XAPP948 (v1.0) December 5, 2006

Hardware Acceleration of 3GPP Turbo Encoder/Decoder BER Measurements Using System Generator

Author: David Lawrie

Summary

Determining the bit error rate (BER) performance of modern high performance forward error correction (FEC) algorithms, such as the Turbo Encoder/Decoder, can be a time consuming process. In some cases, software simulations are so time consuming that they might not be practical.

This application note describes a system for accelerating BER measurements providing:

- Hardware execution speeds
- Reduced run time
- Increased accuracy
- Automated generation of results

The application note describes how a hardware implementation of the Xilinx 3GPP Turbo Encoder and Decoder cores is incorporated into a System Generator design to provide BER performance measurements. Within the hardware design, a Noisy Channel model is used to test the encoder/decoder combination under a variety of Additive White Gaussian Noise (AWGN) conditions. A combination of Simulink blocks and MATLAB scripts is used to control the hardware and collect and display the results with a minimum of user interaction.

The ability to execute BER performance measurements within hardware can reduce many hours (perhaps even days) of software simulations to a few minutes of hardware execution. Significantly larger amount of data can be passed through the system with this approach, thus, increasing the accuracy of the final results.

The simple control scripts provide a mechanism for the user to control all aspects of the system, including the capability to automatically measure BER performance with many different combinations of parameters.

This application note describes these concepts and provides a complete set of files so that the full power of this approach can be demonstrated.

Additional Reading

It is recommended that the reader be familiar with data sheets for the Xilinx Turbo Encoder and Decoder cores before proceeding with this application note. The two data sheet files are included within the zip file supplied for this application note:

- `tcc_encoder_3gpp_v2_0` - the data sheet for the Turbo Encoder V2_0.
- `tcc_decoder_3gpp_v1_0` - the data sheet for the Turbo Decoder V1_0.

A familiarity with System Generator and Xilinx Tools is also assumed. Further information can be found from the software manuals on the Xilinx website:

- http://www.xilinx.com/support/software_manuals.htm
- http://www.xilinx.com/ise/optional_prod/system_generator.htm

Introduction to BER Testing

The concept of BER testing is quite simple:

- Create a random set of data
- Encode the data
- Add random errors to the data to simulate a noisy channel
- Decode the data
- Measure the number of errors between the original and decoded data

Figure 1 shows this process in block diagram form. Running this entire process as a software simulation can be extremely time consuming. Therefore, the main goal here is to create hardware versions of each of these blocks to enable the system to be run at hardware speeds.

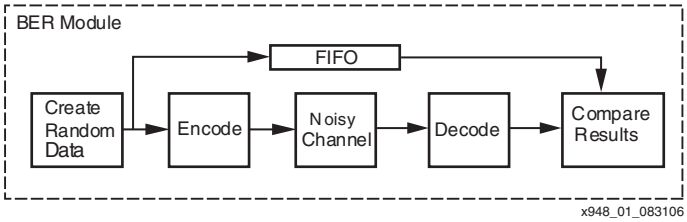

Figure 1: **BER Testing Block Diagram**

Looking at each block of Figure 1 in turn,

- The input to the encoder is a block of data a single bit wide. For the 3GPP specification, each block is between 40 and 5114 bits. A simple linear feedback shift register (LFSR) can therefore be used to create the random data input. The LFSR is shipped with the System Generator design environment.
- Xilinx has produced a 3GPP encoder core. A data sheet is available form the Xilinx website and this design uses the tcc_encoder_v2_0 core. An evaluation version of the encoder can be downloaded from the Xilinx Website, but it is also included within the ZIP file associated with this application note.
- An AWGN noise source has been produced from a hierarchy of System generator blocks to create the Noisy Channel.
- Xilinx has produced a 3GPP decoder core. A data sheet is available from the Xilinx Website and this design uses the tcc_decoder_v1_0 core. An evaluation version of this core can be downloaded from the Xilinx Website, but it is also included within the ZIP file associated with this application note.
- A FIFO memory is shipped with the System Generator design environment.
- The Compare Results block has been developed for this application. This block is made up of the various logic blocks available within System Generator, e.g., AND gates, multiplexors, etc.

The function of the compare block is straightforward. Considering the data sheet for the decoder, then the decoded output is available serially when the RDY signal is High. The compare block simply waits for the RDY signal of the decoder to go High, and on each clock cycle it compares the decoded data with the original from the FIFO. A simple counter is used to count the number of errors with each block. A second counter is used to count the number of blocks (sometimes referred to as frames) that contain one or more errors. A count of the total number of bits that has been transmitted is also maintained and, hence, with these three counters the bit error rate (BER) and frame error rate (FER) can be determined.

Quick Start Overview

This section provides a fast and simple overview of the steps necessary to run the BER test system. Further expanded details can be found in later sections of this application note.

To immediately run the BER system:

1. One of the supported platform's hardware and software must be installed.
2. Within MATLAB, change to the directory where the BER design has been installed (specifically to where the `demo_ctrl.m` file has been installed)
3. Open the `demo_ctrl.m` MATLAB control script and change the platform variable to match with the supported platform. Do not change any of the other default settings yet.
4. Run the `demo_ctrl.m` script. The FPGA should be downloaded with the correct bit file and BER graphs will start to be drawn within the MATLAB environment. It can take 10's of seconds to download some hardware platforms and display the first graph, which is normal operation. As the Eb/No value increases, the time taken to produce each BER point increases as more bits are generally passed through the simulation for each BER point. Again, this is normal operation.

Following a successful run with the default parameters, the user might wish to start varying the test parameters.

This application note provides eight BER data files that have been previously generated. These files will get overwritten by using the default settings in the `demo_ctrl.m` script. The user can display certain combinations of these files before attempting to run the BER system with the `plot_script.m` script file. Simply run the `plot_script` from within MATLAB and various BER combinations will be displayed. It is useful to compare the default files with those generated by the first run of the BER system to verify the correct operation of the design. See "Displaying Output" for more details.

Parameters that do not require a different implementation of the decoder core, such as changing

- Number of iterations
- Fast termination threshold
- Block size values
- Minimum BER value
- Maximum number of bits to process to create each BER point

can be varied by changing the corresponding values within the `demo_ctrl.m script.`(See the `demo_ctrl.m` script for more details). The overall number of tests is determined by the number of values specified for each variable. For example, increasing the number of block sizes tested by a factor of two increases the number of tests and, consequently, the run time by a factor of two.

Parameters that require a different implementation of the decoder core, such as changing

- Input bit width
- Metric bit width
- Whether the core has a Fast Termination facility of not, and
- Addition of optional signals, such as SCLR, etc.,

can only be changed by regenerating a new version of the decoder core from within the Xilinx CORE Generator™. After the new core has been generated (usually in the form of an EDN or NGC file), the entire BER design must be resynthesized using the new file. For more details on this process, see "Creating a New FPGA Programming File."

Installation Tip: System Generator has a limitation of 256 characters for any file name. Therefore, do not install the software files associated with it within a deeply nested directory structure.

Introduction to System Generator Hardware in the Loop Testing

Documentation is available within the MATLAB/Simulink/System Generator environments, so it is not the intention of this document to duplicate that information here. However, it is important to understand the relationship between the hardware and software components in this application.

See Figure 2 where the block labeled BER module represents the blocks shown in Figure 1. As described in the previous section, the BER module block can be constructed from various blocks already shipped with System Generator (e.g., FIFO, logic gates, etc.) and custom blocks, such as the Xilinx Turbo encoder and decoder. Figure 2 shows that the BER module receives input from the Simulink/System Generator environment which also captures output from the system.

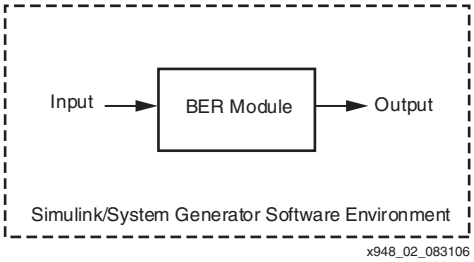

Figure 2: **MATLAB/Simulink Interfaces**

- The input takes the form of setting various control values, such as the current block size, the number of iterations, the fast termination threshold, etc. These values are all set by software in the form of MATLAB scripts from the Simulink environment.
- The output is in the form of several counter values which are read and interpreted using MATLAB scripts to calculate the BER and FER performance.

Generally, a System Generator design module can be run in two different ways:

1. As a Software Simulation. Here, simulation models of all the various design blocks are simulated in software. This approach benefits being able to test the design functionality, but it does not provide the speed of operation required for this application. Hence, true simulation models of the encoder and decoder have not been created. An alternative method of simulation would be to use a VHDL model of the encoder and decoder cores. Then, instruct System Generator to use a simulation tool (e.g., ModelSim) to simulate these cores, while using System Generator to simulate the remainder of the design. All of these approaches are valid, but they are not provided or described herein because this application note focuses on providing fast simulation speed. See the System Generator documentation for more details of these approaches. Note that any communication between the Simulink/System Generator environment and the BER module is straightforward in this case because it is all handled in software by the tools.
2. As Hardware In The Loop (HWITL). In this case, the various blocks of the BER module (Figure 1) are combined together using a synthesis tool. This process is handled automatically by the System Generator environment and a variety of synthesis tools can be specified. The final output of this process is a hardware realization of the BER module for use on a Xilinx FPGA. Communication between the hardware realization and the software control environment is now slightly more complicated. The input and output on Figure 2 occurs over a hardware interface. A variety of hardware platforms are supported in the System Generator tools and the user simply selects the appropriate hardware platform. Specific hardware modules are added by System Generator in the synthesis stage of the user design, and appropriate software drivers are used within the Simulink environment.

System Timing

Generally, FPGA designs are synchronous processes and, therefore, System Generator incorporates the concept of a clock rate. Practically, the speed of execution of a System Generator simulation (which could be considered as the software clock rate) depends on the complexity of the task, CPU performance, etc. However, when considering HWITL, and especially the transfer of data between the hardware and software environments, the concept of the System Generator software clock is useful.

Two modes of operation are available when considering HWITL:

1. Single Stepping. Here the hardware and software clocks are essentially fixed together. The software simulation runs at a maximum rate (given by complexity, etc.), and each time a new simulation clock cycle is started, the hardware is clocked on by one cycle. This mode is useful for debugging, etc., but it does not use the fact that the hardware could run at a much faster rate. It does, however, mean that data transfer between the hardware and software is straightforward because it is all handled within System Generator.
2. Free Running. This mode uses the full power of the hardware to run at a high clock rate. This hardware clock rate is orders of magnitude faster than the software simulation clock rate and, therefore, the hardware and software clocks must be considered to be asynchronous. This means that much more care has to be taken when transferring data between the software and hardware to ensure that communication is not corrupted. This application uses a simple single-bit handshaking mechanism, where all inputs and outputs are ensured to be valid before raising a control signal to indicate that data is available to read or write. System Generator 8.1 provides a *shared memory* block that can be used to easily handle large data transfers between the hardware and software domains. However, this design has been created to maximize performance and, therefore, it uses a simple handshaking mechanism; any unnecessary communication across the software/hardware domain will simply reduce performance.

Design Overview

Associated with this application note is a complete set of design files that allows the user to create and run the HWITL BER design. A brief description of the files is contained in the "Reference Design Files" section of this document. A lot of hints and tips have been added to the various design files in the form of comments; the reader is encouraged to open the design files to enhance the following description.

The most important design file is the System Generator model of the entire system as shown in screen shot of Figure 3.

This application utilizes two versions of this top-level model:

1. `tcc_3gpp_v1_0_demo_synth.mdl` - This is the synthesizable version of the design. Pushing down into the BER module reveals a hierarchy of various sub-blocks that make up the BER module design.
2. `tcc_3gpp_v1_0_demo_hwitl.mdl` - This is the post-synthesis version of the design and this is the model that can be executed in hardware. This is identical to the structure shown in Figure 3, except the BER module block has now been replaced by its hardware equivalent. Pushing down into the new module reveals a single block representing the FPGA hardware. More details are given in the "Different Hardware Options" section.

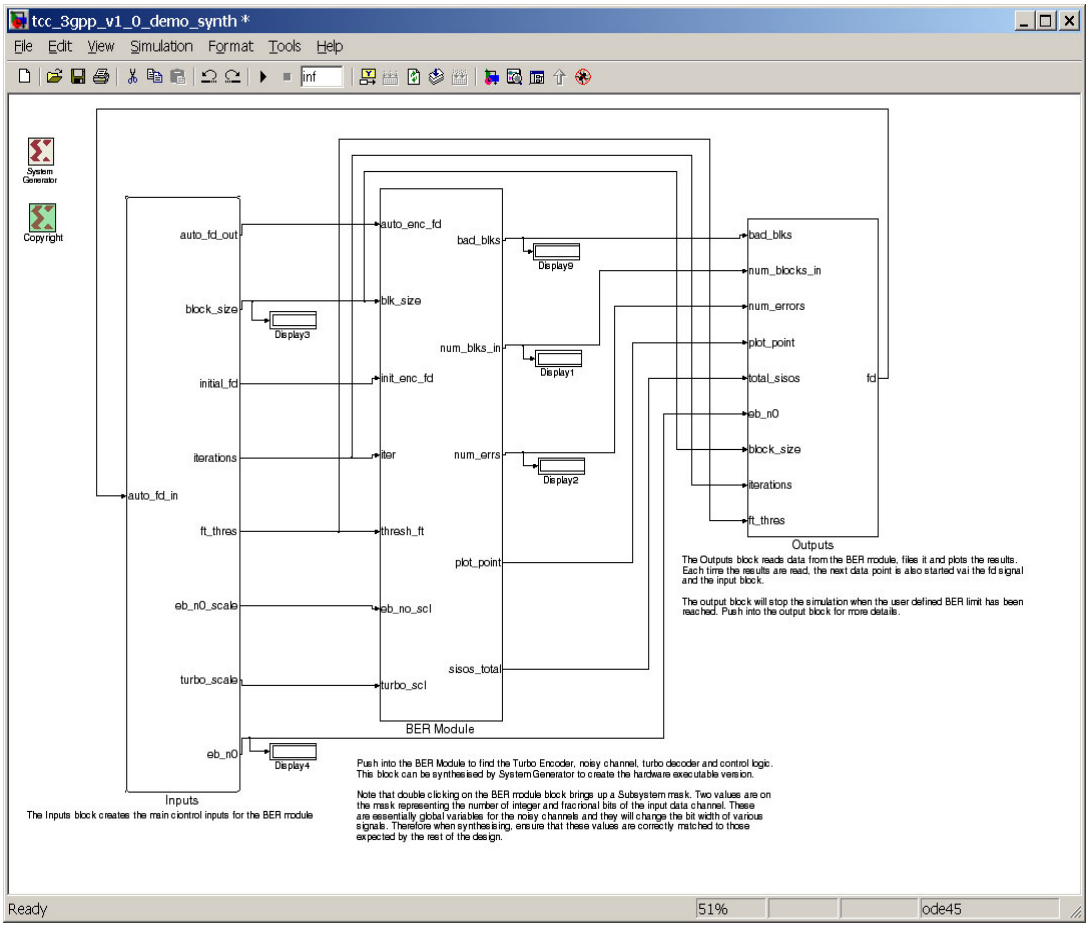

Figure 3: **Top-Level Synthesizable Design**

Figure 3 consists of three submodules:

1. **Input Module** - This module defines various system control parameters and is always executed as software within the System Generator environment.
2. **BER Module** - This module represents the design of Figure 1 and is either executed as a software simulation or as hardware. This module must be synthesized to create a hardware executable version. (Note that software simulation of the block is not the target of this application note and, therefore, it has not been implemented)
3. **Output Module**- This module graphically outputs data and stores it to a text file and is always executed as software within the System Generator Environment.

Each of the top-level signals in these modules are described below, followed by a description of how the overall design operates.

Input Module

The Input module is responsible for defining the various input control values for each test. Ports on this module include:

- AUTO_FD_OUT - This is the automatic first data (FD) signal. Each time a BER point has been completed, the Output module automatically starts data generation for the next BER point by driving this signal High. The Output module automatically stops the model if the correct conditions are met. See the "Output Module" section.

- BLOCK_SIZE - This is the block size used in the test. 3GPP block sizes are between 40 and 5114 bits. See the Xilinx Turbo encoder or decoder data sheets for more details (provided within the ZIP file associated with this application note).
- INITIAL_FD - This is the initial FD signal. This signal is generated in software and it starts the entire process. This is described later in more detail.
- ITERATIONS - This is the number of iterations to implement on the decoder.
- FT_THRES - This is the fast termination threshold value. The decoder core used in this application has been created with the fast termination option activated. See the Xilinx Decoder data sheet for more details of how this value effects operation and the maximum number of iterations performed.
- EB_N0_SCALE - This is the scale value fed to the noisy channels to create the specified Eb/No (i.e., energy per bit in relation to white noise) value. The noise created by the AWGN channels must be scaled to account for the rate of the Turbo encoder/decoder. See the Turbo decoder data sheet for more details.
- TURBO_SCALE - Ideally, the decoder must have knowledge of the amount of noise entering the system. Often this is estimated, but in this case, the noise scaling is completely known so the optimal scaling value is applied to the turbo decoder inputs. See the Turbo decoder data sheet for more details.
- EB_N0 - This is the Eb/No (i.e., energy per bit in relation to white noise) point currently being calculated.

BER Module

Because this module directly uses the signals described in the "Input Module" and "Output Module" sections, see these sections for signal descriptions.

Output Module

- BAD_BLKS - This is the number of blocks where one or more bit errors have been detected.
- NUM_BLOCKS_IN - This is the total number of blocks processed.
- NUM_ERRORS - This is the total number of bit errors within all the processed blocks.
- PLOT_POINT - This signal is active when the current BER point is finished. It activates the data storage and display functions and also starts the next BER point (by being fed into the Input Module.
- TOTAL_SISOS - This is the total number of Soft Input Soft Output (SISOs) used to process all the blocks (See the Turbo Decoder data sheet for further information).
- EB_NO - This is the current Eb/No value used to display the data point at the correct point on the graphs.
- BLOCK_SIZE - This is the current block size used to label the output results file.
- ITERATIONS- This is the maximum number of iterations used to label the output results file.
- FT_THRES - This is the fast termination threshold value used to label the output results file.

Design Operation

Control of the design comes from a MATLAB script file (`demo_ctrl.m`) which defines the parameters for each specific test. This script defines global variables within the MATLAB work space and also starts the System Generator simulation. The global variables are read by the Input and Output modules of the System Generator design to implement each test.

The test process is started by running the `demo_ctrl.m` script, which initializes the global variables and executes the hardware in the loop model.

1. When the model executes, the Input module pulses the INITIAL_FD signal to indicate the start of the process. The Input module resets the Eb/No value to zero and calculates the appropriate EB_NO_SCALE and TURBO_SCALE values. It also reads global variables defining the BLOCK_SIZE, ITERATIONS, and FT_THRES values as defined in the `demo_ctrl.m` script.
2. The INITIAL_FD pulse propagates into the BER module (or its hardware equivalent) which starts the encode and decode cycle. Random data is encoded using the Turbo Encoder and is passed through the noisy channel. The amount of noise added to the random data is defined by the EB_NO_SCALE value. The encoder's RDY signal (see the Turbo encoder data sheet indicates when data is valid from the encoder. This signal is delayed to match any delays introduced into the data path by the noisy channel and then is used to start the decoder.
3. The decoder operates on the noisy random data and indicates that it has finished using its RDY signal. As the decoder starts to output data, counters are updated to monitor the number of bits and frames in error.
4. What has been described thus far is a single encode, add noise, decode, and update counter operation, i.e., a single block has been processed. After the decoder has finished outputting data, the BER module automatically repeats the process and sends the next block of data through the processing chain. This entire process continuously repeats until many thousands of blocks have been processed. Note that the BER module is functioning independently. A single start pulse from the Input Module has initiated this process and the Output module has not been used yet.
5. Blocks are continuously processed in the BER module until a stop condition is met. A stop condition occurs when the maximum number of blocks has been processed or a maximum number of errors has been reached. These maximum values are global variables defined by the `demo_ctrl` script. When the stop condition is met, the PLOT_POINT signal is held High and output from the BER module.
6. The PLOT_POINT signal is monitored by the Output module and triggers the data storage and display functions. The Output module checks the BER value against a user defined minimum. If the current BER value does not exceed the required value, the Output module signals to the Input module that another point is required, using the AUTO_FD_IN signal of the Input module.
7. On receiving the AUTO_FD_IN signal, the Input module increases the Eb/No value by a user defined step. The appropriate TURBO_SCALE and EB_NO_SCALE values are calculated and the encode/decoder process is started again. This process repeats with the Output module continuously measuring the BER value of each generated point. When the measured BER value betters the required minimum, the entire model will be stopped using the STOP SIMULATION block (which can be found by pushing into the Output module).
8. On stopping the model, control returns to the `demo_ctrl` script where the entire process is repeated again using the next set of test parameters. In this way, a series of tests can be executed automatically and without user intervention.

Different Hardware Options

Previous sections have shown how a number of tests parameters can be varied, such as the number of iterations, the fast termination threshold, the maximum number of blocks to be processed, etc. Another set of parameters that might be varied are those that require different hardware implementations. For example, the decoder switching between different algorithms, different input bit widths, different internal bit widths, etc., all require different hardware implementations. Any change to hardware options mean that all the design must be re-synthesized to create a new FPGA programming model.

An added complication is that it might be necessary to run this application on a variety of hardware platforms. For example, the Annapolis Wildcard uses a PCMCIA interface, the Nallatech XtremeDSP kit can use a USB or PCI interface, while the ML402 board uses an Ethernet interface. When the design is synthesized for these platforms, different hardware components will be added to the design to enable communication over the required interface. Each interface has a maximum data rate that it can support; the BER system is designed to utilize a very low communication bandwidth between System Generator and the hardware. Therefore, the hardware platform should not impact system performance significantly, but some variation is expected.

It is clear that the combination of supporting a number of hardware platforms and a number of core hardware variations can result in the requirement for many different FPGA programming files.

Within this application, the FPGA programming files have been renamed to indicate what core hardware options have been selected. Secondly, files associated with different hardware platforms have been placed into separate directories under the `sysgen_bit_files` directory. In this way, many different hardware options can be stored and automatically selected from the appropriate hardware directory.

The issue of automating the synthesis process has not yet been tackled, so it is still necessary to manually synthesize each different core and platform option. However, by storing and naming each synthesis result appropriately, a library of hardware options will soon develop.

Hardware File Naming Convention

This application supports four hardware platforms:

1. The Annapolis Wildcard II using a PCMCIA interface (using a XC2V3000 FPGA).
2. The Annapolis Wildcard 4 using a PCMCIA interface (using a XC4VSX35 FPGA).
3. The Nallatech Extreme DSP Kit using both the PCI and USB interfaces (using a XC2V3000 FPGA).
4. The ML402 board using an Ethernet communication interface (using a XC4VSX35 FPGA).

Consequently, under the `sysgen_bit_files` directory are four sub-directories:

1. `wc2` - programming files for the Wildcard II.
2. `wc4` - programming files for the Wildcard 4.
3. `xdsp` - programming files for the XtremeDSP kit.
4. `ml402_ethernet` - programming files for the ml402 board.

A specific file naming strategy has been adopted to indicate the hardware options selected for each programming file. For example, the file name

```
tcc_3gpp_v1_0_1r3_2i3_6m3_w32_scale.x86
```

can be decoded in the following way:

- `tcc_3gpp_v1_0` = tcc 3GPP design version 1_0.
- `1r3` = supports rate 1/3.
- `2i3` = input bit width is 2 integer and 3 fractional.
- `6m3` = metric width is 6 integer and 3 fractional bits.
- `w32` = window size is equal to 32.
- `scale` = selected algorithm is MAX_SCALE.
- `.x86` = the default bit file name for the Wildcard.

When executing a specific test, the demo_ctrl script reads the user's hardware test requirements and tries to find an FPGA programming file with the appropriate file name. If none is found, the selected test will not run.

Data Output and Display

Data output consists of drawing data to graphs each time a new BER point is calculated and storing the data to text files for later use. Both these functions are controlled from the `plot_ber_fer.m` MATLAB script.

Each time there is a new BER point available, the PLOT_POINT signal (Figure 3) goes High, which causes the Output module to execute the `plot_ber_fer` function. This function reads the various outputs from the BER module to calculate the bit error rate (BER), frame error rate (FER), and the average number of SISOs. These values are drawn on graphs as they are created to provide an immediate performance measure. Text files are created to store the data for later use and these text files are uniquely named as described in the "Output File Naming Convention" section.

To gain further understanding of how global variables are used in the design and how the output file name is created, it is useful to consider Figure 4. This figure shows the inputs into the MATLAB function and the special STOP simulation block that finally stops the simulation. Note that the MATLAB function block can only have one input and output but these can be vectors. Consequently, the black multiplexor function is used to combine the individual signals into a single signal vector.

Note that there are some *constant* blocks at the bottom of Figure 4 such as the one containing MIN_BER and WS. Inside, the *constant* blocks can be fixed numerical values, but in this case the variable MIN_BER and WS have been used. These variables are initialized in the `demo_ctrl.m` script and passed as global variables into these *constant* blocks. In this way, various test parameters can be passed into the `plot_ber_fer` function to create a unique output file name. Note that these variables do not alter during a specific test; they are simply used to help in the automation of the output file name creation.

The constant parameters used in Figure 4 are:

- `MIN_BER` - This is the BER value that the test must achieve. When a BER point that is less than this value is received, the STOP block is executed. Control then passes back to the `demo_ctrl.m` script where the next test is started.
- `WS` - This is the window size used for this test.
- `ALGNUM` - This is the algorithm type used for the test, 0=MAX_STAR, 1= MAX, 2=MAX_SCALE.
- `II` - This is the number of integer bits in the input.
- `IFF` - This is the number of fractional bits in the input.
- `MI` - This is the number of integer bits used in the metric calculations.
- `MF` - This is the number of fractional bits used in the metric calculations.
- `CODE` - This is the code rate used, e.g., 3 represents a rate of 1/3.

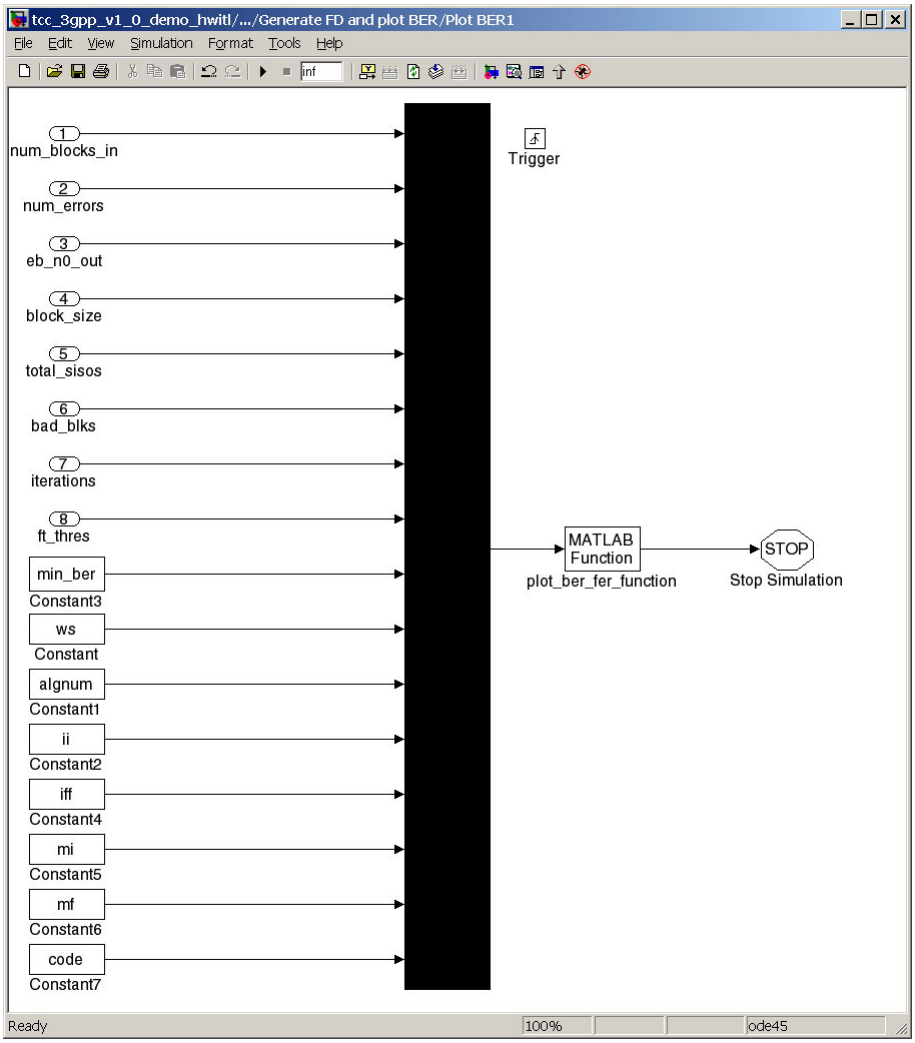

Figure 4: **Inputs to plot_ber_fer.m MATLAB Function**

Output File Naming Convention

The real time displayed data is also stored in a text file for later use. As with the FGPA programming file, the data file is given a name that uniquely identifies the test parameters that were used. All *result* files are automatically placed in the *results* directory. For example, the file

```
tcc_3gpp_v1_0_1r3_2i3_6m3_scale_w32_bs40_i3_ft3_rm0.txt
```

represents results when

- `bs40` = block size of 40
- `i3` = 3 iterations
- `ft3` = `ft_thres` value set to 3
- `rm0` = no rate matching, this is the default for this application. Future versions could support rate matching (puncturing) at a later date.
- other parameters are as described in the programming file naming convention ("Hardware File Naming Convention").

Hardware Library

How the synthesizable BER module in Figure 3 is replaced with the hardware implementation to gain hardware execution speeds has been described earlier. This application creates a complete copy of the design shown in Figure 3 and replaces the BER module with a Configurable subsystem library element to create the top level shown in Figure 5 (`tcc_3gpp_v1_0_demo_hwitl.mdl`).

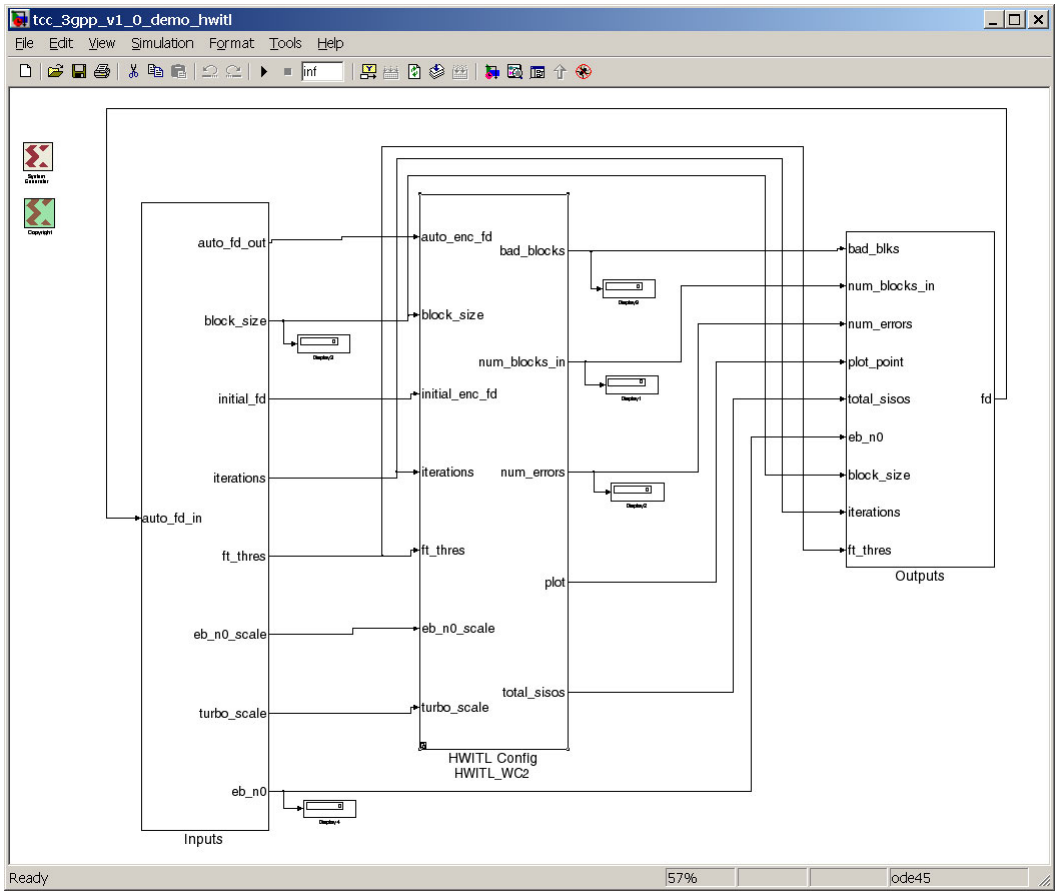

Figure 5: **Top Level with Hardware Library Element**

The Configurable subsystem library element is a convenient way of easily selecting between different hardware platforms. Running the `demo_ctrl.m script` automatically selects the correct hardware platform from the list of available platforms (this is done by changing the *platform* value within the script). Consequently, there is no further user interaction required when using the automated script. However, to provide an understanding of the process and to enable the user to add their own hardware targets, an overview of how Configurable subsystems are used in relation to this design is provided here.

For example, Figure 5 contains the HWITL_config/HWITL_WC2, which indicates that the HWITL_config library element is currently selected to HWITL_WC2, i.e., the Wildcard II hardware platform. To change the hardware platform, simply right click on the HWITL_config block and go to the block choice where five options can be selected:

1. `HWITL_WC2` - Annapolis Wildcard II hardware.
2. `HWITL_WC4` - Annapolis Wildcard4 hardware.
3. `HWITL_ML402_ETHERNET` - ML402 board.
4. `HWITL_XDSP_PCI` - Nallatech XtremeDSP kit PCI version.
5. `HWITL_XDSP_USB` - Nallatech XtremeDSP kit USB version.

The library symbol containing the different choices can be found in the `hwitl_lib.mdl` file a subset of which is shown in Figure 6. As well as being able to manually select the hardware

platform, this process is controlled from the `run_tests.m` script, which is called from the `demo_ctrl.m` script. This method used to select between different hardware platforms automatically within the `demo_ctrl.m` script.

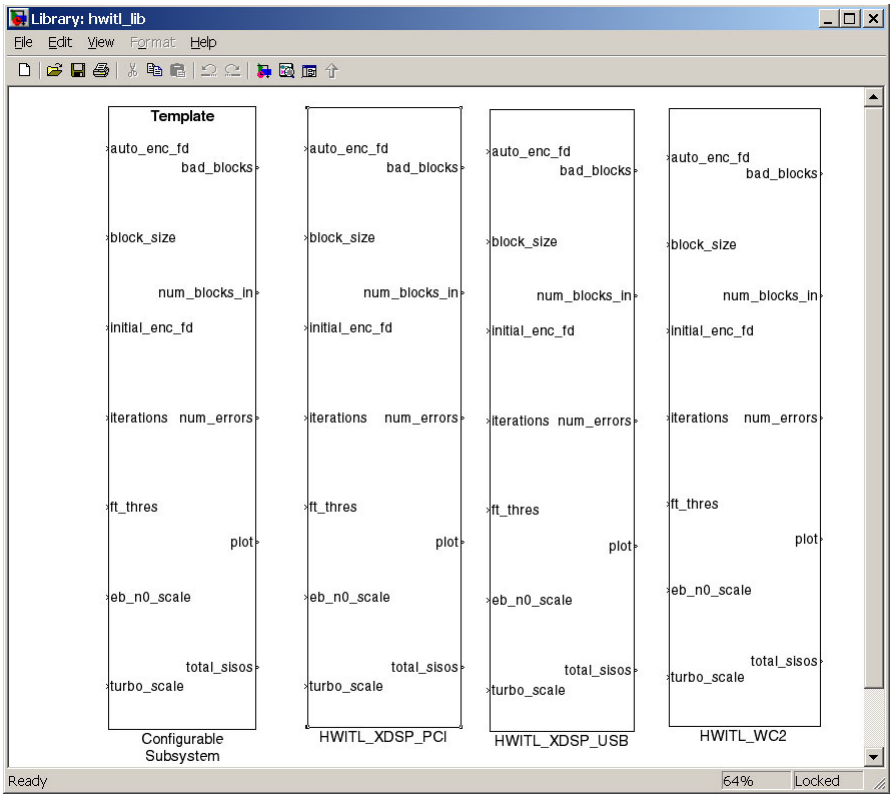

Figure 6: **Library Containing Different Hardware Platforms**

Pushing into the HWITL_WC2 library element shows the hardware block for the Wildcard II as shown in Figure 7. Double clicking on the Wildcard II block produces the control window for the hardware as shown in Figure 8. The important point to note from Figure 8 is the Config File Name, which represents the FPGA programming file. This edit box usually specifies the filename of the FPGA programming file, but in this case it has been replaced by a call to the `select_bit_file.m` function. Passed into this function is the `bit_file` variable which is one of the global variables set up in the `demo_ctrl` script. The aim of this function is that the FPGA programming file can be selected from the `sysgen_bit_files` directory using the naming convention specified in the *"Hardware File Naming Convention."*

Using this combination of hardware library modules and software control of the FPGA programming file provides a powerful and flexible system for automated control of hardware parameters.

Important Note: System Generator parameters that are assumed to be constants cannot be marked as *evaluatable*. In this case, the Config File Name option shown in Figure 8 is not marked as an *evaluate* field by default. This means that if any function is used in this option, it will not be evaluated. This application relies on the fact that the `select_bit_file.m` can be evaluated to apply the correct programming file. Consequently, the evaluate flag must be enabled. All supported platforms given in this application note have been setup correctly, so there is no need to change the evaluate flag in any library element. However, if new hardware platform is added and synthesized, the following procedure must be followed for the correct bit file to be specified. Open the `hwitl_lib.mdl` file and unlock the library from the Edit option. Go to the hardware block (e.g., Figure 7) and right click on it. Select Edit Mask and select the parameters tab. Ensure that the *evaluate* tick box is selected next to the Config File Name. The `select_bit_file` function can now be evaluated (this function will probably need to be edited if a new hardware module is set up so that it can select the correct bit file for the new platform).

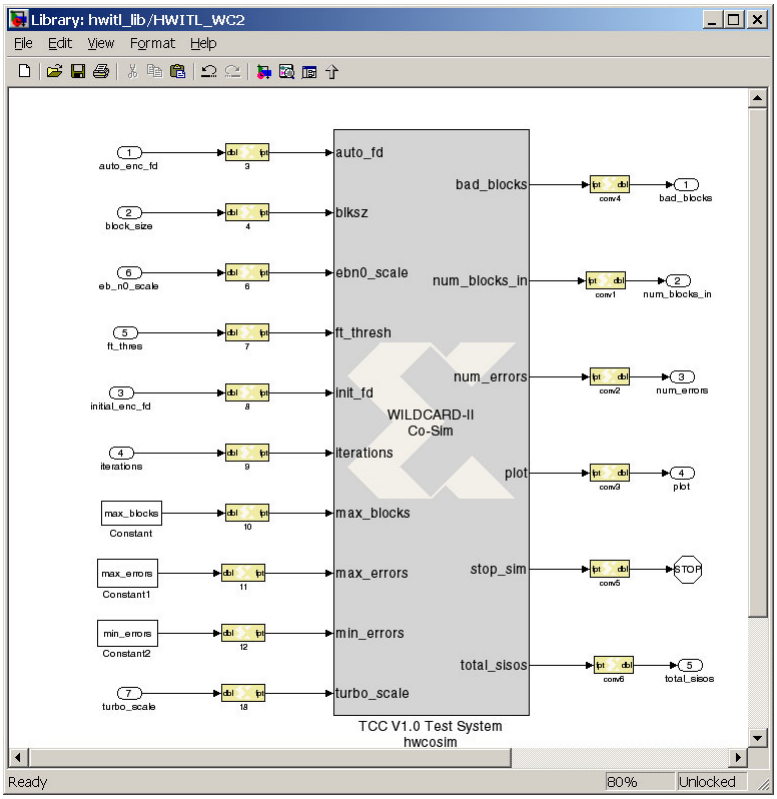

Figure 7: **Hardware Block for Annapolis Wildcard II**

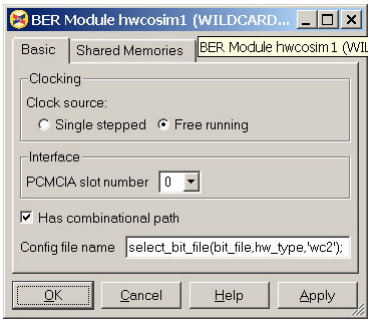

Figure 8: **Main Control Window for the Wildcard II**

Creating a New FPGA Programming File

The files associated with this application contain all the files necessary to run this example. However, if a different hardware platform needs to be supported or if another hardware variation of the Xilinx Turbo Encoder or Decoder is required, the design must be re-synthesized. This section describes the steps necessary to create and synthesize the design from scratch. These steps are not necessary to execute this application but they are included only as additional background information.

1. Create the Xilinx Turbo encoder and decoder hardware design files. The encoder and decoder are brought into the simulation as NGC or EDN files. These files are created using the Xilinx CORE Generator™ tool, where the user can select different hardware design options for the cores. The output from CORE Generator is usually an NGC or and EDN file containing the hardware design.

2. Valid CORE Generator licenses are required for the cores and two license types are available:
   a. **Evaluation License** - This can be downloaded from the Xilinx website and enables a core to be used for evaluation purposes. The core is fully functional but stops operating after *several* hours. Evaluation version examples of the encoder and decoder are provided with this example and, consequently, the BER simulation stops working after a *few* hours of continuous use.
   b. **Purchased License** - If the core is purchased, then a full license is provided. In this case, the core operates without restriction
3. The encoder and decoder are brought into the rest of the design as *black boxes*. This is a System Generator process where a top-level VHDL entity description is read to create the System Generator encoder and decoder blocks. This process also creates a configuration file (e.g., `tcc_encoder_3gpp_v2_0_config.m`) that creates the various ports on the block diagram. The configuration file is also used to associate the synthesizable EDN or NGC file with the block diagram.
4. Before the model can be synthesized, it is important to check that the synthesizable encoder and decoder files are compatible with the overall design. For example, this design uses a decoder supporting rate 1/3 and not rate 1/5. Therefore, the correct option must be selected in the CORE Generator stage for the decoder. Similarly, the soft input width of the decoder must match with that generated by the channel models. Some of these global settings are passed through the synthesizable model as *mask parameters*. For example, double click on the BER module block (Figure 3) from the synthesizable model file and an edit box will appear. This is used to set the integer and fractional bit width of the channel model and it must match up with that selected during the decoder CORE Generator stage.
5. System Generator uses the clock enable (CE) port of a core to control its execution within the System Generator environment. For example, single stepping of the hardware block is controlled via the CE port. Therefore, it is always necessary to create a core with the CE port included.
6. Navigate to the BER module block in the synthesizable model and push down the hierarchy using the *look under mask* option. Double click on the System Generator icon and various hardware platforms and synthesis tools can be selected. Select the appropriate options (i.e., the synthesis tool and hardware platform available) and hit the *generate* button to start the synthesis process.
7. Following the successful run of the synthesis tools, a hardware library block is created that contains the synthesized design. This block is then copied to the `hwitl_lib.mdl`, so that this hardware version can be selected and executed. Double clicking on one of these hardware blocks opens an edit box with various parameters, including the *bit file name* or *config file name*. The "Hardware Library" section provides more detail on this process. Note that after a suitable hardware platform has been placed into the library, the `select_bit_file` function can be used to recover specific versions for that platform. Only one copy of each hardware platform is required in the hardware library. In cases where a hardware library element already exists within the `hwitl_lib.mdl`, the key operation is to capture the hardware programming file (bit file) and rename it as described in the next section.
8. The programming file created for the FPGA is renamed to describe the various hardware options selected and put into the `sysgen_bit_files` directory under the appropriate hardware subdirectory. (Note that creating a new hardware platform requires changes to the `demo_ctrl.m` and the `run_tests.m` files, so that they know the directory name where to find programming files.)
9. Processes 1-6 are repeated until the required hardware platforms and options have been created to meet the Turbo encoder and decoder testing requirements.

Design Operation

Design operation is straightforward and driven from the `demo_ctrl_script.m` file. The user can edit this file to select different test options, such as how many iterations to perform, what block sizes to execute, etc. These types of test parameters do not change the hardware platform or decoder hardware options. A second set of parameters, such as selecting a different algorithm, different channel, or internal bit widths require a different FPGA hardware version. In this case, the *d*`emo_ctrl.m` script tries to find an appropriately named bit file with the correct hardware options.

The application is started by changing the directory in MATLAB to where the design files are stored and simply executing the `demo_ctrl.m` script. This script sets up a variety of global variables and calls the `run_tests.m` file that executes a series of software loops, each performing the tests specified in the `demo_ctrl.m` script.

A variety of options can be selected from the `demo_ctrl.m` script with the main ones below. Read the Comments in this script to gain more information.

- The hardware platform can be selected.
- A demonstration mode can be selected that runs on a continuous loop.
- File overwriting can be disabled or enabled. With file overwriting disabled, if the output results file already exists the test is skipped. This can save a lot of computation time.

There are a number of parameters in the `demo_ctrl` script that significantly change the time taken to implement a specific set of tests:

- **Maximum Number of Bits Processed**. After the number of bits through the encode, add noise, and decode process has met or exceeded this value, the calculations for this point finishes. The larger the number, the larger the simulation time.
- **Maximum Number of Errors Per Point.** When the original and decoded data sequences have differed by more that this value, the point finishes. Most of the time at low EB/No values, it will be the maximum number of errors that stops calculation of a point, while the maximum number of bits processed stops the point for higher Eb/No values. (Generally, there are less errors for higher Eb/No values).
- **Minimum BER Value**. This is the main driver to finish a particular test. The test runs until this minimum BER value has been passed. For example, setting this to 1e-5 means that the test ends when a BER of better than 1e-5 has been measured. This has a large impact on test time. Values in the region of 1e-4 produce tests that last a few minutes, while values of 1e-8 can take many hours.
- **Eb/No Step**. There are usually many BER points calculated before the specified BER is achieved. Increasing the Eb/N0 step size reduces the time taken to get to a specific Eb/No value and, therefore, reduces simulation time.

It is necessary to adjust all the above parameters to achieve particular test goals. The larger the amount of data processed, the more accurate the results, and the smoother the graphs, but at the expense of testing time.

By default the `demo_ctrl.m` script is set to use the Annapolis Wildcard II and implement two different block sizes (378 and 762) with two different number of iterations (3 and 5) and with two different fast termination thresholds (0 and 3). This produces a combination of 2x2x2=8 tests with eight result files written to the results directory. The minimum BER value has been set to 1e-4 and the maximum number of errors is 5000. These combinations ensures relatively short test times.

As well as the BER and FER graphs, the application also shows the average number of SISOs that have been used to decode each block. With Fast Termination enabled, the decoder attempts to decode a specific block in the minimum time by monitoring the output for each SISO operation. If it is a user-defined number of SISO operations, all give the same decoded result, then the decoder assumes that it has correctly decoded the block. Internally, the decoder can be made to keep a count of how many SISO operations this takes, and this application averages this result and displays it. This means that for a given Eb/N0 value the maximum

throughput for the core can be optimized by implementing a minimum number of iterations. If the decoder does not fast terminate, then it executes the maximum number of iterations as defined by the ITERATIONS port. If Fast Termination is enabled, but the FT_THRES value is zero, Fast Termination is essentially disabled. In this case, the average SISO graph shows a fixed number of average SISOs = 2 x the number of iterations.

Note: This application supplies evaluation copies of the Xilinx Turbo Encoder and Decoder which automatically fail after several hours operation (depending on clock rate). Therefore, this application will fail if it is left running for many hours. See the Xilinx website for obtaining full versions of these cores. (www.xilinx.com/ipcenter/ipevaluation/index.htm)

Displaying Output

It is possible to generate many hundreds of results with different block sizes, numbers of iterations, algorithm type, etc. Consequently, a useful script for displaying data has been developed `plot_script.m`. This script allows the user to display virtually any combination of results that have been stored in the *results* subdirectory. For example, with a fixed block size, the performance with different numbers of iterations can be monitored or the performance of the core with and without Fast Termination can also be displayed.

A number of examples in the way the plot script can be used are provided within the script itself. These examples use the eight files generated in the default test case and display a variety of combinations. Note that the eight result files are provided within this application file set so it is not necessary to run the hardware before using this script.

Comments have been added to the file to show how this can be used. To generate the graphs, simply run the script from within the MATLAB environment.

Reference Design Files

This section lists and briefly describes the reference design files used within this application. These files can be downloaded from zapp948.zip.

System Generator Models

- `tcc_3gpp_v1_0_demo_synth.mdl` - the design containing the synthesizable test module (Figure 3).
- `tcc_3gpp_v1_0_demo_hwitl.mdl` - the design containing the synthesizable test module (Figure 5).
- `hwitl_lib.mdl` - the design containing the FPGA programming files for different hardware platforms (Figure 3).
- `tcc_awgn_lib.mdl` - the AWGN noise channel converted into a library component.
- `select_bit_file.m` - MATLAB function to select a specified FPGA programming file.

MATLAB Control Scripts

- `demo_ctrl.m` - the main control file for running the test suite.
- `run_tests.m` - this script is called from the `demo_ctrl.m` file and is used to run the test suite.
- `calc_scale_values.m` - a MATLAB script called from within System Generator to calculate the noise channel and Turbo decoder scaling values.
- `plot_ber_fer.m` - a MATLAB script called from within System Generator to display results graphically and store data to a file.
- `show_ber_plots.m` - a script to combine various result files and display them graphically.
- `combine_ber_plots.m` - a script called by the *s*`how_ber_plots.m` file.
- `tcc_encoder_3gpp_v2_0_config.m` - this is the System Generator configuration file for the Xilinx Turbo encoder. This file is created when the encoder component is created

as a *black box* within System Generator. This file is edited to link the black box with its associated hardware file `tcc_encoder_3gpp_v2_0.ngc` generated from CORE Generator. This allows the synthesis tool to combine the encoder into the overall design.

- `tcc_decoder_3gpp_v1_0_config.m` - this is the System Generator configuration file for the Xilinx Turbo decoder. This file is created when the decoder component is created as a black box within System Generator. This file is edited to link the black box with its associated hardware file `tcc_decoder_3gpp_v2_0.edn` generated from CORE Generator. This allows the synthesis tool to combine the decoder into the overall design.

Xilinx Files

- `tcc_encoder_3gpp_v2_0.vhd` - dummy top-level file containing the port descriptions of the Xilinx Turbo Encoder V2_0. This file is selected and read during the creation of the System Generator black box to create the System Generator encoder block. The port list from this file is used to create the *t*`cc_encoder_3gpp_v2_0_config.m` file.
- `tcc_decoder_3gpp_v1_0.vhd` - dummy top-level file containing the port descriptions of the Xilinx Turbo Decoder V1_0. This file is selected and read during the creation of the System Generator black box to create the System Generator decoder block. The port list from this file is used to create the `tcc_decoder_3gpp_v1_0_config.m`.
- `tcc_encoder_3gpp_v2_0.ngc` - the synthesizable encoder design file.
- `tcc_decoder_3gpp_v1_0.edn` - the synthesizable decoder design file.

System Requirements

The following software and hardware are required to successfully execute and run these design files.

The following software items are mandatory:

1. MATLAB Release 14 with Service Pack 3 or later
2. ISE™ 8.2i Service Pack 3.
3. System Generator 8.2
4. ISE IP update 8.2i IP2

Installation Tip: System Generator has a limitation of 256 characters for any file name. Therefore, do not install the software files associated with it within a deeply nested directory structure.

At least one of the following hardware development boards must be available:

1. The Nallatech XtremeDSP kit and associated FUSE software.
2. The Annapolis Wildcard II or Wildcard 4 PCMCIA card and associated driver software.
3. The ML402 board and associated software, configured for Ethernet co-simulation.

   Note: This example assumes that the ML402 is part of a network (different to a point-to-point link). Refer to the System Generator documentation on the use of Ethernet-based hardware co-simulation. This can be found in the *Using FPGA Hardware in the Loop* section of the Xilinx System Generator documentation within the MATLAB “help” facility.

Other hardware can be used with these design files, but it is left to the user to create these versions.

Note: If a particular hardware platform’s software is not installed on the user machine, the user might not have access to all the features described in this document for that platform.

Troubleshooting

When working within the `hwitl_lib.mdl` file and trying to change parameters within any of the different hardware blocks, the user might sometimes experience a System Generator error similar to `Parameter bitfileName did not evaluate to type 'String'`. For example, this may occur if the user wants to change the Ethernet address on the ML402 board.

The "Hardware Library" section described how the programming file for the FPGA is obtained by evaluating the `select_bit_file.m` function. This function expects input parameters of bit_file, hw_type, and device. The bit_file parameter defines the bit file name, the hw_type parameter defined the hardware type (e.g., wildcard or ML402) and the device defines the FPGA. If the user has not executed the `demo_ctrl.m` script, then each of these input parameters will not be defined within the MATLAB work space and, therefore, the `select_bit_file.m` function will not evaluate to a string, hence, the System Generator error. There are two solutions to this problem:

1. Run the `demo_ctrl.m` script before attempting to change any of the hardware block parameters. It should not matter if it produces an error at this stage, because the point here is to simply set the input variables within the MATLAB work space.
2. Within MATLAB, define each of the three input parameters to create MATLAB work space variable with strings such as:

   ```
   bit_file='temp'
   hw_type='wc2'
   device='xc2v3000'
   ```

Each of the processes above will assign default values to the input parameters and, therefore, remove the System Generator error described above.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/05/06	1.0	Initial Xilinx release.