevel(EasyVR::EASY); //Level of detection can be
selected as HARDER or EASY or MEDIUM
```

```
    easyvr.playSound(SND_Hello, EasyVR::VOL_FULL);
      # endif
      /* initialization done */
      xD[0]=1;
}
```

The EasyVR shield is used as a limited word handler board in this experiment. It has to be connected on top of the UNO board. The EasyVR shield pin configuration is similar to the UNO board. Therefore, to connect both of these boards one has to make sure that EasyVR shield pins coincide with the UNO board pins and push it on top. The connection of EasyVR shield on top of Arduiino UNO will appear as shown in Fig.24 and Fig.25. The EasyVR shield has an inbuild microphone provided to it. This microphone should not be removed at any point of time. The EasyVR shield consists of an external headphone jack to which s head set can be connected. The process of loading commands into the EasyVR shield is explained in detail in the Appendix A1.

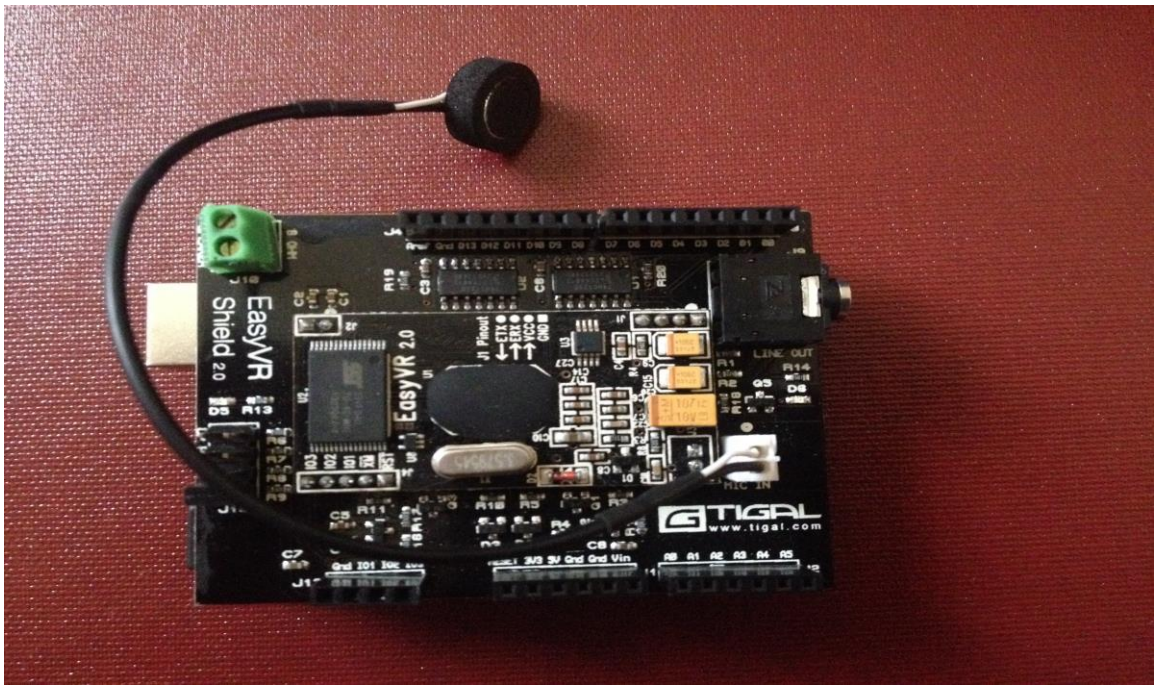


Fig.24 EasyVR shield mounted on top of Arduino™ UNO

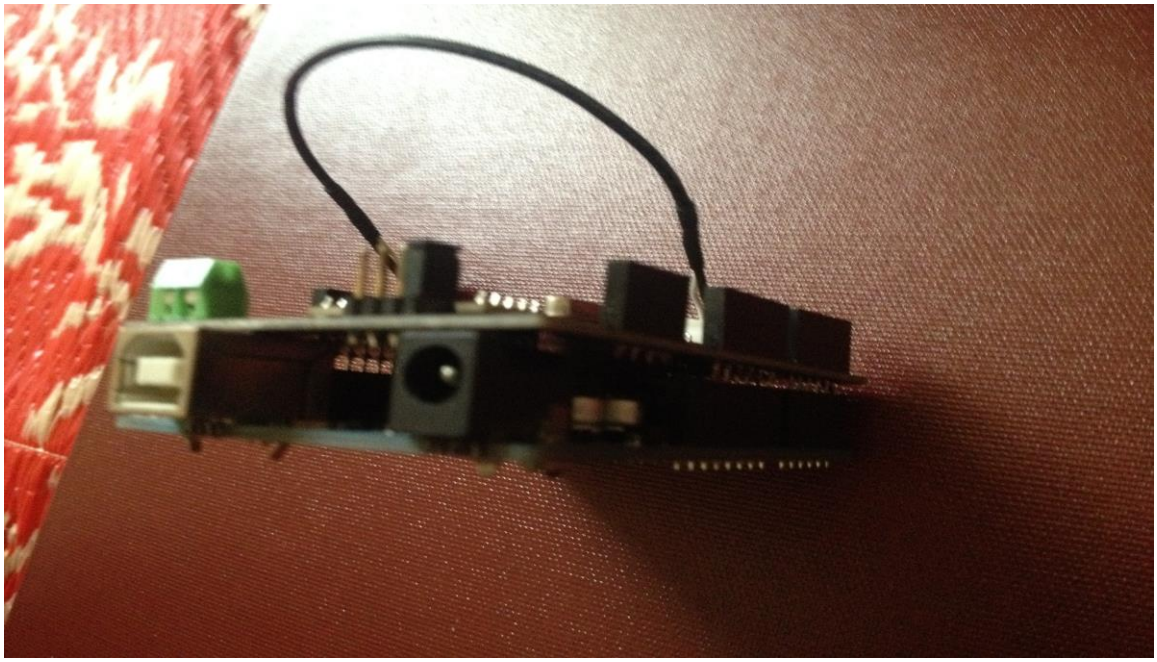


Fig.25 Side view of UNO board on bottom and EasyVR board on top

CHAPTER 4

4.1 CONCLUSION

This thesis provides a base overview of Simulink™, Arduino™ and how to integrate Arduino™ in to Simulink™. Based on these initial introductions, the author has explained a mechanism that has to be followed for creating customized Simulink™ blocks. A few generic examples of blocks are explained in detail with code segments involved and hardware connections required. The main objective of this code is to control a prosthetic arm with voice commands. The author has used a servo motor to demonstrate the control of it using the voice command. The servo motor control using the voice command is explained in detail in Section 3.6 with circuit connection and code segments. The number of components controlled by voice can be increased by adding code segments of particular component.

4.2 FUTURE WORK

Future work will focus on the LCD display block and the Voice controlled block. The LCD block can be modified to take touch input similar to smartphones. The voice controlled block's code segment can be enhanced to integrate the internet connection. The author's idea was to create not only a prosthetic arm with voice control but also an interface which can serve as a smart gadget. This smart gadget will also serve as a confidence booster or a morale lifter apart from serving as a prosthetic arm.

REFERENCES

[1] Iphone, Samsung mobiles, Android phones.

[2] Goolgle API, IOS API and Android API for speech to text and text to speech conversion.

[3a] http://www.dekaresearch.com/deka_arm.shtml as of 12/9/2014.

[3b] http://www.chalmers.se/en/news/Pages/Mind-controlled-prosthetic-arms-that-work-in-daily-life-are-now-a-reality.aspx as of 12/9/2014.

[3c] http://www.neurotechreports.com/pages/darpaprosthetics.html as of 12/9/2014.

[4] http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1455479/ as of 12/9/2014.

[5] REAL-TIME Semg-BASED FINGER JOINT ANGLE CONTROL FOR A SMART PROSTHETIC HAND, PAVAN KUMAR YARLAGADDA,ISU, MAY 2013.

[6] Digital assistants Siri, Google Now , ECHO, Cortana.

[7]http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Dev/Arduino/Shields/EasyVR_User_Manual_3.3.pdf as of 12/9/2014.

[8] CHARACTERIZATION OF MYOELECTRIC SIGNALS USING SYSTEM IDENTIFICATION TECHNIQUES. Jeffrey T. Bingham, Marco P. Schoen, 2004 ASME International Mechanical Engineering Congress
Anaheim, California, November 13-19, 2004.

[9] http://en.wikipedia.org/wiki/Simulink as of 12/9/2014.

[10] http://arduino.cc/en/Guide/Introduction as of 12/9/2014.

[11] http://www.mathworks.com/hardware-support/arduino-Simulink.html as of 12/9/2014.

[12] http://www.mathworks.com/help/Simulink/sfg/s-function-builder-dialog-box.html as of 12/9/2014.

[13] http://arduino.cc/en/Reference/pinMode as of 12/9/2014.

[14] http://www.mathworks.com/help/Simulink/sfg/s-function-builder-dialog-box.html as of 12/9/2014.

[15] http://www.dx.com/p/produino-ams1117-3-3v-sd-card-slot-reader-module-w-sd-spi-works-with-arduino-official-boards-297664#.VGwzZ8nk91U as of 12/9/2014.

[16] C:\Program Files (x86)\Arduino\examples (local copy of Arduino in computer)

APPENDIX A

A1 – Steps involved in new Voice command creation

Steps for creating and uploading the voice commands in to EasyVR shield:

i. To load the commands in to the EasyVR shield one has to first download the EasyVR commander software and it can be downloaded from the following site http://www.veear.eu/downloads/

ii. One the package was downloaded successfully it has to be installed in the computer.

iii. After the installation procedure is completed the if one double clicks on the EasyVR commander icon a GUI shown in Fig.26 will appear

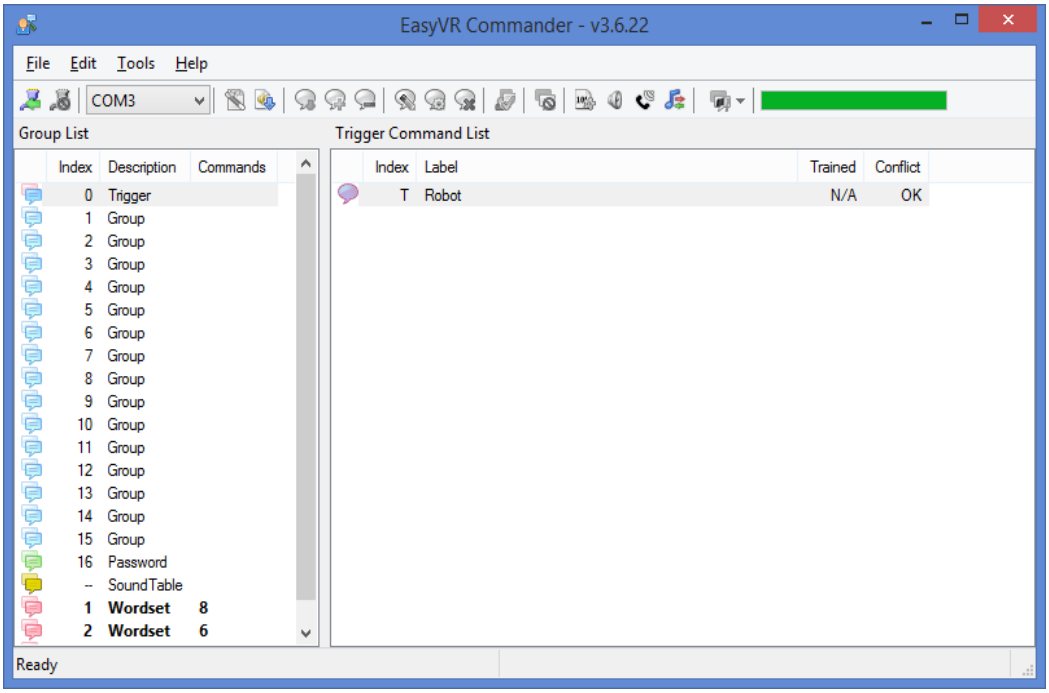


Fig.26 EasyVR commander GUI

iv. The EasyVR shield connected on top of UNO board should be connected to computer now.

v. After connecting it to the computer change the jumper pin on the J12 on the EasyVR shield to PC.

vi. After that check the com port to which the UNO board is connected to the system by opening the device manager and ports section in it.

vii. Select the same port name in the GUI of EasyVR commander placed below the Tools option.

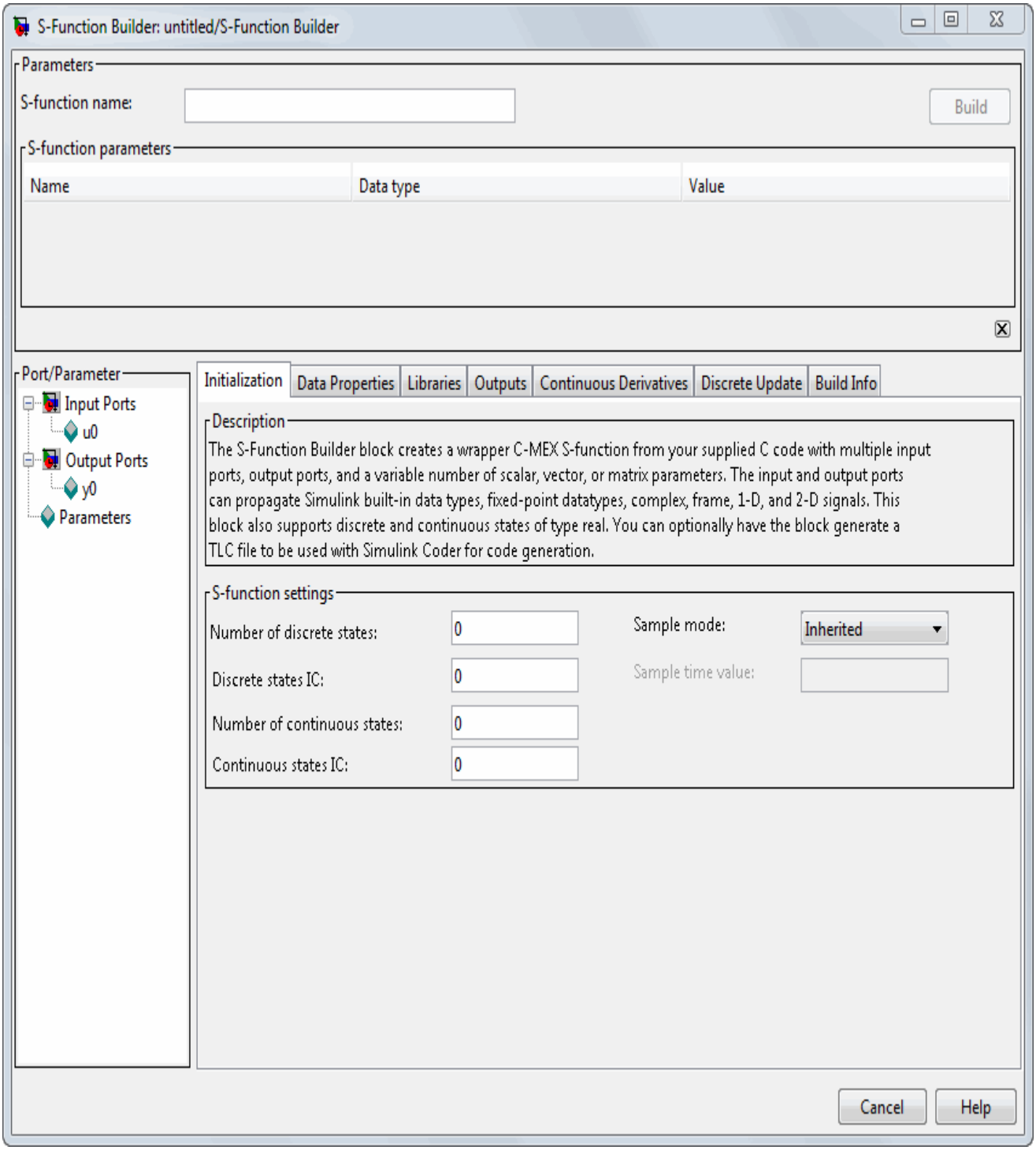
viii. After that click on Connect button below the File tab.

ix. Now a connection will form between the EasyVR and the GUI.
x. Click on the Group1 and select the option of Add command (4th icon from COM Port selected) and a new empty box will appear.
xi. In the empty box type in the command that you want to use and click enter after completing.
xii. Repeat the steps x and xi for adding more commands.
xiii. Once the commands are given the user has to train command so that system can recognize his/her voice.
xiv. To perform the training click on the command and the click on the Train command tab (3rd right to the add command).
xv. A new pop up window will appear asking one to say the command and the command should be recorded twice.
xvi. On successful recording of all the commands click on the disconnect icon placed next to connect icon.
xvii. Now the jumper on the EasyVR shield has to be removed from PC and should be placed in to the SW pin section.
xviii. Now the EasyVR shield is ready with the new set of commands.

A2 – Detailed Information on S-Function Builder contents

About S-Function Builder

The S-Function Builder dialog box enables you to specify the attributes of an S-function to be built by an S-Function Builder block. To display the dialog box, double-click the S-Function Builder block icon or select the block and then select **Open Block** from the **Edit** menu on the model editor or the block's context menu.



The dialog box contains controls that let you enter information needed for the S-Function Builder block to build an S-function to your specifications. The controls are grouped into panes. See the following sections for information on the panes and the controls that they contain.

Note The following sections use the term *target S-function* to refer to the S-

function specified by the S-Function Builder dialog box.

See Example: Modeling a Two-Input/Two-Output System for an example showing how to use the S-Function Builder to model a two-input/two-output discrete state-space system.

Parameters/S-Function Name Pane

This pane displays the target S-function name and parameters and contains the following controls.

S-function name

Specifies the name of the target S-function.

S-function parameters

This table displays the parameters of the target S-function. Each row of the table corresponds to a parameter, and each column displays a property of the parameter as follows:

- **Name** — Name of the parameter. Define and modify this property from the Parameters Pane.
- **Data type** — Lists the data type of the parameter. Define and modify this property from the Parameters Pane.
- **Value** — Specifies the value of the parameter. Enter a valid MATLAB® expression in this field.

Build/Save

Use this button to generate the C source code and executable MEX file from the information you entered in the S-Function Builder. If the button is labeled **Build**, the S-Function Builder generates the source code and executable MEX file. If the button is labeled **Save**, it generates only the C source code. Use the **Save code only** check box on the **Build Info** pane to toggle the functionality of this button.

Hide/Show S-function editing tabs

Use the small button at the bottom-right of the **Parameters/S-Function Name** pane, to collapse and expand the bottom portion of the S-Function Builder dialog box.

Port/Parameter Pane

This **Port/Parameter** pane on the left displays the ports and parameters that the dialog box specifies for the target S-function.

The pane contains a tree control whose top nodes correspond to the target S-function input ports, output ports, and parameters, respectively. Expanding the Input Ports, Output Ports, or Parameter node displays the input ports, output ports, or parameters,

respectively, specified for the target S-function. Selecting any of the port or parameter nodes selects the corresponding entry on the corresponding port or parameter specification pane.

Initialization Pane

The **Initialization** pane allows you to specify basic features of the S-function, such as the width of its input and output ports and its sample time.

The S-Function Builder uses the information that you enter on this pane to generate the `mdlInitializeSizes` callback method. The Simulink® engine invokes this method during the model initialization phase of the simulation to obtain basic information about the S-function. (See Simulink Engine Interaction with C S-Functions for more information on the model initialization phase.)

The **Initialization** pane contains the following fields.

Number of discrete states

Number of discrete states in the S-function.

Discrete states IC

Initial conditions of the discrete states in the S-function. You can enter the values as a comma-separated list or as a vector (e.g., `[0 1 2]`). The number of initial conditions must equal the number of discrete states.

Number of continuous states

Number of continuous states in the S-function.

Continuous states IC

Initial conditions of the continuous states in the S-function. You can enter the values as a comma-separated list or as a vector (e.g., `[0 1 2]`). The number of initial conditions must equal the number of continuous states.

Sample mode

Sample mode of the S-function. The sample mode determines the length of the interval between the times when the S-function updates its output. You can select one of the following options:

- `Inherited`

  The S-function inherits its sample time from the block connected to its input port.

- `Continuous`

  The block updates its outputs at each simulation step.

- `Discrete`

  The S-function updates its outputs at the rate specified in the **Sample time value** field of the S-Function Builder dialog box.

Sample time value

Scalar value indicating the interval between updates of the S-function outputs. This field is enabled only if you select `Discrete` as the **Sample mode**.

> **Note:** The S-Function Builder does not currently support multiple-block sample times or a nonzero offset time.

Data Properties Pane

The **Data Properties** pane allows you to add ports and parameters to your S-function. The column of buttons to the left of the panes allows you to add, delete, or reorder ports or parameters, depending on the currently selected pane.

- To add a port or a parameter, click the **Add** button.
- To delete the currently selected port or parameter, click the **Delete** button.
- To move the currently selected port or parameter up one position in the corresponding S-Function port or parameter list, click the **Up** button.
- To move the currently selected port or parameter down one position in the corresponding S-function port or parameter list, click the **Down** button.

This pane also contains tabbed panes that enable you to specify the attributes of the ports and parameters that you create. See the following topics for more information.

- Input Ports Pane
- Output Ports Pane
- Parameters Pane
- Data Type Attributes Pane

Input Ports Pane

The **Input Ports** pane allows you to inspect and modify the properties of the S-function input ports. The pane comprises an editable table that lists the properties of the input ports in the order in which the ports appear on the S-Function Builder block. Each row of the table corresponds to a port. Each entry in the row displays a property of the port as follows.

Port name

Name of the port. Edit this field to change the port name.

Dimensions

Lists the number of dimensions of the input signal accepted by the port. To display a list of supported dimensions, click the adjacent button. To change the port dimensionality, select a new value from the list. Specify `1-D` to size the signal dynamically, regardless of the actual dimensionality of the signal.

Rows

Specifies the size of the first (or only) dimension of the input signal. Specify `-1` to size the signal dynamically.

Columns

Specifies the size of the second dimension of the input signal (only if the input port accepts 2-D signals).

> **Note:** For input signals with two dimensions, if the rows dimension is dynamically sized, the columns dimension must also be dynamically sized or set to 1. If the columns dimension is set to some other value, the S-function will compile, but any simulation containing this S-function will not run due to an invalid dimension specification.

Complexity

Specifies whether the input port accepts real or complex-valued signals.

Bus

If the input signal to the S-Function Builder block is a bus, then use the drop-down menu in the `Bus` column to select '`on`'.

Bus Name

Step 2 of the Build S-Functions Automatically instructs you to create a bus object, if your input signal is a bus. In the field provided in the `Bus Name` column, enter the bus name that you defined while creating the inport bus object.

Output Ports Pane

The **Output Ports** pane allows you to inspect and modify the properties of the S-function output ports. The pane consists of a table that lists the properties of the output ports in the order in which the ports appear on the S-Function block. Each row of the table corresponds to a port. Each entry in the row displays a property of the port as follows.

Port name

Name of the port. Edit this field to change the port name.

Dimensions

Lists the number of dimensions of signals output by the port. To display a list of supported dimensions, click the adjacent button. To change the port dimensionality, select a new value from the list. Specify `1-D` to size the signal dynamically, regardless of the actual dimensionality of the signal.

Rows

Specifies the size of the first (or only) dimension of the output signal. Specify `-1` to size the signal dynamically.

Columns

Specifies the size of the second dimension of the output signal (only if the port outputs 2-D signals).

> **Note:** For output signals with two dimensions, if one of the dimensions is dynamically sized the other dimension must also be dynamically sized or set to 1. If the second dimension is set to some other value, the S-function will compile, but any simulation containing this S-function will not run due to an invalid dimension specification. In some cases, the calculations that determine the dimensions of dynamically sized output ports may be insufficient and both dimensions of the 2-D output signal may need to be hard coded.

Complexity

Specifies whether the port outputs real or complex-valued signals.

Bus

If the output signal to the S-Function Builder block is a bus, then use the drop-down menu in the `Bus` column to select '`on`'.

Bus Name

Step 2 of the Build S-Functions Automatically instructs you to create a bus object. In the field provided in the `Bus Name` column, enter the name that you defined while creating the outport bus object.

Parameters Pane

The **Parameters** pane allows you to inspect and modify the properties of the S-function parameters. The pane consists of a table that lists the properties of the S-function parameters. Each row of the table corresponds to a parameter. The order in which the parameters appear corresponds to the order in which the user must specify them in the **S-function parameters** field. Each entry in the row displays a property of the parameter as follows.

Parameter name

Name of the parameter. Edit this field to change the name.

Data type

Lists the data type of the parameter. Click the adjacent button to display a list of supported data types. To change the parameter data type, select a new type from the list.

Complexity

Specifies whether the parameter has real or complex values.

Data Type Attributes Pane

This pane allows you to specify the data type attributes of the input and output ports of the target S-function. The pane contains a table listing the data type attributes of each of the S-functions ports. You can edit only some of the fields in the table. The other fields are grayed out. Each row corresponds to a port of the target S-function. Each column specifies an attribute of the corresponding port.

Port

Name of the port. This field displays the name entered in the **Input ports** and **Output ports** panes. You cannot edit this field.

Data Type

Data type of the port. Click the adjacent button to display a list of supported data types. To change the data type, select a different data type from the list.

The remaining fields on this pane are enabled only if the **Data Type** field specifies a fixed-point data type. See Fixed-Point Data for more information.

Libraries Pane

The **Libraries** pane allows you to specify the location of external code files referenced by custom code that you enter in other panes of the S-Function Builder dialog box. It includes the following fields.

Library/Object/Source files

External library, object code, and source files referenced by custom code that you enter elsewhere on the S-Function Builder dialog box. List each file on a separate line. If the file resides in the current folder, you need specify only the file name. If the file resides in another folder, you must specify the full path of the file.

Alternatively, you can also use this field to specify search paths for libraries, object files, header files, and source files. To do this, enter the tag `LIB_PATH`, `INC_PATH`, or `SRC_PATH`, respectively, followed by the path name. You can make as many entries of this kind as you need but each must reside on a separate line.

For example, consider an S-Function Builder project that resides at `d:\matlab6p5\work` and needs to link against the following files:

- `c:\customfolder\customfunctions.lib`
- `d:\matlab7\customobjs\userfunctions.obj`
- `d:\externalsource\freesource.c`

The following entries enable the S-Function Builder to find these files:

```
SRC_PATH d:\externalsource
LIB_PATH $MATLABROOT\customobjs
LIB_PATH c:\customfolder
customfunctions.lib
userfunctions.obj
freesource.c
```

As this example illustrates, you can use `LIB_PATH` to specify both object and library file paths. You can include the library name in the `LIB_PATH` declaration, however you must place the object file name on a separate line. The tag `$MATLABROOT` indicates a path relative to the Matlab™ installation. You include multiple `LIB_PATH` entries on separate lines. The paths are searched in the order specified.

You can also enter preprocessor (`-D`) directives in this field, for example,

```
-DDEBUG
```

Each directive must reside on a separate line.

> **Note:** Do not put quotation marks around the library path, even if the path name has spaces in it. If you add quotation marks, the compiler will not find the library.

Includes

Header files containing declarations of functions, variables, and macros referenced by custom code that you enter elsewhere on the S-Function Builder dialog box. Specify each file on a separate line as `#include` statements. Use brackets to enclose the names of standard C header files (e.g., `#include <math.h>`). Use quotation marks to enclose names of custom header files (e.g., `#include "myutils.h"`). If your S-function uses custom include files that do not reside in the current folder, you must use the `INC_PATH` tag in the **Library/Object/Source files** field to set the include path for the S-Function Builder to the directories containing the include files (see Library/Object/Source files).

External function declarations

Declarations of external functions not declared in the header files listed in the **Includes** field. Put each declaration on a separate line. The S-Function Builder includes the specified declarations in the S-function source file that it generates. This allows S-function code that computes the S-function states or outputs to invoke the external functions.

Outputs Pane

Use the **Outputs** pane to enter code that computes the outputs of the S-function at each simulation time step. This pane contains the following fields.

Code description

Code for the `mdlOutputs` function that computes the output of the S-function at each time step (or sample time hit, in the case of a discrete S-function). When generating the source code for the S-function, the S-Function Builder inserts the code in this field in a wrapper function of the form

```
void sfun_Outputs_wrapper(const real_T *u,
                                                  real_T       *y,
                                                  const real_T *xD, /*
optional */
                                                  const real_T *xC, /*
optional */
                                                  const real_T  *param0, /*
optional */
                                                  int_T p_width0 /* optional
*/
                                                  real_T  *param1 /*
optional */
                                                  int_t p_width1 /* optional
*/
                                                  int_T y_width, /* optional
*/
                                                  int_T u_width) /* optional
*/
{

/* Your code inserted here */
}
```

where `sfun` is the name of the S-function. The S-Function Builder inserts a call to this wrapper function in the `mdlOutputs` callback method that it generates for your S-function. The Simulink engine invokes the `mdlOutputs` method at each simulation time step (or sample time step in the case of a discrete S-function) to compute the S-function output. The `mdlOutputs` method in turn invokes the wrapper function containing your output code. Your output code then actually computes and returns the S-function output.

The `mdlOutputs` method passes some or all of the following arguments to the outputs wrapper function.

Argument	Description
`u0, u1, ... uN`	Pointers to arrays containing the inputs to the S-function, where `N` is the number of input ports specified on the **Input ports** pane found on the **Data Properties** pane. The names of the arguments that appear in the outputs wrapper function are the same as the names found on the **Input ports** pane. The width of each array is the same as the input width specified for each input on the **Input ports** pane.

Argument	Description
	If you specified -1 as an input width, the width of the array is specified by the wrapper function's `u_width` argument (see below).
`y0, y1, ... yN`	Pointer to arrays containing the outputs of the S-function, where `N` is the number of output ports specified on the **Output ports** pane found on the **Data Properties** pane. The names of the arguments that appear in the outputs wrapper function are the same as the names found on the **Output ports** pane. The width of each array is the same as the output width specified for each output on the **Output ports** pane. If you specified -1 as the output width, the width of the array is specified by the wrapper function's `y_width` argument (see below). Use this array to pass the outputs that your code computes back to the Simulink engine.
`xD`	Pointer to an array containing the discrete states of the S-function. This argument appears only if you specified discrete states on the **Initialization** pane. At the first simulation time step, the discrete states have the initial values that you specified on the **Initialization** pane. At subsequent sample-time steps, the states are obtained from the values that the S-function computes at the preceding time step (see Discrete Update Pane for more information).
`xC`	Pointer to an array containing the continuous states of the S-function. This argument appears only if you specified continuous states on the **Initialization** pane. At the first simulation time step, the continuous states have the initial values that you specified on the **Initialization** pane. At subsequent time steps, the states are obtained by numerically integrating the derivatives of the states at the preceding time step (see Continuous Derivatives Pane for more information).
`param0`, `p_width0`, `param1`, `p_width1`, ... `paramN`, `p_widthN`	`param0`, `param1`, ...`paramN` are pointers to arrays containing the S-function parameters, where `N` is the number of parameters specified on the **Parameters** pane found on the **Data Properties** pane. `p_width0`, `p_width1`, ...`p_widthN` are the widths of the parameter arrays. If a parameter is a matrix, the width equals the product of the dimensions of the arrays. For example, the width of a 3-by-2 matrix parameter is 6. These arguments appear only if you specify parameters on the **Data Properties** pane.
`y_width`	Width of the array containing the S-function outputs. This argument appears in the generated code only if you specified -1 as the width of the S-function output. If the output is a matrix, `y_width` is the product of the dimensions of the matrix.
`u_width`	Width of the array containing the S-function inputs. This argument appears in the generated code only if you specified -1 as the width

Argument	Description
	of the S-function input. If the input is a matrix, `u_width` is the product of the dimensions of the matrix.

These arguments permit you to compute the output of the block as a function of its inputs and, optionally, its states and parameters. The code that you enter in this field can invoke external functions declared in the header files or external declarations on the **Libraries** pane. This allows you to use existing code to compute the outputs of the S-function.

Inputs are needed in the output function

Select this check box if the current values of the S-function inputs are used to compute its outputs. The Simulink engine uses this information to detect algebraic loops created by directly or indirectly connecting the S-function output to the S-function input.

Continuous Derivatives Pane

If the S-function has continuous states, use the **Continuous Derivatives** pane to enter code required to compute the state derivatives. Enter code for the `mdlDerivatives` function to compute the derivatives of the continuous states in the **Code description** field on this pane. When generating code, the S-Function Builder takes the code in this pane and inserts it in a wrapper function of the form:

```
void sfun_Derivatives_wrapper(const real_T *u,
                                                    const real_T *y,
                                                    real_T *dx,
                                                    real_T *xC,
                                                    const real_T
*param0,  /* optional */
                                                    int_T p_width0, /*
optional */
                                                    real_T  *param1,/*
optional */
                                                    int_T p_width1, /*
optional */
                                                    int_T y_width, /*
optional */
                                                    int_T u_width) /*
optional */
{

        /* Your code inserted here. */

}
```

where `sfun` is the name of the S-function. The S-Function Builder inserts a call to this wrapper function in the `mdlDerivatives` callback method that it generates for the S-function. The Simulink engine calls the `mdlDerivatives` method at the end of each time step to obtain the derivatives of the continuous states (see Simulink Engine Interaction with C S-Functions). The Simulink solver numerically integrates the derivatives to determine the continuous states at the next time step. At the next time step, the engine passes the updated states back to the `mdlOutputs` method (see Outputs Pane).

The `mdlDerivatives` callback method generated for the S-function passes the following arguments to the derivatives wrapper function:

- `u`
- `y`
- `dx`
- `xC`
- `param0, p_width0, param1, p_width1, ... paramN, p_widthN`
- `y_width`
- `u_width`

The `dx` argument is a pointer to an array whose width is the same as the number of continuous derivatives specified on the **Initialization** pane. Your code should use this array to return the values of the derivatives that it computes. See Outputs Pane for the meanings and usage of the other arguments. The arguments allow your code to compute derivatives as a function of the S-function inputs, outputs, and, optionally, parameters. Your code can invoke external functions declared on the **Libraries** pane.

Discrete Update Pane

If the S-function has discrete states, use the **Discrete Update** pane to enter code that computes at the current time step the values of the discrete states at the next time step.

Enter code for the `mdlUpdate` function to compute the values of the discrete states in the **Code description** field on this pane. When generating code, the S-Function Builder takes the code in this pane and inserts it in a wrapper function of the form

```
void sfun_Update_wrapper(const real_T *u,
                                                const real_T *y,
                                                real_T *xD,
                                                const real_T  *param0,  /*
optional */
                                                int_T p_width0, /*
optional */
                                                real_T  *param1,/*
optional */
                                                int_T p_width1, /*
optional */
                                                int_T y_width, /* optional
*/
                                                int_T u_width) /* optional
*/
{

        /* Your code inserted here. */

}
```

where `sfun` is the name of the S-function. The S-Function Builder inserts a call to this wrapper function in the `mdlUpdate` callback method that it generates for the S-function. The Simulink engine calls the `mdlUpdate` method at the end of each time step to obtain the values of the discrete states at the next time step (see Simulink Engine Interaction

with C S-Functions). At the next time step, the engine passes the updated states back to the `mdlOutputs` method (see Outputs Pane).

The `mdlUpdates` callback method generated for the S-function passes the following arguments to the updates wrapper function:

- `u`
- `y`
- `xD`
- `param0`, `p_width0`, `param1`, `p_width1`, ... `paramN`, `p_widthN`
- `y_width`
- `u_width`

See Outputs Pane for the meanings and usage of these arguments. Your code should use the `xD` (discrete states) variable to return the values of the discrete states that it computes. The arguments allow your code to compute the discrete states as functions of the S-function inputs, outputs, and, optionally, parameters. Your code can invoke external functions declared on the **Libraries** pane.

Build Info Pane

Use the **Build Info** pane to specify options for building the S-function MEX file. This pane contains the following fields.

Compilation diagnostics

Displays information as the S-Function Builder is generating the C source and executable files.

Show compile steps

Log each build step in the **Compilation diagnostics** field.

Create a debuggable MEX-File

Include debug information in the generated MEX file.

Generate wrapper TLC

Selecting this option allows you to generate a TLC file. You need to generate a TLC file if you are running your model in Rapid Accelerator mode or generating Simulink Coder™ code from your model. Also, while it is not necessary for Accelerator mode simulations, the TLC file will generate code for the S-function and thus makes your model run faster in Accelerator mode.

Save code only

Do not build a MEX file from the generated source code.

Enable access to SimStruct

Makes the `SimStruct` (`S`) accessible to the wrapper functions that S-Function Builder generates. This enables you to use the `SimStruct` macros and functions with your code in the **Outputs**, **Continuous Derivatives**, and **Discrete Updates** panes. For example, with this option enabled, you can use macros such as `ssGetT` in code that computes the S-function outputs:

```
double t = ssGetT(S);
  if(t < 2 ) {
    y0[0] = u0[0];
  } else {
    y0[0]= 0.0;
  }
```

Additional methods

Click this button to include additional TLC methods in the TLC file for your S-function. Check the methods you want to add and click the **Close** button to include the methods in your TLC file. For more information, see Block Target File Methods.

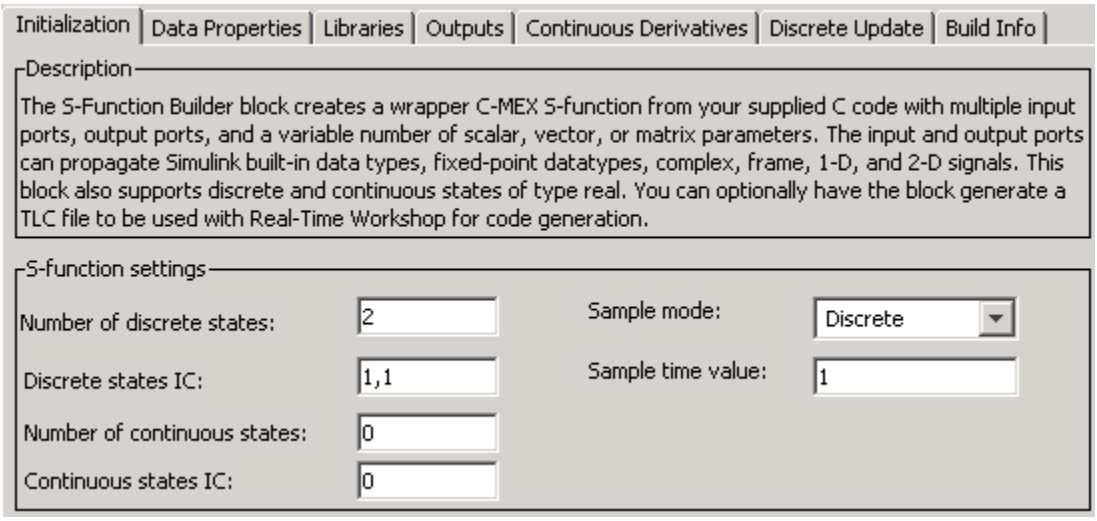
Example: Modeling a Two-Input/Two-Output System

The example `sfbuilder_example` shows how to use the S-Function Builder to model a two-input/two-output discrete state-space system with two states. In the example, the state-space matrices are parameters to the S-function and the S-function input and output are vectors. You can find a manually written version of the S-function in `dsfunc.c`.

> **Note** You need to build the S-function before running the example model. To build the S-function, double-click on the S-Function Builder block in the model and click **Build** on the S-Function Builder dialog box that opens.

Initializing S-Function Settings

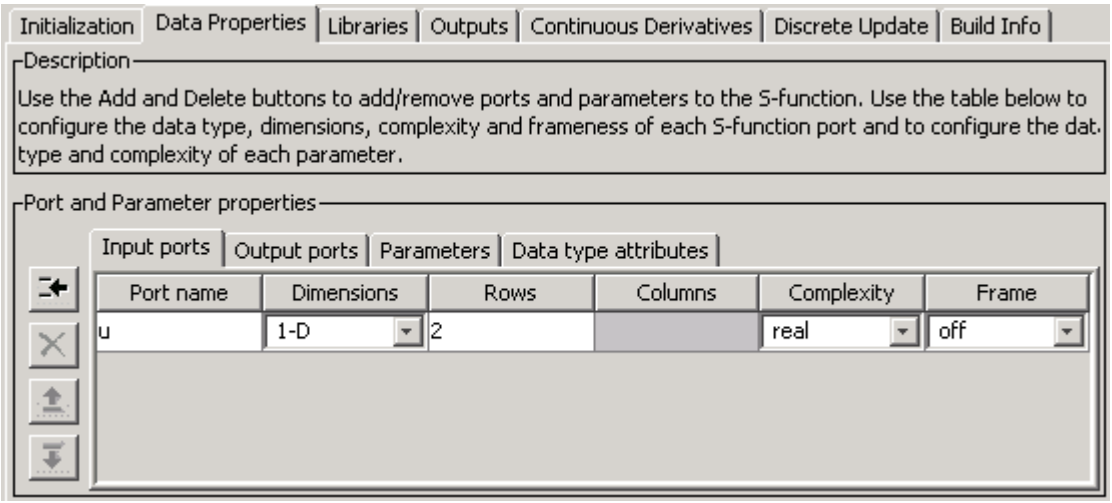
The **Initialization** pane specifies the number of discrete states and their initial conditions, as well as sets the sample time of the S-function. This example contains two discrete states, each initialized to 1, and a discrete sample mode with a sample time of 1.



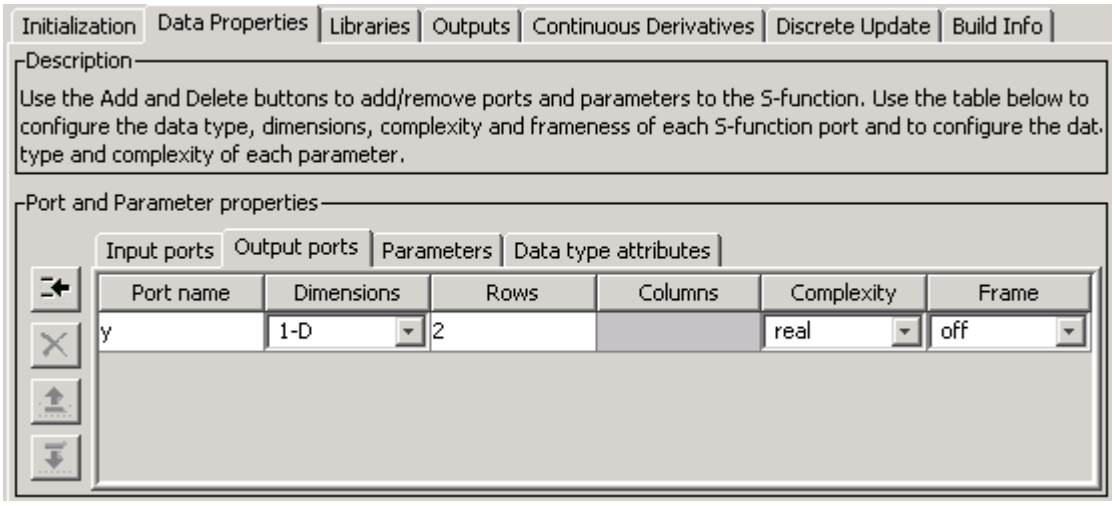
Initializing Inputs, Outputs, and Parameters

The **Data Properties** pane specifies the dimensions of the S-function input and output, as well as initializes the state-space matrices.

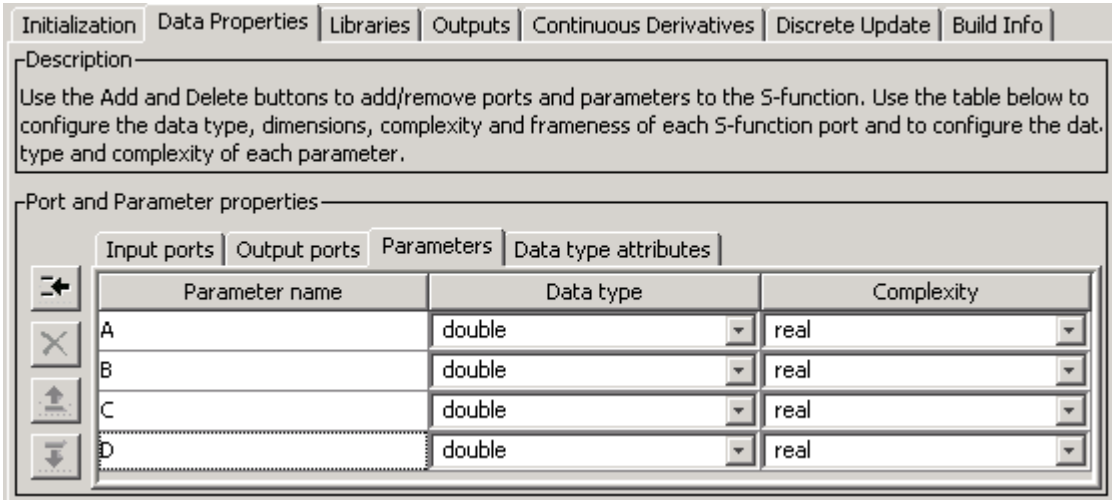
The **Input ports** pane defines the one S-function input port as a 1-D vector with two rows.



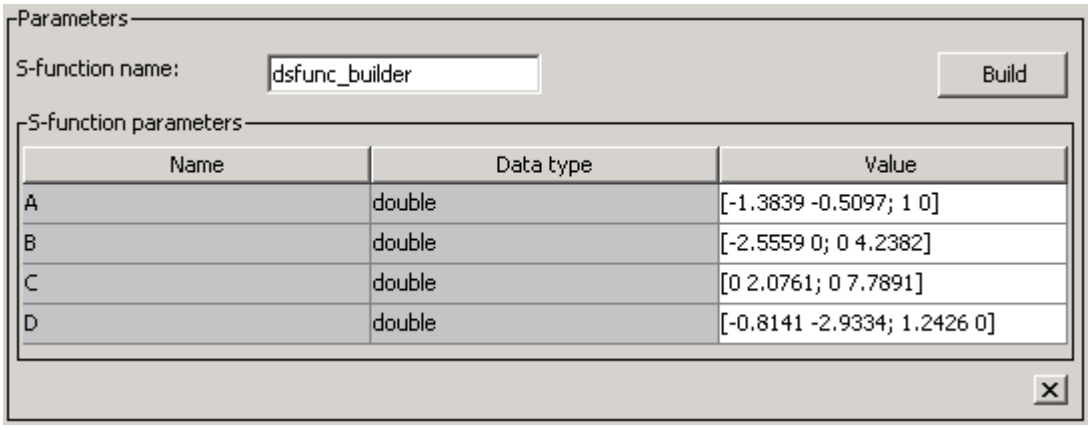
The **Output ports** pane similarly defines the one S-function output port as a 1-D vector with two rows.



The **Parameters** pane defines four parameters, one for each of the four state-space matrices.

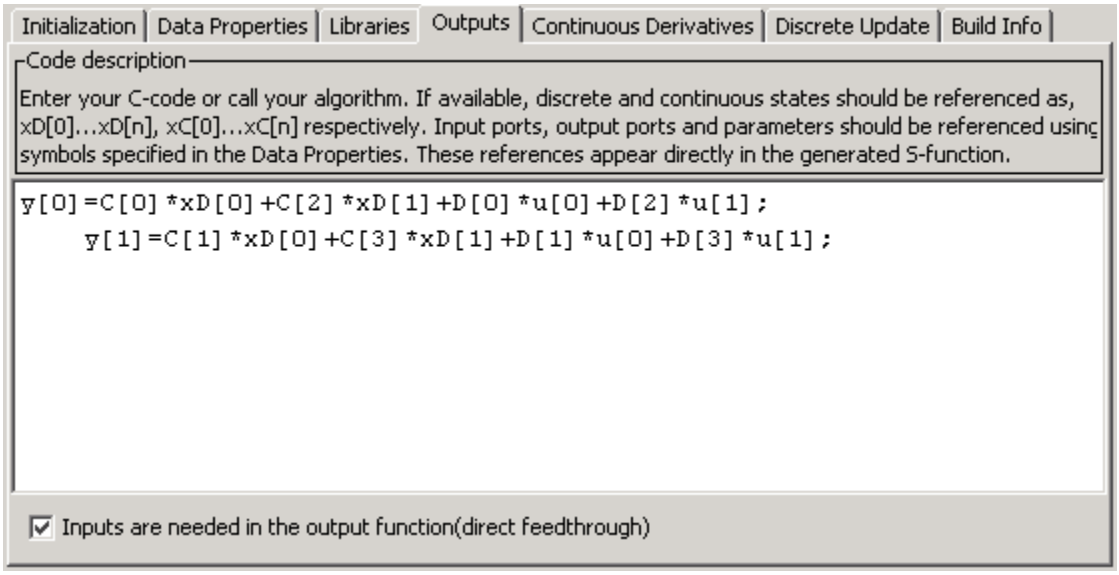


The **S-function parameters** pane at the top of the S-Function Builder contains the actual values for the state-space matrices, entered as Matlab™expressions. In this example, each state-space parameter is a two-by-two matrix. Alternatively, you can store the state-space matrices in variables in the Matlab™ workspace and enter the variable names into the **Value** field for each parameter.



Defining the Output Method

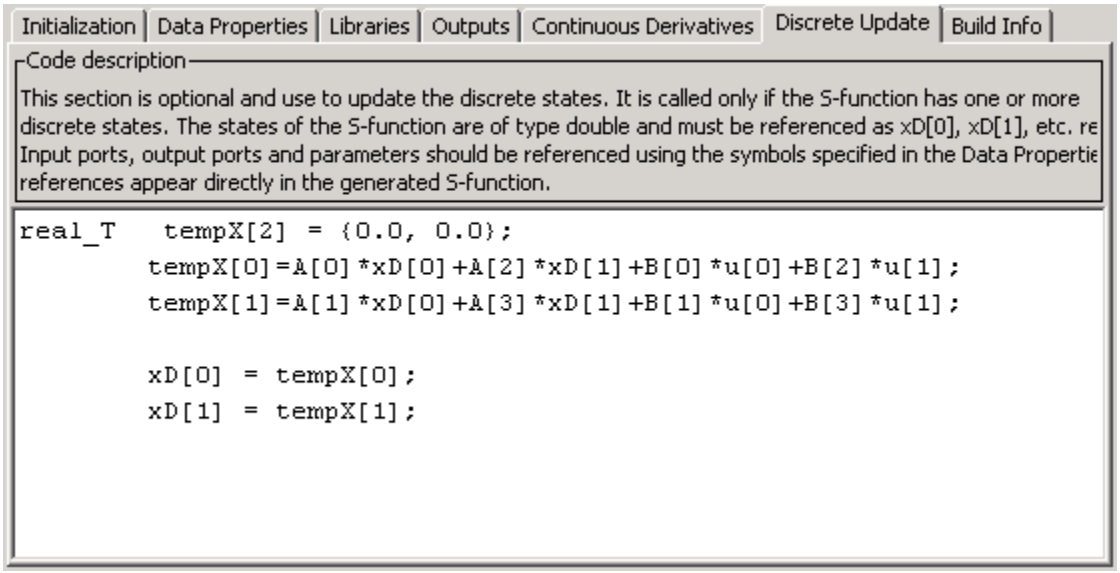
The **Outputs** pane calculates the S-function output as a function of the input and state vectors and the state-space matrices. In the outputs code, reference S-function parameters using the parameter names defined on the **Data Properties** — **Parameters** pane. Index into 2-D matrices using a scalar index, keeping in mind that S-functions use zero-based indexing. For example, to access the element `C(2,1)` in the S-function parameter `C`, use `C[1]` in the S-function code.



The **Outputs** pane also selects the **Inputs are needed in the output function (direct feedthrough)** option since this state-space model has a nonzero `D` matrix.

Defining the Discrete Update Method

The **Discrete Update** pane updates the discrete states. As with the outputs code, use the S-function parameter names and index into 2-D matrices using a scalar index, keeping in mind that S-functions use zero-based indexing. For example, to access the element `A(2,1)` in the S-function parameter `A`, use `A[1]` in the S-function code. The variable `xD` stores the final values of the discrete states.



Building the State-Space Example

Click the **Build** button on the S-Function Builder to create an executable for this S-function. You can now run the model and compare the output to the original discrete state-space S-function contained in `sfcndemo_dsfunc`.