Chapter No. 4
"Automatic Code Generation"
In this package, you will find:

- A Biography of the authors of the book
- A preview chapter from the book, Chapter NO.4 "Automatic Code Generation"
- A synopsis of the book’s content
- Information on where to buy this book

About the Authors

Dr Xuewu Dai graduated (BEng) in Electronic Engineering and received his MSc in Computer Science, both from the Southwest University, Chongqing, China, in 1999 and 2003, respectively, and completed his PhD study at the School of Electrical and Electronic Engineering, University of Manchester, in 2008. He joined the School of Electronic and Information Engineering, Southwest University, as a Lecturer Assistant in 2002 and did research projects at University College London and University of Oxford.

As a researcher and R&D engineer in signal processing and dynamic system modeling, he has over 10 years’ experience in MATLAB/Simulink simulation and embedded software development. More recently, he has been actively involved in wireless sensor actuator networks for various research and industrial projects (such as condition monitoring of aircraft engines, buildings, DFIG wind generators, CAN field-bus for SCADA, and optic sensors for water quality monitoring).

I would like to thank my wife Liping and my parents for their love, and allowing me to realize my own potential. I would like to thank Joel Goveya and Mohammed Fahad at Packt Publishing for their guidance throughout this process, and Amit Karpe and Ezequiel Aceto for their reviews. Finally, I would also like to acknowledge the partial financial support from the NSFC under grant 61101135.

For More Information:
Dr Fei Qin is currently an Assistant Professor in the Department of Electronic and Communications, University of Chinese Academy of Science, Beijing, China. He received his PhD degree from University College London, UK, in 2012. Prior to the start of his PhD, he was working for Crossbow Technology, Beijing Rep. Office as a Sr Application Engineer.

He has been working on the development of embedded systems for many different products and applications for almost ten years, including wireless network, sensor, and radar systems.

I would like to thank Dr Han, Zhengjun for his kindest advice on the motion detect algorithm, Joel Goveya and Mohammed Fahad at Packt Publishing for their support throughout this process, and Amit Karpe and Ezequiel Aceto for their reviews.
Rapid BeagleBoard Prototyping with MATLAB and Simulink

The world of embedded system development has been evolving over the past few years with many emerging technologies in both the hardware and software fields. As an open source embedded single-board computer with many standard interfaces, BeagleBoard is an ideal embedded system development kit. Integrated with both an ARM 1GHz microprocessor and an TI's C6000 DSP processor, BeagleBoard's hardware is powerful enough to meet most demands of audio/video processing. The challenge now is how to design and implement a good digital processing algorithm on BeagleBoard quickly and easily, without intensive low-level coding. This book addresses this with the rapid prototyping tools of MATLAB/Simulink, including automatic embedded code generation and visual programming techniques, in a friendly Windows development environment.

This book is a hands-on guide for those interested in learning more about MATLAB/Simulink rapid prototyping techniques and practicing these techniques on a BeagleBoard.

Combing the power of BeagleBoard and MATLAB/Simulink, this book will walk you through a number of step-by-step exercises to give you a good grounding in rapid prototyping, and help you build your audio/video applications on a BeagleBoard. You can enjoy your ideas and algorithm development and let the big dog run your fancy inspiration.

What This Book Covers

Chapter 1, Introducing BeagleBoard, starts with an introduction to the BeagleBoard, followed by the concept of BeagleBoard-based rapid prototyping with MATLAB/Simulink. By the end of this chapter, the user will have a clear idea about the BeagleBoard and rapid prototyping on the Microsoft Windows platform.

Chapter 2, Installing Linux on the BeagleBoard, serves as a quick installation reference for new users and will look at setting up the BeagleBoard for rapid prototyping. We will then set up the development environment at a Windows 7 PC by installing some software and tools. Finally, we will connect the hardware and configure the BeagleBoard for rapid prototyping. By the end of this chapter, we will be ready to get started with our rapid prototyping and developing our applications.

For More Information:
Chapter 3, C/C++ Development with Eclipse on Windows, covers how to build our first program, a simple Hello World, at the Windows 7 host PC, and run it on the Linux BeagleBoard. We will be installing Eclipse Integrated Development Environment (IDE) and CodeBench Lite, a pure Windows-based cross-platform compiler on a Windows 7 host PC. Compared with setting up a cross-platform toolchain on a Linux host PC, the installation of a Windows-based toolchain is relatively straightforward, in which the configuration and path variable management are simplified with one click. Upon completion of this chapter, we will have a fully functional cross-development environment on Windows 7 and have a taste of the work flow of cross-platform embedded system development.

Chapter 4, Automatic Code Generation, looks at automatic code generation by MATLAB/ Simulink for rapid prototyping. In this chapter, instead of typing lines of C code manually, we will develop our applications either in high-level MATLAB scripts or in a Simulink graphical programming environment, and generate executable standalone applications for the BeagleBoard. Two projects will be demonstrated: a program for average operation and a music player, where the techniques of tuning parameters on the fly will be used for performance optimization.

Chapter 5, Digital I/O and Serial Communication, discusses how to utilize digital I/O and serial communication ports on the BeagleBoard to drive external sensors, for example, an IR sensor for motion detection in smart home applications. We will talk in detail about voltage shifting, digital I/O operation, serial communication, and the data processing of motion detection via an IR sensor.

Chapter 6, Voice Recognition, demonstrates the rapid prototyping of a voice recognition system on a BeagleBoard, including RMS voice detection, feature extraction, pattern matching, and decision making. The demo is designed to operate for single users, and may be used in multiuser applications. The program in the demo can be further extended for various applications, such as voice control and speaker authorization in smart home applications.

Chapter 7, Digital Video-Based Motion Detector, is a step-by-step tutorial on converting a low-cost USB web-camera and a BeagleBoard into a motion detector. We will move into the rapid prototyping of video processing and develop an algorithm for capturing video frames and tracking of a moving object in the sequence of video frames. With the video processing algorithm being validated, you will be able to easily integrate it into your own embedded system and build various advanced applications, such as home security, vision analysis, and traffic monitoring.

Appendix, Wrapping Up, reviews what we have learned and looks at other interesting projects and techniques that you may use to prototype your own BeagleBoard project. It also provides a collection of links pointing you towards the resources used in this book and other information and projects you may be interested in.

For More Information:
Automatic Code Generation

Throughout the previous chapters, we have set up our BeagleBoard and the cross-platform development environment at our Windows 7 PC.

In this chapter, we will look at automatic code generation by MATLAB/Simulink for rapid prototyping. Automatic code generation and visual programming are fundamentally changing the way engineers work. In this chapter, instead of typing lines of C code manually, we develop our applications either in high-level scripts or in graphical programming and generate executable standalone applications.

This chapter serves as the introduction to automatic C code generation, graphical programming, and modular rapid prototyping. The following two approaches of automatic code generation will be explored:

1. We will first demonstrate how to generate C code and BeagleBoard executables from MATLAB scripts.
2. Then we will develop a program for playing music in Simulink by graphically manipulating function blocks rather than coding textually. And then run the executables as standalone applications on the BeagleBoard.

Through these two projects, you will experience the power of MATLAB and Simulink in terms of automatic code generation and graphical programming.

MATLAB code generation

First up, we are going to look at a simple MATLAB program calculating mean values of 10 numbers. Then we will set a code generation project in MATLAB to generate C code from the MATLAB program and compile the C code in Eclipse.

For More Information:
MATLAB and m-language

MATLAB provides a very powerful high-level programming language, known as m-language or m-script, for numerical computation and data processing. We are going to go through the following steps and quickly take a look at a very simple program written in the MATLAB m-language:

1. Launch MATLAB and create a folder (for example, C:\myMatlab) as your working folder to save your MATLAB code and project files.

2. Navigate to your working folder that we just created, through File | New | Function to create an empty MATLAB script file calcavg.m. Type in the following code in the text editor and save it to calcavg.m:

   function avg = calcavg(a) %#codegen
   avg=mean(a);

   Here, variable a is the input argument that indeed is an array storing 10 numbers; variable avg is the return value of the average of these 10 numbers. From a math standpoint, variable a is a vector of length 10 and avg is scalar. The simple example program calculates the mean values of these 10 numbers. mean() is a basic function provided in MATLAB and mean(a) returns the mean values of all the elements in vector a.

As shown in the simple example, the main advantage of MATLAB is that a data processing algorithm requiring tens or hundreds of lines of C code can be achieved with a few lines of MATLAB.

Code generation workflow in MATLAB

Once we get an idea about the applications, the first step in the code generation process is to prepare the algorithms in MATLAB. Then we test our algorithm in the MATLAB environment. If the algorithm works well, we generate C/C++ code from the m-scripts by making use of the MATLAB Coder toolbox. Then we compile the C code in Eclipse to get the executable and test it as a standalone application on the BeagleBoard. If we are satisfied with its performance on the BeagleBoard, we can deploy the software. If we are not satisfied, we go back to the m-scripts to improve the algorithm's performance.

For More Information:

Selecting a compiler for MATLAB Coder

MATLAB Coder requires a compiler for code generation. Various compilers can be used by MATLAB Coder and, by default, an LCC compiler comes with MATLAB. However, when we first use MATLAB Coder to generate code, we have to explicitly tell the MATLAB Coder which compiler is to be used for code generation.

If you have not already set up a compiler in MATLAB by `mex -setup` command, you can type command `mex -setup` in the MATLAB command window to select the default LCC compiler for MATLAB Coder.

When prompted, type `y` to confirm that you are going to find a compiler. And all the compilers available on your host PC will be listed with a number ID.

Then select the LCC-win32 compiler by entering its number ID. For example, type `1` followed by pressing the `Enter` key. And type `y` to confirm your selection.

When the configuration of the compiler is finished, some information will be shown in the command window to suggest a successful selection.

C/C++ code generation with MATLAB Coder

The tool we are going to use for code generation is called MATLAB Coder. To generate C code from function `calcavg.m`, we need to create a MATLAB Coder project.

From the MATLAB main menu, navigate to File | New | Code Generation Project. A Matlab Coder Project dialog box opens: enter a name for the project (for example, we use the name `mycalcavg`), set the folder to location `C:\myMatlab`, and click on OK. As a result, MATLAB Coder creates a project file `mycalcavg.prj`. Follow these steps to generate the C code:

1. A new window MATLAB Coder: MEX Function opens. Click on Add files in the Overview tab to add the m-file (`calcavg.m`) into the project.
2. Specify the type of input argument. Because C uses static typing, MATLAB Coder has to determine the properties of all variables in the MATLAB files at compiling time. Therefore, we must specify the properties (type and size) of the function's input variables before you compile the file.

For More Information:

Automatic Code Generation

In the Overview tab, the type of the input parameter `a` was classified as undefined. Click on the gear icon next to the input parameter `a`, and select Edit Type... from the context drop-down menu. A new window opens to define the parameter type: set the Class field to double and the Size field to `1 \times 10`. Click on the gear button next to the Size field and select Mark Sizes Variable. Leave the checkbox of Complex unchecked. Click on OK to finish the parameter type definition and the Overview tab will look like the following screenshot:

![Screenshot](image.png)

In the previous example, the input variable `a` is a `1 \times 10` vector and we specify its type as double with size of `1 \times 10`.

MATLAB Coder also supports input arguments with variable size. For example, if we set the argument type to `1X:10` (please note the colon between `x` and `10`), it means the size of argument is variable with upper bound to `1 \times 10`.

Now we can generate C code by a few clicks. Follow these steps to generate code:

1. Switch to the Build tab in the MATLAB Coder Project dialog.
2. Select C/C++ Static Library from the Output type drop-down list. By default, the output file is the same as the m file, that is, `calcavg`.
3. Check the checkbox Generate code only.
4. Click on the Build button to generate the C code.

For More Information:
The **Build progress** dialog box will appear. Once it is finished, you will get a success message and a link to a code generation report. If you want, click on the link to view the report in which you can easily navigate the C files generated.

The generated C files are located in the subfolder `<WorkingFolder>\codegen\lib\<matlabfilename>` under your current folder, where `<WorkingFolder>` represents the current MATLAB working folder (that is, C:\myMatlab) and `<matlabfilename>` represents the name of the m-file for code generation (that is, mycalcavg).

The generated code include several C source and header files. In particular, the two source/header files are as follows:

- **calcavg.h/.c:** The core function of average calculation is declared in the header file and defined in `calcavg.c`.

  ```
  real_T calcavg(const real_T a[10])
  {
    real_T y;
    int32_T k;
    /* UNTITLED2 Summary of this function goes here */
    /* Detailed explanation goes here */
    y = a[0];
    for (k = 0; k < 9; k++) {
      y += a[k + 1];
    }
    return y / 10.0;
  }
  ```

  As you can see, the generated C function `calcavg()` is to calculate the average of 10 numbers, which is what we expect. The variable type `real_T` is defined as `typedef double time_T;` in file `rtwtypes.h` in the same folder.

- **calcavg_initialize.h/.c:** These two files declare and define the initialization function `void calcavg_initialize(void)`. We need to initialize something by referring to the function `calcavg_initialize()` before we make the first call to `calcavg()`.

### Creating BeagleBoard applications in Eclipse

In most cases, the algorithm developed in MATLAB is part of a larger application and the generated C code is in the format of a static library. Once the C code is generated, we will take the generated code and integrate it with the rest of the application.
In this section, we will integrate the generated code into an Eclipse project and make use of Eclipse IDE and CodeBench Lite to create the executable for BeagleBoard.

### Creating an Eclipse project for BeagleBoard applications

We now create a new blank C/C++ project for the BeagleBoard in Eclipse and include the generated C code into the project by following the steps:

1. Create a new Eclipse C project targeting the BeagleBoard with the project name `mycalavgC`. We need an ARM Cross Target Application C/C++ project. Navigate to `File | New | C project` to open the C Project dialog. Select `ARM Cross Target Application` as the project type and `ARM Windows GCC (Sourcery G++ Lite)` for toolchains. Then click on `Finish`.

2. Configure the project `mycalavgC` for using the CodeBench Lite compiler. More details of how to create and configure an Eclipse project for BeagleBoard were discussed in the section `Create your first project in Eclipse`, in Chapter 3, C/C++ Development with Eclipse on Windows.

3. Include the generated code into the project. This is done by a link folder that links to the folder at which the C code generated by MATLAB Coder are stored. Right click on the project `mycalavgC` and navigate to `Properties | C/C++ General | Paths and Symbols`: select the `Source Location` tab, as shown in the following screenshot. Click on `Link Folder...` to add the source folder `C:\myMatlab\codegen\lib\calavg` into the Eclipse project. Once it is done, you will see a new source folder `/mycalavgC/calavg` in the tree of `Source folders on build path`, as shown in the following screenshot. The link folder `calavg` will also appear in your project explorer tab.

For More Information:

4. Create a source file (main.c) and set up the main function to kick off the application. The main function performs basic actions to take the input from the user, exercise the code generated from MATLAB, and display the results.

```c
#include <stdio.h>
#include "calcavg.h"
#include "calcavg_initialize.h"
int main(void)
{
    int i;
    double x[10]; // ten numbers
    double y; // the average of ten numbers

    calcavg_initialize();
    printf("Please enter ten numbers separated by space:\n");
    for (i=0; i<9; i++) scanf("%lf ", &x[i]);
    scanf("%lf", &x[9]);

    y=calcavg(x); //call matlab generated code to calculate avg

    printf("The numbers are: \n ");
    for (i=0; i<10; i++) printf("%f ", x[i]);
    printf("\n");
    printf("and their average is %f\n",y);
    return 0;
}
```

5. Build the project to get the executable for the BeagleBoard. From the Eclipse menu, navigate to Project | Build All (or hit the short key Ctrl + B) to build the project. You will get the executable mycalcavgC.elf at folder Debug under your project directory.

Running the executable at the BeagleBoard

We have got the executable mycalcavgC.elf, and we are going to run it on the at BeagleBoard remotely through the Remote System Explorer (RSE) in Eclipse. We need to first create a new Run configuration for our project. From the Eclipse main menu, navigate to Run | Run Configurations to open a dialog window, where we set the configuration to run mycalcavgC.elf. The configuration tells Eclipse how to run the application on the BeagleBoard. See the following screenshot.

For More Information:  
**Automatic Code Generation**

We first select the type of configuration as **C/C++ Remote Application**, and then click on the new button to create a new configuration. We can change the name of the configuration and select the connection to **myBBxM-01_in_RSE**, which is a remote connection we created during the installation of RSE. Click on **Browse** to find the project *mcalcavgC* and tell Eclipse that the C/C++ application we want to run is *mcalcavgC.elf*, located at the project's Debug subfolder as shown in the following screenshot:

We also need to specify which executable we want to run remotely on the BeagleBoard. By default, we set it to */home/ubuntu/mcalcavgC.elf*. As a result, the output file of the compiler *Debug/mcalcavgC.elf* on the host PC will be first downloaded to */home/ubuntu/mcalcavgC.elf*. The Linux command *chmod a+x mcalcavgC.elf* will be executed remotely to give users the right to run *mcalcavgC.elf*. It is common to uncheck the checkbox of **Skip download to target path**. Check it unless you want to update the executable on the BeagleBoard.

Finally, click on the **Apply** button to make the configuration effective, and then click on the **Run** button to run the application on the BeagleBoard. You will see the output at the remote **Console** tab in the lower panel of Eclipse as shown in the following screenshot:

For More Information:

Congratulations! You have written a data processing algorithm in MATLAB, generated C code, integrated the automatically generated code into an Eclipse project, and run it as a standalone application on the BeagleBoard.

For experienced users, you may want to log into the BeagleBoard through PuTTY and you will see is been an executable file in the Ubuntu home folder. Simply type the command `./mycalcavgC.elf` at the PuTTY terminal window to run the application.

**Simulink code generation**

Simulink, a graphical tool for modeling and simulating systems, provides an interactive graphical programming environment and enables rapid prototyping to explore design ideas at an early stage. An attracting feature of Simulink is that you can tune parameters of your design "on the fly" and immediately see what happens on the BeagleBoard, for "what if" exploration.

There are two major kinds of elements in Simulink, namely function blocks and signal lines. As a graphical programming language, a program is presented as a block diagram (referred to as a Simulink model) where blocks are connected by lines. In this section, we will build a simple program by creating a Simulink model and then run that model to play a music file (a *.wav file) on the BeagleBoard. We will also adjust the parameters of the model (in this example model, it is the volume or balance) on the fly.

For More Information:

A Simulink model of a music player

The Simulink model of the music player we are going to build is shown in the following screenshot:

The model consists of five blocks in three groups:

- Data acquisition (Read wav File block)
- Data processing (Gain and Data Type Conversion blocks)
- Data output (ALSA Audio Playback block)

The Read wav File block is a source block that opens the music file on the BeagleBoard and reads the digital audio data. The Gain block takes the digital audio data as input and modifies the data's amplitude by multiplying its gain value. Varying the gain value between \([0, 1]\), we can adjust the amplitude of the audio stream. The result of product could be a decimal fraction (analog value) and we need a Data Type Conversion block to convert it into a digital value. This block works like an analog-to-digital converter (ADC). Finally, the modified digital audio data is transferred to the ALSA Audio Playback block that drives the BeagleBoard audio hardware to make sounds. The ALSA Audio Playback block is called a sink block, as the data achieves its final destination and is converted into other forms.

From the perspective of software, the source and sink blocks are hardware drivers interfacing with the BeagleBoard hardware. The built-in BeagleBoard support package has an integrated ALSA Audio Playback block that drives the sound hardware through the open-source Advanced Linux Sound Architecture (ALSA) framework. We will develop our own device driver block of Read wav File to open and read a .wav music file.

For More Information:
Building the Simulink model

Building a model is done by bringing up a new Simulink model window and creating a block diagram through a series following these steps:

1. Drag a block from Library and place it in your model.
2. Double-click on the block to open its properties window, where the block's parameters are configured to fit our system and requirements.
3. Finally, the blocks are connected with signal/data lines to complete the model.

If you have already launched MATLAB, entering command `simulink` will start Simulink and the Simulink Library Browser window will appear.

To create a new model, you can navigate to File | New | Model, and a new window opens with an empty model. Alternatively, you can open the example model Beagleboard_Player by double-clicking on the Beagleboard_Player.mdl that can be found in this chapter's source code folder.

The ALSA Audio Playback block is in the folder of Simulink Support Package for BeagleBoard Hardware, which can be found at the bottom of the library tree, in the left panel of the Simulink Library Browser. When you place it into your model, you can double-click on it to verify its parameters.

The Gain and Data Type Conversion block are in the Math Operations subfolder under the Simulink folder in the library tree. Once you've added the Gain block into your model, double-click on it and set its field Gain to a number between [0, 1], which will be used to adjust the amplitude of the audio data.

The Read wav File block is an S-Function Builder block. This block is a so-called "device driver block" that performs specific functions (that is, opening and reading the audio file from the BeagleBoard's microSD card). The procedure of developing a custom Simulink device driver block will be discussed in detail in the following section.

Writing a Simulink device driver block for a BeagleBoard

In this section, we will make use of the S-Function Builder block, which is under User Defined Functions in the Simulink library, to generate S-Functions (system-functions). An S-Function consists of a set of sub-functions describing the behavior of a dynamic system (that is, how the system's output is changed with response to the input). These sub-functions are organized into three groups and executed in three steps: taking input, updating states, and driving outputs.

For More Information:
For the purpose of developing a BeagleBoard driver block, we want the S-Function to actually perform a reading file operation on the BeagleBoard. The output of the S-function is the audio data streaming read from the WAV file. The file name (a character string) is regarded as a parameter to the S-Function. Thus, there is no need for input signals and state variables.

To begin with, we add an S-Function Builder block to our model from the Simulink library. Double-click on the block to open the configuration window (see the following screenshot), where we will follow these steps to develop our S-function:

1. The first step is to give a name to the S-Function by typing name in the input box of S-function name. In this example, we use rdwav_SF (for read WAV S-Function).

2. The first pane is called Initialization, where we set sample mode to discrete, set sample time value to 0.1, and leave other configurations to the default value zero.

3. In the next Data Properties pane, perform the following configurations:
   - Select the Input ports tab and click on the delete button to delete the input variable, as we do not need any input to this block.
   - In the Output Ports tab, change Dimension to 2-D and set Rows to 4410, Columns to 2, Complexity to real, and Frame to on, as shown in the following screenshot.
In the Parameters tab, add one parameter with Parameter name of fileName, Data type of uint8, and Complexity of real. You will see this parameter appears just below the S-function name field, at the top-half of the window. In its field Value, type in the name of the file to be opened, for example, 'test.wav' (quoted by single quotation marks).

In the Data type attributes tab, set Data type of the output variable y0 to int16 and leave the other options at their default values.

4. In the Libraries pane, we include the source and header files used by our code. We only include three standard C libraries, as shown in the following screenshot. The preprocessor macros ifndef MATLAB_MEX_FILE specify the conditional inclusion of these C libraries. The code typed in the Libraries pane will be placed directly at the beginning of the generated C file. Thus, it is common to make use of the Library pane to declare global variables (if you need), which will be accessible from the Update and Outputs functions as shown in the following screenshot:

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Data Properties</th>
<th>Libraries</th>
<th>Outputs</th>
<th>Continuous Derivatives</th>
<th>Discrete Update</th>
<th>Build Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter any library/object or source files used by the S-function. Then, specify any necessary include files or enter the external function declarations. These functions can be called in the Outputs, Derivatives and Update methods.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Library/Object/Source files (one per line) | Include files and external function declarations |
- |
- Includes: |
  # ifndef MATLAB_MEX_FILE  
  #include <stdio.h>  
  #include <stdlib.h>  
  #include <string.h>  
  # endif  

External function declarations:  
/* extern double func(double a); */

5. In the Outputs pane, write the following code as the block's output function:

```c
/* Do nothing for mex file generation */
#ifndef MATLAB_MEX_FILE
static int firstOpenFile=1;
static FILE * pFile=NULL;
static short *dataframe;
static int nBytedataFrame;
#endif
```

For More Information:
int i, nByteRead, nSamPerFrame;
char *fName;
char wavheader[44];
int nChn, sampRate, bitPerSamp, datalen;

nSamPerFrame=4410; //One frame are 4410 samples for 0.1
//second

if (firstOpenFile==1) // open file once at
// initialization
{
    firstOpenFile=0;
    fName = (char *) malloc(p_width0+1);
    memcpy(fName,fileName,p_width0);
    fName[p_width0]=0x0;
    pFile=fopen(fName,"r");
    if (pFile==NULL){ printf("Open file %s failed.
", fName);
        return;
    }
    // read wav file information
    fread(wavheader, 1, 44, pFile);
    nChn = (short) wavheader[22]+(wavheader[23]<<8);
    sampRate=*(int *)&wavheader[24]);
    bitPerSamp = *(short *)&wavheader[34]);

    // allocate nBytedataFrame bytes memory for data reading
    nBytedataFrame=nSamPerFrame*2*sizeof(short);
    dataframe= (short *) malloc(nBytedataFrame);
}
else{ // read one frame of data from was file
    nByteRead = fread(dataframe, 1, nBytedataFrame, pFile);
    if (nByteRead != nBytedataFrame)
    { //if file ends or any errors, go back to beginning
        fseek(pFile, 44, SEEK_SET);
        fread(dataframe, 1, nBytedataFrame, pFile);
    }
    //convert row-wise array data into column-wise framed
    //data
    for (i=0;i<nSamPerFrame;i++)
    { y0[i]=dataframe[i*2];
        y0[nSamPerFrame+i]=dataframe[i*2+1];
    }
}
#endif

For More Information:
The previous code consists of two parts: initialization (opening the specified file once) and repeated operation of reading 4410 samples of audio data from the file. We use a static variable to store the file handle to avoid frequent file opening. As we define an input parameter (`fileName`, uint8-type) to our S-Function, we can refer to `fileName` to get the file name. The S-Function Builder also provides an additional but implicit input parameter `p_width0` that represents how many characters (uint8) the first input parameter `fileName` has. It is worth noting that `fileName` is an array of char, rather than a string, and does not come with the string termination indicator (0x00). We have to manually pad a 0x00 to `fileName`. Furthermore, the input `fileName` will be declared as a constant input in the generated C code; we define a new variable `char *fName` for the purpose of padding the string terminator 0x00.

In the file-reading part, once we read the desired amount of data into dataframe, it is a row-wise array in C. However, MATLAB/Simulink use column-wise arrays. We use a for-loop to convert the row-wise array into a column-wise array. Furthermore, when we reach the end of the file, we go back to the first audio data (44-byte offset from the file's beginning) and repeat reading the data again. As a result, the music will be played repeatedly. More details of the *.wav file format can be found at https://ccrma.stanford.edu/courses/422/projects/WaveFormat/.

6. The last (rightmost) pane is Build Info. Check the Generate wrapper TLC and uncheck the Enable access to SimStruct checkboxes. Enabling access to SimStruct will have the generated code include a header file SimStruct.h, which may be incompatible with your target hardware.

7. The final step is to generate C code for our block by clicking on the Build button. If everything goes well, we will get four files:
   - A wrapper file (`rdwav_SF_wrapper.c`)
   - A TLC file (`rdwav_SF.tlc`)
   - An S-Function file (`rdwav_SF.c`)
   - A MEX-file (`rdwav_SF.mexw32`)

The most important file is the wrapper file, which contains the C code that you just typed in those configuration panes.

So far, we have built our own Simulink block that works as a hardware driver reading audio data from file `test.wav`. In execution, Simulink makes repeated calls during specific stages to each block in the model. As a result, the output function we just built will be called at the specified rate (0.1 second) to read the file and put recent audio data in the block's output. The audio file we are reading is a standard WAV file containing two channel stereo data, sampled at 44100 Hz with 16-bit per sample.
As we set our block's simulation interval to 0.1 second, every calling to our output function will acquire a frame of data containing the audio of 0.1 second, which are 4410 samples for each channel. More specifically, the output is a 4410 x 2 matrix (referred to as framed data) and each element of the matrix is of \texttt{int16} type corresponding to 16-bit per sample.

**Configuring the model to run on a BeagleBoard**

We have developed our Simulink model of a music player. Now we will configure the model to run on a BeagleBoard. Before we move on to the next step, it is worth checking that our host PC and BeagleBoard are connected via Ethernet. This can be done by launching the PuTTY utility and seeing if you can log into the BeagleBoard. You may press and release the \texttt{USER} button on the BeagleBoard to hear the BeagleBoard hardware speak its IP address.

From the main menu of our Simulink model window, navigate to \texttt{Tools | Run on Target hardware | Prepare to Run}. The target hardware information will be automatically populated. Check the \texttt{Enable External mode} option and leave the \texttt{TCP/IP port} parameter at its default value 17725. Your host PC will use this port to communicate with the BeagleBoard. Verify the IP address, username, and password. Then click on \texttt{OK}.

In the toolbar of the Simulink model window, set the simulation stop time to \texttt{inf} to run our model until we click on the pause or stop button. Save the model to finish the configuration.

For your convenience, a pre-configured model \texttt{Beagleboard_Player.mdl} is enclosed in this chapter's source code folder. It can be used as a template for future development.

**Running the music player on the BeagleBoard**

Before we start running our model, we need to put a WAV audio file named \texttt{test.wav} in the home directory (/home/ubuntu) on the BeagleBoard. An example audio file \texttt{test.wav} is included in this chapter's source code folder for convenience. You can download this file to the BeagleBoard easily through either the WinSCP or the RSE in Eclipse.

Now, navigate to \texttt{Tools | Run on Target Hardware | Run} to run the model on the BeagleBoard. It may take some time for the Simulink engine to prepare, compile, and download the model to the BeagleBoard. Once done, a system command window will open that shows the messages coming from our model running on the BeagleBoard. Please note that we have to explicitly select \texttt{Run on Target Hardware} to have our model run on the BeagleBoard.

For More Information:

You should now hear the music, which will be repeatedly played by the BeagleBoard. If you want to stop the music, click on the stop button (a black square).

**Playing music without Simulink**

When logging into the BeagleBoard at a PuTTY terminal or RSE terminal, you will find the executable file of your Simulink model at `/home/ubuntu/$ModelName_rtt/MW/$ModelName`, where `$ModelName` represents the main name of your `.mdl` file, for example, `/home/ubuntu/Beagleboard_Player_rtt/MW/Beagleboard_Player`. In the home folder (`/home/ubuntu`) where you store the music file, enter the command `.Beagleboard_Player_rtt/MW/Beagleboard_Player`, and you will play the music as well.

**Tuning model parameters on the fly**

In the model we just developed, we reduce the amplitude of audio data by an attenuation rate of 50 percent, which is regarded as a parameter by our model. When the model is running on the BeagleBoard hardware, the volume of music is fixed to half of the full scale. We may want to update the attenuation rate on the fly, so that we can change the volume when we are hearing the music. To accomplish this, follow these steps:

1. We can declare a variable (that is, `attRatio`) in the MATLAB base workspace to represent the attenuation rate and assign a value, say 0.1, to it. The command is `attRatio=0.1`.

2. In the Simulink model `Beagleboard_Player`, double-click on the `Gain` block and set the `gain` field to `attRatio`. This is to tell the Simulink block to take the value of the variable `attRatio` as its gain. When we assign different values to `attRatio` at the MATLAB command line, the gain of the `Gain` block will change and, in turn, the attenuation rate of audio data is changed. However, this treatment takes effect only when we start running the model. Every time we change the value of `attRatio`, we have to restart the model.

3. In order to tune the attenuation ratio on the fly without rebooting, once we change the value of `attRatio` at the MATLAB base workspace, we make a call to the MATLAB function `set_param()` in the MATLAB command window:

   ```matlab
   set_param(bdroot,'SimulationCommand','update');
   ```

   where `bdroot` is an object representing the Simulink model that is currently running on the BeagleBoard. This command is to update the Simulink model (that is our `Beagleboard_Player` running on the target hardware) according to the current values of variables in MATLAB’s base workspace.
A pre-configured model (Beagleboard_Player2.mdl) with the capability of tuning parameters on the fly is provided in this chapter's source code folder for your convenience.

**Tuning model parameters through GUIs**

You may also want to tune the parameters through a friendly graphic interface. MATLAB also provides utilities to develop custom **graphic user interfaces** (GUIs), eliminating the need to use extensive commands in the command line.

In this chapter's source code folder, you can find a Simulink model (Beagleboard_Player3.mdl) with associated GUIs (balanceCtr.fig, balanceCtr.m). When the Beagleboard_Player3 model starts on the BeagleBoard, the GUI balanceCtrl will start automatically, in which you can adjust the stereo's balance on the fly by varying the slider bars to change the attenuations of left and right channels individually, as shown in the following screenshot:

To understand how this works, open the example model file and navigate to **File | Model Properties** to open a dialog window. Select the **Callbacks** tab: under **InitFcn**, we use two commands, \( vL=0.05; \ vR=0.05; \) to set initial values to variables \( vL \) and \( vR \), representing the attenuation ratio of the left and right channels, respectively. Under **StartFcn**, we invoke balanceCtrl_GUI to start the GUI window.

**Other things to try**

In order to explore the advances of rapid prototyping by Simulink graphical programming and parameter tuning on the fly, a revised parametric audio equalizer (Beagleboard_PlayerEqualizer.mdl) is included in the chapter's source folder. In this model, while you are listening to the music from the speaker connected to your BeagleBoard, you can adjust the parameters of the three band filters in the model and see what will happen immediately to the music you are hearing. You can easily enhance the bass components and reduce the high frequency parts, or vice versa.

For More Information:  
Summary

This chapter showed two automatic code generation approaches: generating code from m script with MATLAB Coder and generating code from a Simulink model with the BeagleBoard built-in support in Simulink. A demo of a data-processing algorithm (averaging) is implemented in m script. Several audio processing demos demonstrate the advantages of MATLAB/Simulink-based rapid prototyping. In particular, we have learned how to build a custom hardware driver block for BeagleBoard.

We can move onto our first MATLAB/Simulink project for a more practical project, the rapid prototyping of a motion detector for a smart home.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.