Capsim® Block Generator

LINUX/MAC OS X Environment

Table of Contents

List of Figures

Introduction

This tutorial describes how to generate custom blocks using tools provided with Capsim. The tools use graphical user interfaces to simplify the generation of C embedded in XML source code for a variety of blocks.

Creating a Working Directory for Custom Capsim

The first step in adding a new block to Capsim is to create a working directory to house the new block's source code, C code and also any subroutines that may be used. Startup a console .Type 'cd' to make sure you are in your home directory (usually /home/user_name). Create a new directory (we will create a directory called WORK):

```
mkdir WORK
```

Next type the command:

```
bash  $CAPSIM/TOOLS/precapsim.sh
```

Figure 1 shows the result of executing the *precapsim.sh* script. After executing this command, type the *make* command:

```
make
```

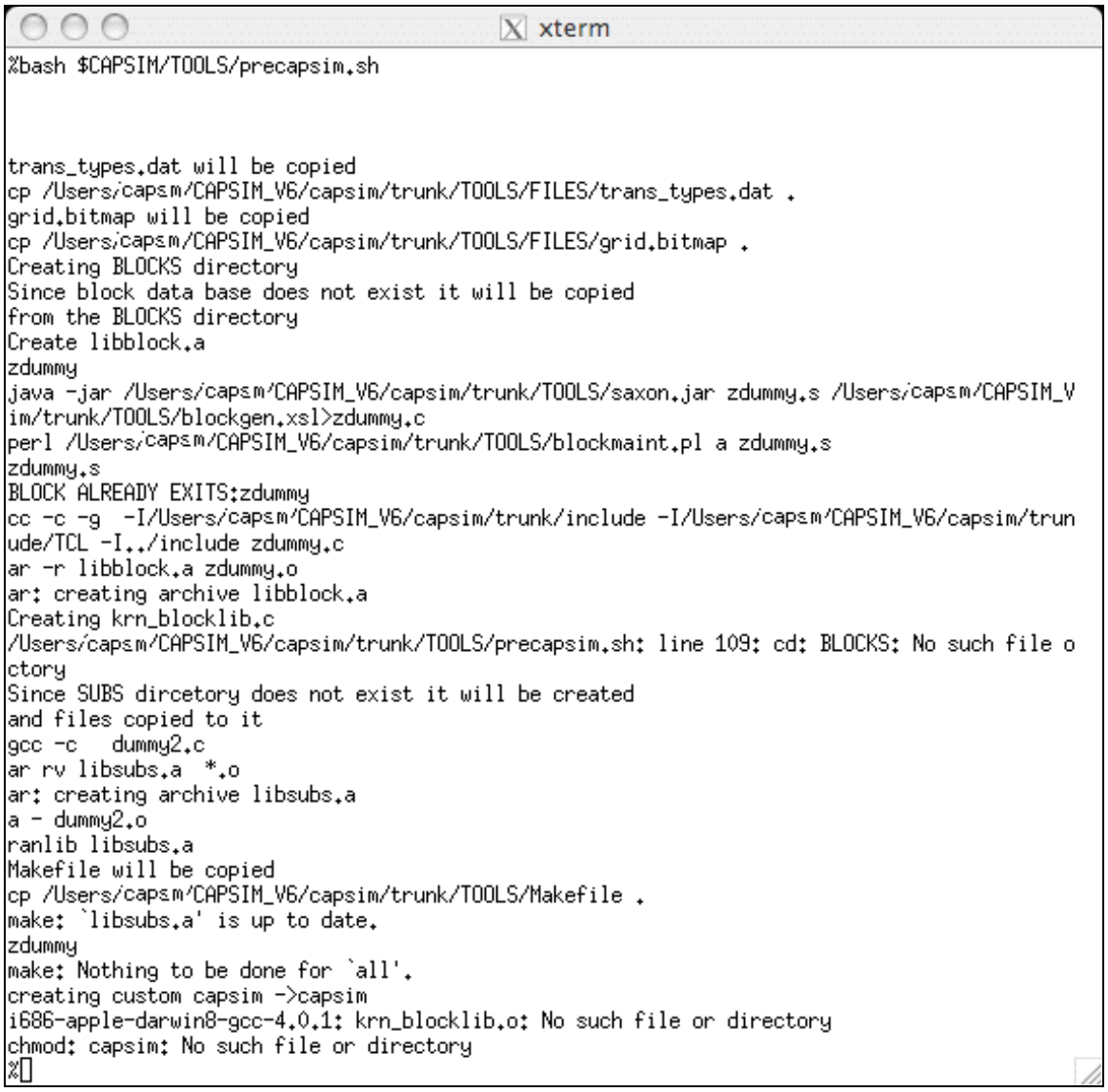

Figure 1 Console After Executing precapsim.sh Script

After executing the *precapsim,sh* script and *make* type 'ls' as shown in Figure 2. Notice that two new directories BLOCKS and SUBS have been created. Also a new CAPSIM executable *capsim* is created. Notice also the *Makefile*.

The directory structure for developing blocks is shown in Figure 3.

When you want to update CAPSIM, after you add a block or subroutine, the *Makefile* is used to create a new *capsim* executable.

To bring up CAPSIM type:

```
./capsim
```

Figure 4 shows the Capsim workspace and console.

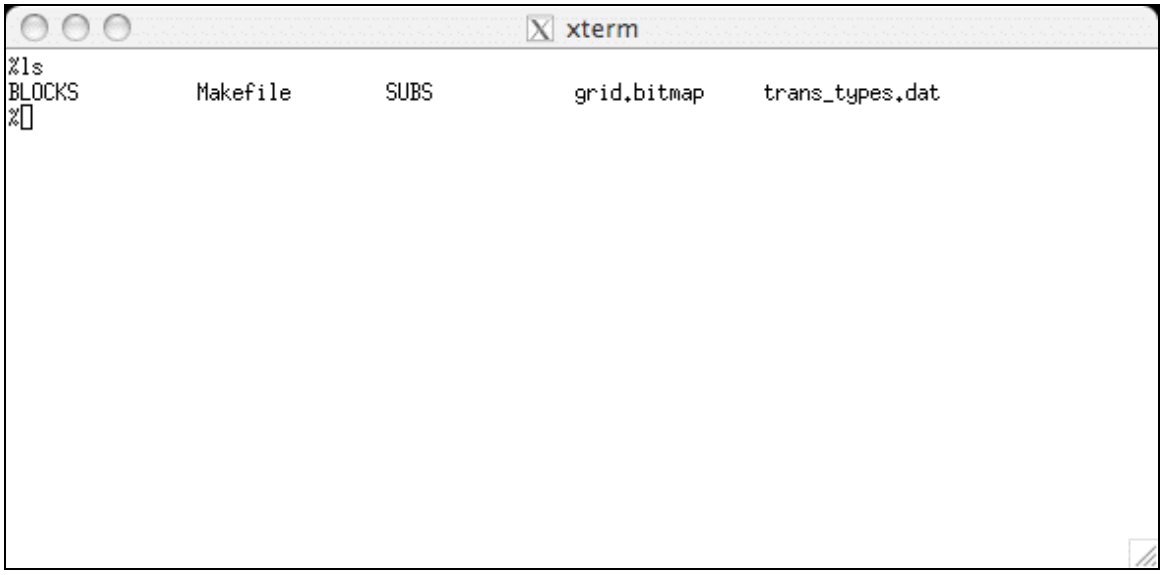

Figure 2 Folders and Files Created in Working Directory

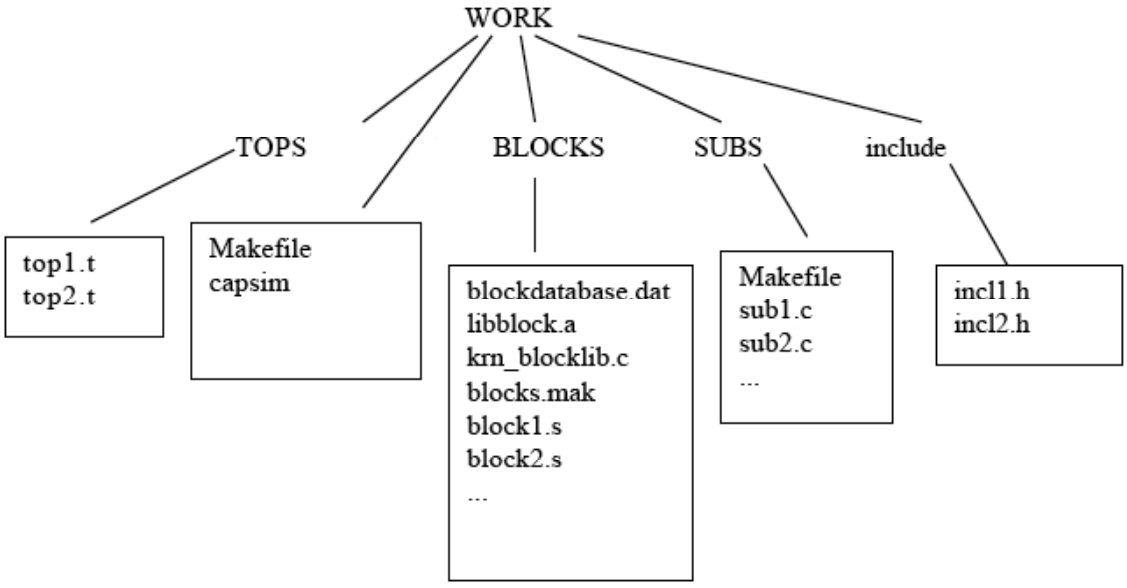

Figure 3 Capsim Block Development Directory Structure

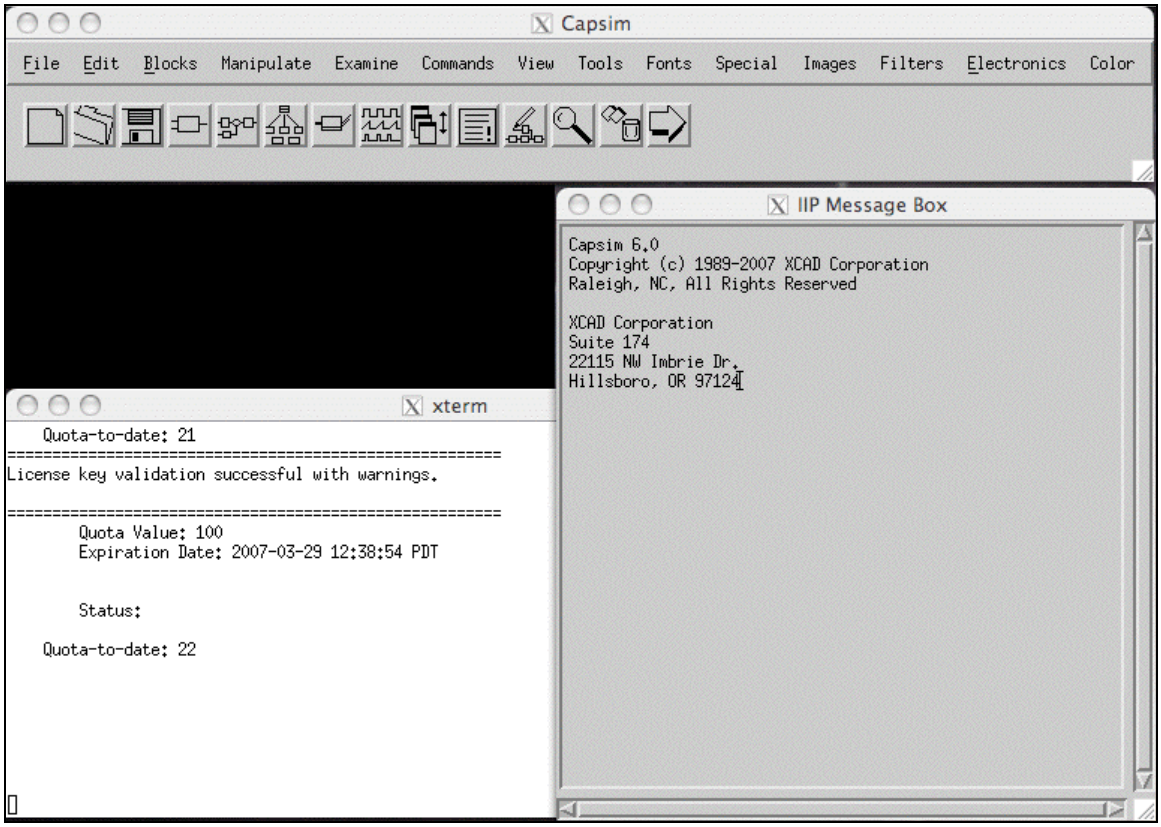

Figure 4 Running Capsim

Notice the Licensing information.

Creating a Custom Source Block

In this section we will create a source block (a block that generates samples) and add it to CAPSIM. Blocks in Capsim are written in "C embedded in XML" to provide the interface to the block both in terms of input/output connections and parameters and the C code that implements the algorithm and functionality. The XML code is transformed to C code using the *saxon* Java XSLT processor. The same XML code is transformed to HTML. This is a flexible methodology in that the code can be transformed for many purposes. See Figure 5.

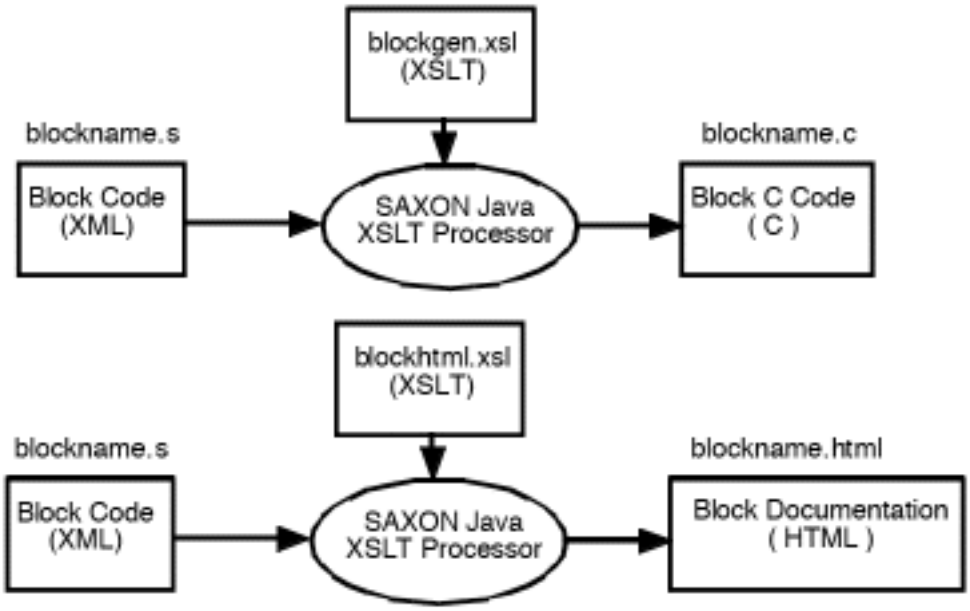

Figure 5 Block C Embedded in XML (*blockname.s*) to C Code Transformation

Change directories to the BLOCK directory by typing

```
cd BLOCKS
```

Type 'ls' as shown in Figure 5. Note the files in the BLOCK directory. There is a dummy block called *zdummy.s*, a make file for blocks, *blocks.mak*, a block database of all blocks included in Capsim, *blockdatabase.dat*. Finally there is the C code *krn_blocklib.c* which is linked into CAPSIM.

To start the block generation tool type the following command:

```
wish $CAPSIM/TOOLS/blockgen.tcl
```

Note that *wish* is the command to invoke TK/TCL.

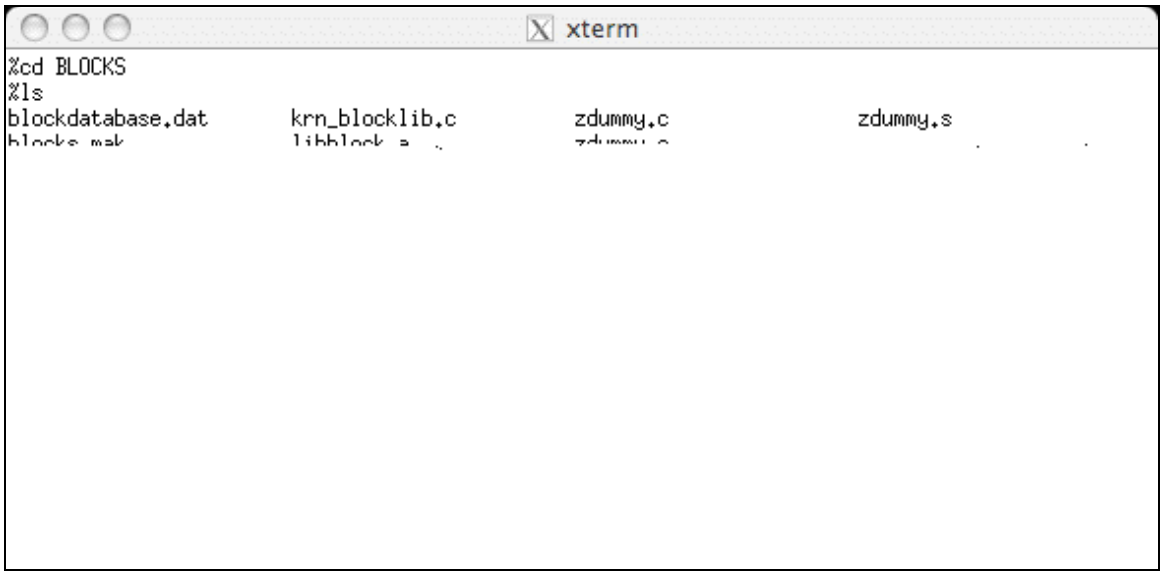

Figure 6 BLOCK Directory

The Block Generation User Interface shown in Figure 7 appears. There are two windows. The key window is the *blockgen* window (to the right). In this window you select the type of block to generate and the type of input/output buffer (float, int, fixed point, complex and image). The second window is for specifying parameters.

We will create a source block called *xsource*. It will have a floating point buffer and a parameter called *ampltude*. Select "Source" for the Block Type Radio Button. Select *float* for the Buffer Type. Type in the blocks name in the name field (*xsource*). In the parameter window type *Amplitude* for the Parameter Prompt. Type 100.0 for the Default Parameter Value. For the Parameter Name type *amplitude*. Click on the Add Parameter button to add the parameter to the list. If you make a mistake you can select the parameter and delete it.

After completing the above steps (double check against Figure 8) click on the Generate button in the *blockgen* window. A confirmation for the the block generation will appear. Click Okay.

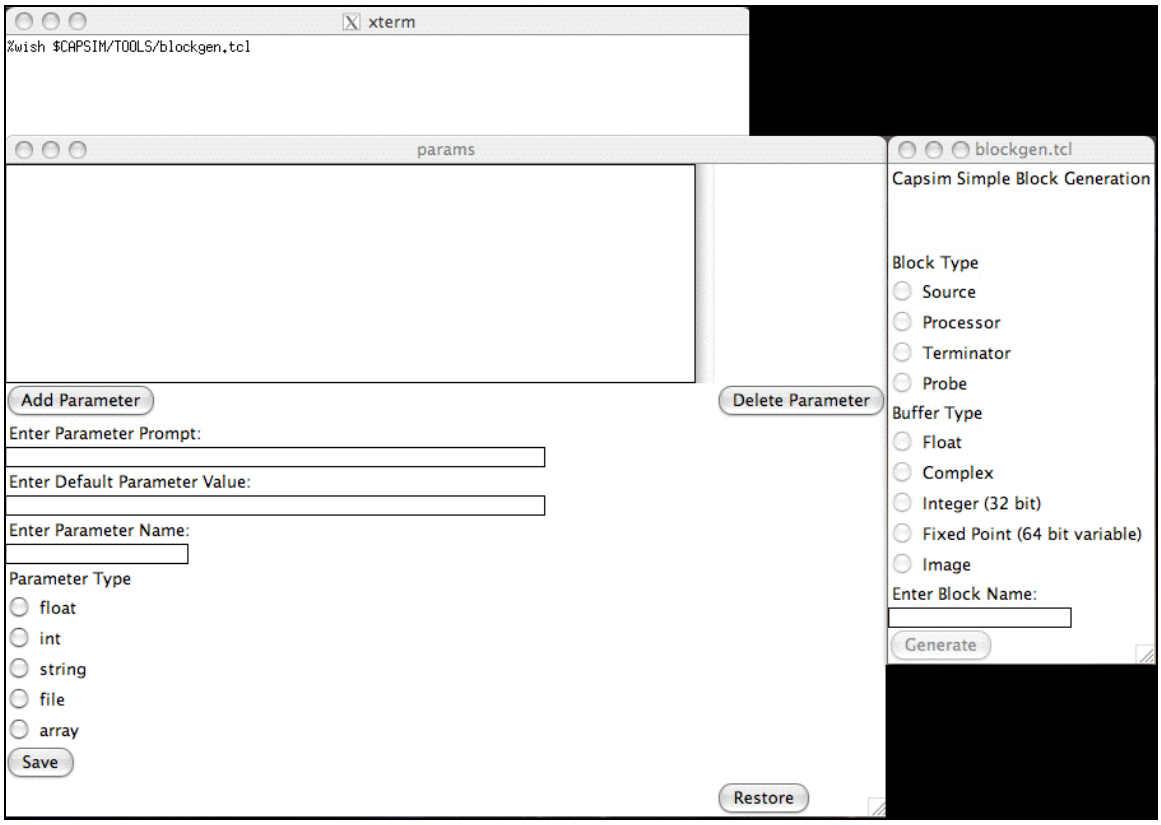

Figure 7 Block Generation Tool

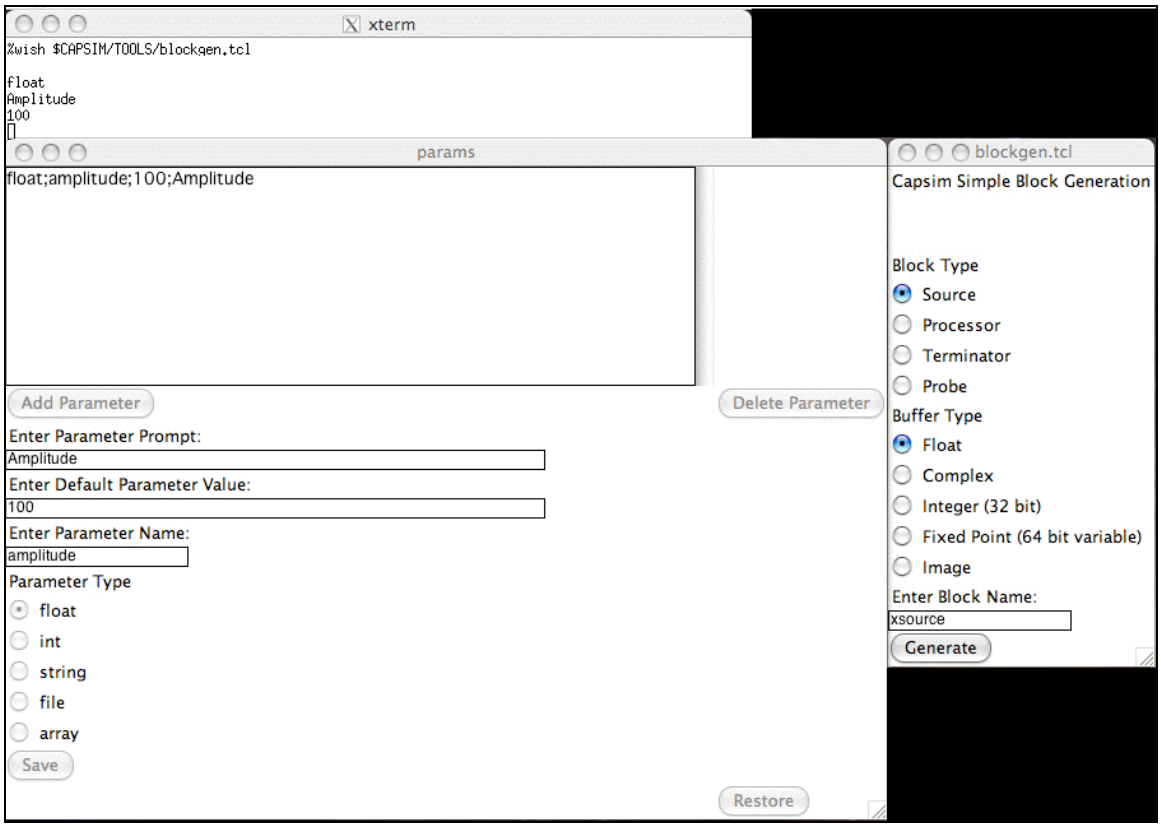

Figure 8 Source Block Settings for xsource

Go ahead and quit the Block Generation Tool by closing the *blockgen* window. Type 'ls' in the BLOCKS directory. The new source code *xsource.s* appears as shown in Figure 9.

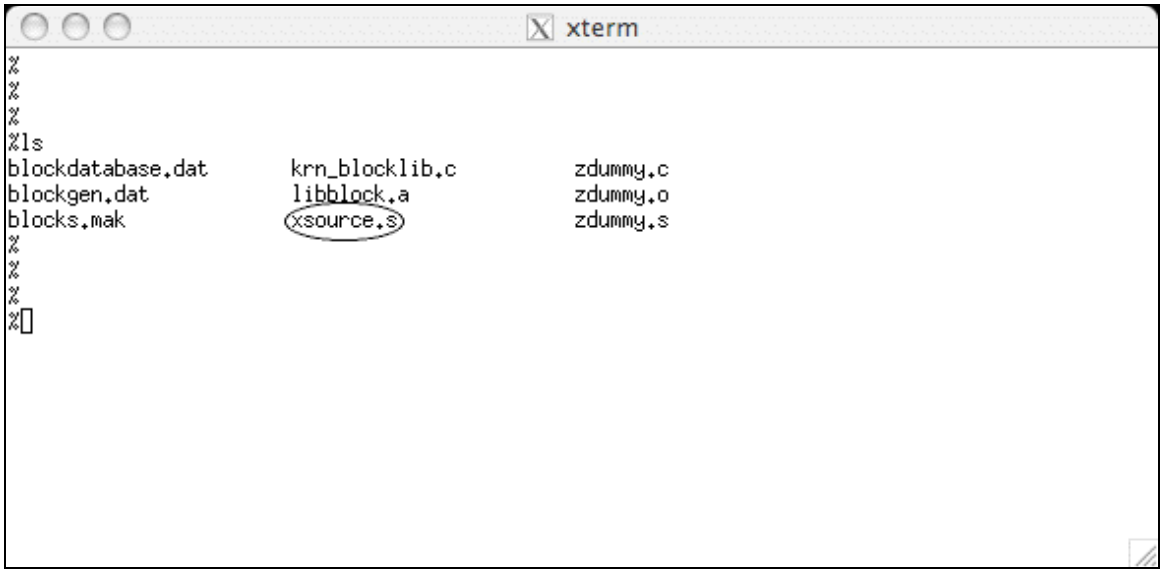

Figure 9 New Block xsource.s Created

Examining the Source Code of the Generated Block

In the following we will examine the source code of the generated BLOCK *xsource.s.* We will modify it so that it produces a linear ramp depending on the amplitude.

```
<BLOCK>
<LICENSE>
/*
 * (c) 2006 AUTHOR_NAME
 */

</LICENSE>
<BLOCK_NAME>
xsource
</BLOCK_NAME>

<DESC_SHORT>
Add short description here. Will appear in HTML documentation.
</DESC_SHORT>

<COMMENTS>
<![CDATA[

/*
 *          xsource()
 *
 * Generated by blockgen
 *
 * Floating point source
 * Notes:
 *    This is a block generated from the floating point source template
 *    It will output numberOfSamples  of samples defined by a parameter.
 *    It outputs the floating point constant 1.0
 *
 *
 *
 * Programmer:
 * Date:
 * Modified:
 *
 */
]]>
</COMMENTS>
```

Figure 10 Source for *xsource.s* Block <BLOCK_NAME> Tag

```
<DECLARATIONS>

      int i,j;

</DECLARATIONS>



<STATES>
      <STATE>
            <TYPE> int </TYPE>
            <NAME> done </NAME>
            <VALUE> FALSE </VALUE>
      </STATE>
      <STATE>
            <TYPE> int </TYPE>
            <NAME> sampleCount </NAME>
            <VALUE> 0 </VALUE>
      </STATE>
</STATES>



<PARAMETERS>
        <PARAM>
              <DEF> Amplitude </DEF>
              <TYPE> float </TYPE>
              <NAME> amplitude </NAME>
              <VALUE> 100.00 </VALUE>
        </PARAM>
      <PARAM>
            <DEF> Total number of samples to output </DEF>
            <TYPE> int </TYPE>
            <NAME> numberOfSamples </NAME>
            <VALUE> 1000 </VALUE>
      </PARAM>
</PARAMETERS>
```

Figure 11 Source Code *xsource.s* STATES and PARAMETERS with Amplitude Parameter

Figure 11 shows the PARAMETER Tag. Note the inclusion of the *Amplitude* parameter. Also the *numberOfSamples* parameter is automatically added by the block generator. This is the total number of samples to generate.

```
<OUTPUT_BUFFERS>
      <BUFFER>
            <TYPE> float </TYPE>
            <NAME> x </NAME>
      </BUFFER>
</OUTPUT_BUFFERS>

<INIT_CODE>
<![CDATA[
]]>
</INIT_CODE>

<MAIN_CODE>
<![CDATA[
      if(done)return(0);
      /*
       * output a maximum of NUMBER_SAMPLES_PER_VISIT samples to output buffer(s)
       */
      for(i=0; i < NUMBER_SAMPLES_PER_VISIT; ++i) {
         sampleCount++;
         if(sampleCount == numberOfSamples) {
             done=TRUE;
             return(0);
         }


         /*
          * ready output buffer for sample
          * check for overflow
          */
         if(IT_OUT(0)) {
                        KrnOverflow("xsource",0);
                        return(99);
         }
         /*
          * output the sample
          */
         x(0)=1.0;

      }
]]>
</MAIN_CODE>
```

Figure 12 Source Code *xsource.s* MAIN_CODE

In the MAIN_CODE tag note that the output buffer x(0) is set to 1.0.

```
/*
 * Output the sample
 */
x(0)=1.0;
```

We will change this to the following (use your favorite editor):

```
        /*
         * Output the sample
         */
       x(0)=amplitude*sampleCount/numberOfSamples;
```

See Figure 13.

```
<OUTPUT_BUFFERS>
      <BUFFER>
            <TYPE> float </TYPE>
            <NAME> x </NAME>
      </BUFFER>
</OUTPUT_BUFFERS>

<INIT_CODE>
<![CDATA[
]]>
</INIT_CODE>

<MAIN_CODE>
<![CDATA[
      if(done)return(0);
      /*
       * output a maximum of NUMBER_SAMPLES_PER_VISIT samples to output buffer(s)
       */
      for(i=0; i < NUMBER_SAMPLES_PER_VISIT; ++i) {
         sampleCount++;
         if(sampleCount == numberOfSamples) {
             done=TRUE;
             return(0);
         }


         /*
          * ready output buffer for sample
          * check for overflow
          */
         if(IT_OUT(0)) {
                       KrnOverflow("xsource",0);
                       return(99);
         }
         /*
          * output the sample
          */
         x(0)=amplitude*sampleCount/numberOfSamples;

      }
]]>
</MAIN_CODE>
```

Figure 13 Code Change to Generate Linear Ramp

In the code, *sampleCount* is a State Variable that keeps track of the number of samples generated. The variable *numberOfSamples* is a parameter that specifies the number of samples to generate. The variable *amplitude* is the parameter that we specified in the block generation tool. (In the generated C code these are actually defined MACROS that point to variables in data structures, i.e. States and Parameters). Save the changes.

Adding the Block to Capsim

At this point, change directories back to the top (the directory WORK) by typing

```
cd ..
```

Next type:

```
make
```

The block code is converted from XML to C code by Java. The C code is compiled. The block is added to the *libblocks.a* archive and linked into Capsim. A new executable *capsim* is created. If a new block is added, always execute make again. Otherwise the block will not appear in the block list. This is not necessary for updating existing blocks. Just type *make* once. So go ahead and type make again since we added a new block.

```
make
```

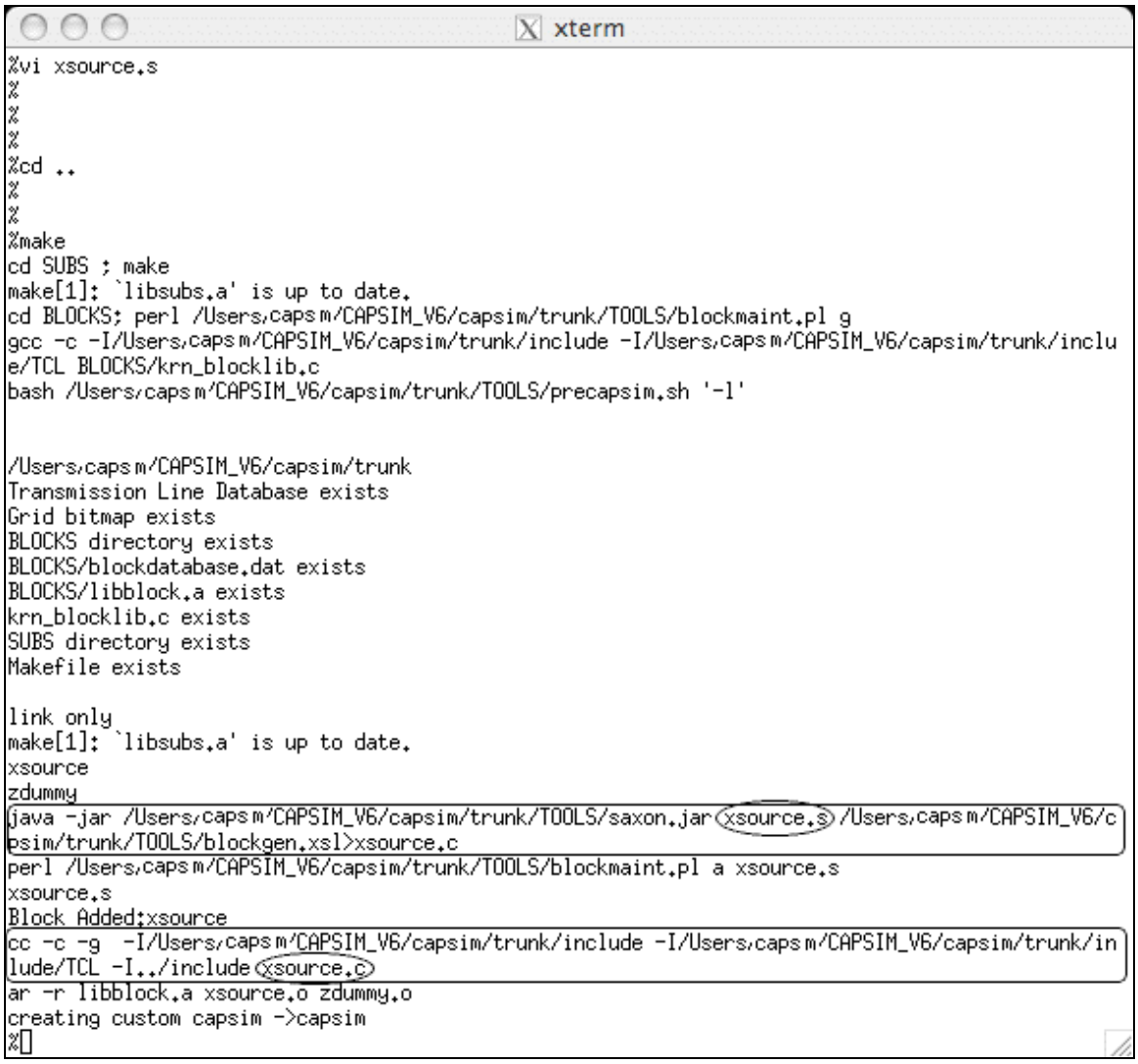

Figure 14 Make Creating C Code, Compiling and Linking Block into Capsim

Hopefully all went well. If you see any errors during compilation there are limited to the code you changed. Focus on that code, fix it in the BLOCKS directory and type make in the WORK directory.

It is time to run CAPSIM with the incorporated *xsource* block. Type *capsim* in the WORK directory. This will bring up CAPSIM.

From the File menu select “New” to create a new blank workspace.

From the Blocks menu select “Blocks …”. The list of blocks will appear. Scroll down (to the buttom) and select *xsource*. (If you can’t find xsource, then exit CAPSIM and type make again. Look for any errors and fix). Place the *xsource* block on the workspace as shown in Figure 15.

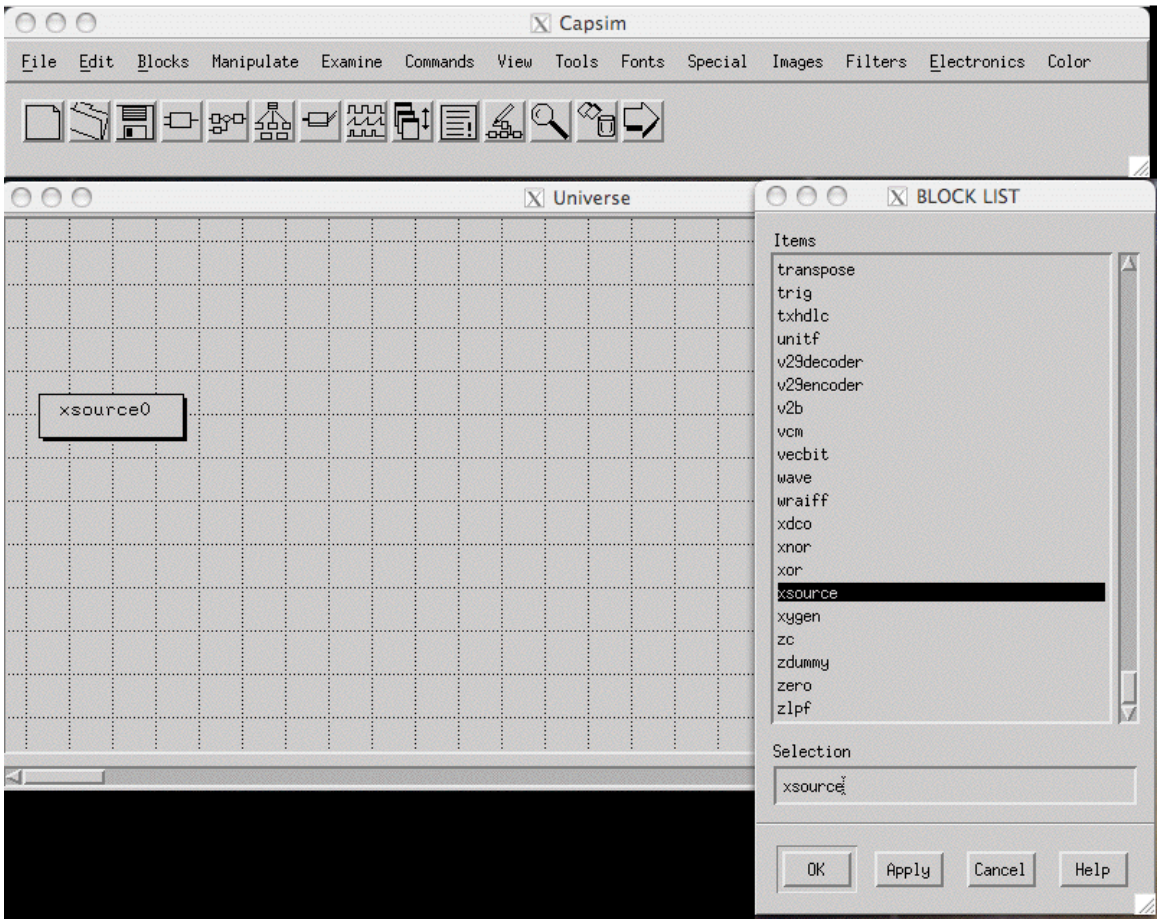

Figure 15 Placing the xsource Block on the Workspace

Next select the *plot* block from the list and place it on the workspace. Connect the *xsource* block to the *plot* block. Save the topology (name it *test_xsource*). Run the simulation. A plot should appear as shown in Figure 16.

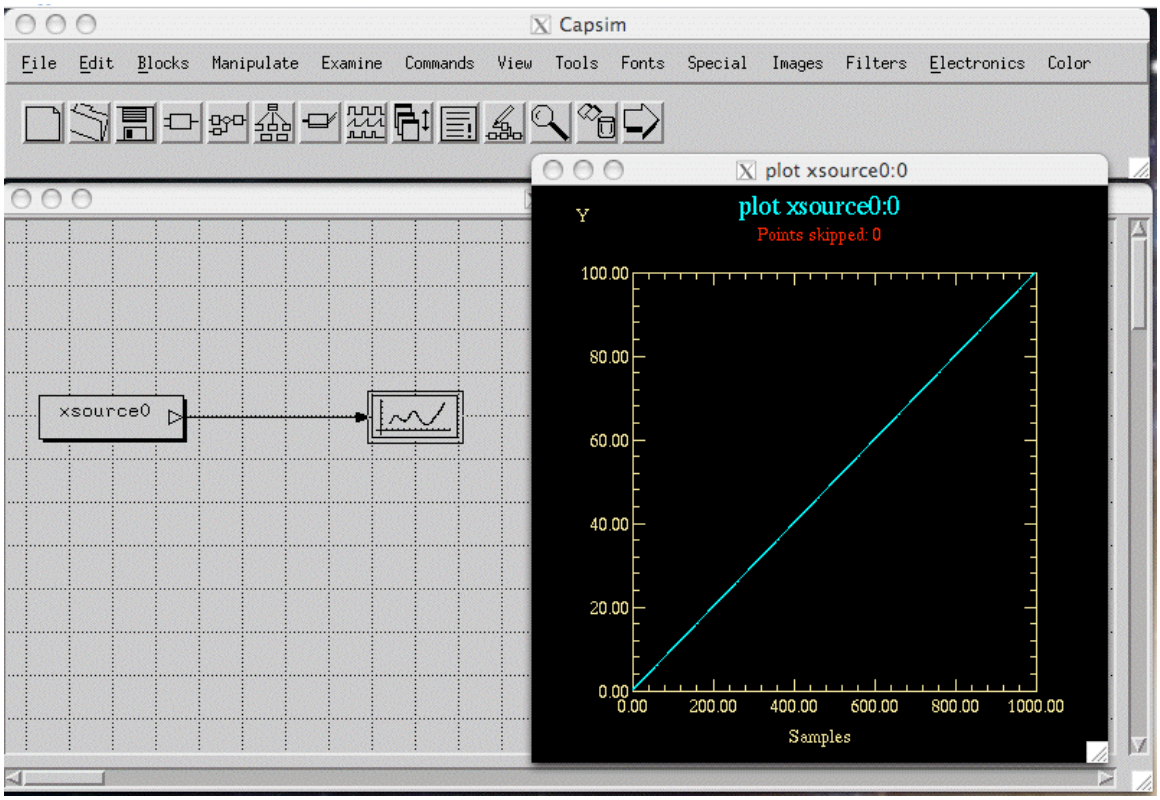

Figure 16 Simulating the xsource Block

Creating a Process Block

In this section we will generate a Processing Block. That is, a block that inputs samples, processes them, and outputs the processed samples. To generate the Processor block bring up the block code generation tool by changing directories to the BLOCKS directory. Then type

```
wish $CAPSIM/TOOLS/blockgen.tcl
```

The block generator window is displayed. Set the Block Type to “Processor”. Set the Buffer Type to “float”. Set the Block Name to “xprocess”. See Figure 17.

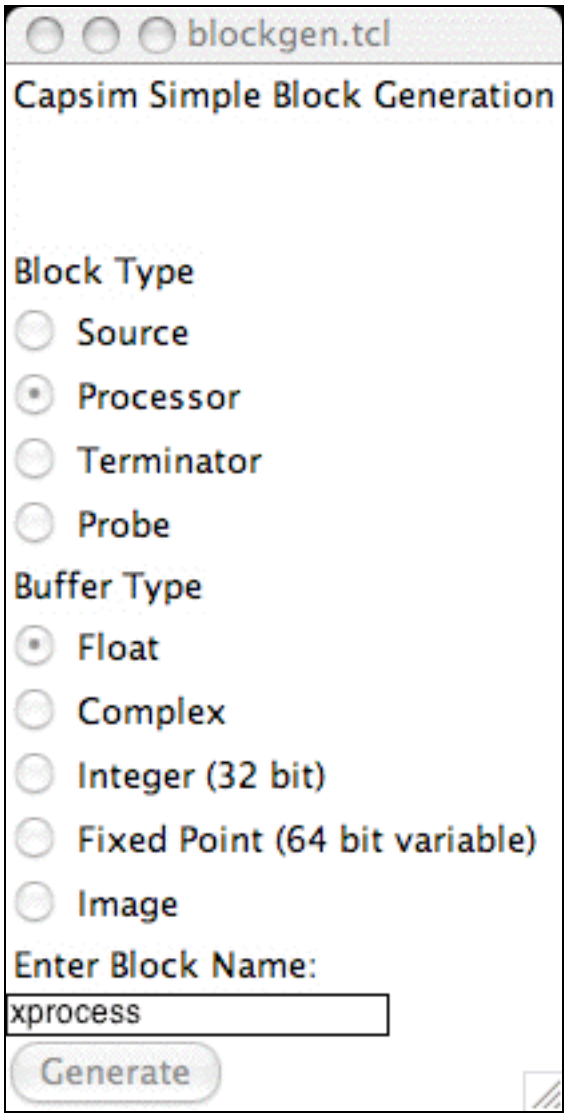

Figure 17 Block Generation Settings for *xprocess.s*

The file "xprocess.s" will be generated with code for a processor block. Edit this file and review the code. In Figure 18, we show the code for the MAIN_CODE tag. Note that the output sample is set to the value of the input sample. We will change the code so that the *xprocess.s* block squares its input sample and outputs the squared values. Make the changes shown in Figure 19, save the file and cd to the WORK directory (cd ..) and type make TWICE (since we are adding a new block).

```
<MAIN_CODE>
<![CDATA[

while(IT_IN(0)) {
       sample=x(0);

         /*
          * ready output buffer for sample
          * check for overflow
          */
         if(IT_OUT(0)) {
                        KrnOverflow("xprocess",0);
                        return(99);
         }
         /*
          * output the sample
          */
         y(0)=sample;


}

]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[

]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure 18 Process Block Main Code

```
<MAIN_CODE>
<![CDATA[

while(IT_IN(0)) {
        sample=x(0);

          /*
           * ready output buffer for sample
           * check for overflow
           */
          if(IT_OUT(0)) {
                          KrnOverflow("xprocess",0);
                          return(99);
          }
          /*
           * output the sample
           */
          y(0)=sample*sample/10.0;


}

]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[

]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure 19 Process Block Modified C Code

After *capsim* exceutable has been successfully built, run CAPSIM and open the previous topology "test_xsource.t". Figure 20 shows the topology. We will insert the *xprocess* block between the *xsource* and *plot* blocks. To do this, we need to move the *plot* block to the right to create some room. See Figure 21.

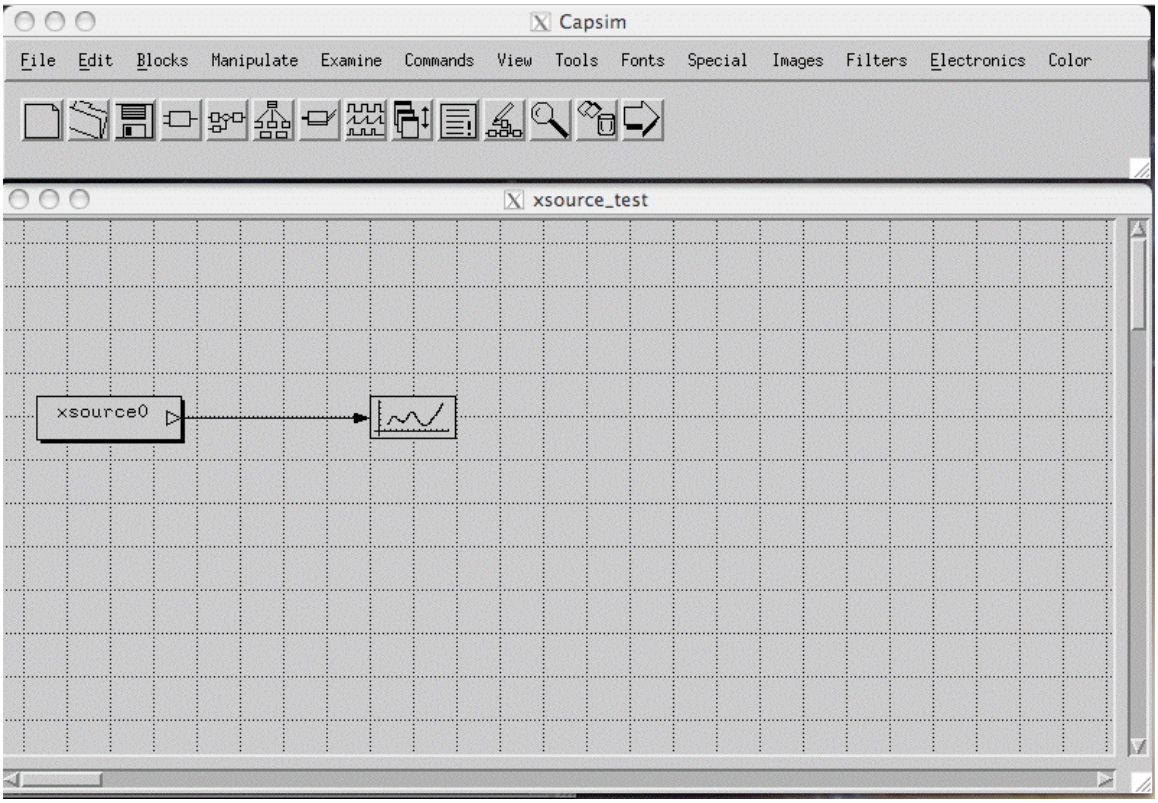

Figure 20 Opening *test_xsource*

Next click on the connection between the two blocks. The Connection Popup will appear. Select Insert. A sub menu will appear. Select Block. A list of blocks will appear. Scroll down to the *xprocess* block. (If it does not appear you need to type make again and look for possible errors in compilation). Click on the Place button, The *xprocess* block will be inserted between the two blocks as show in Figure 24. Run the simulation. You should get a plot of a parabola (square of the linear curve with *xsource* alone).

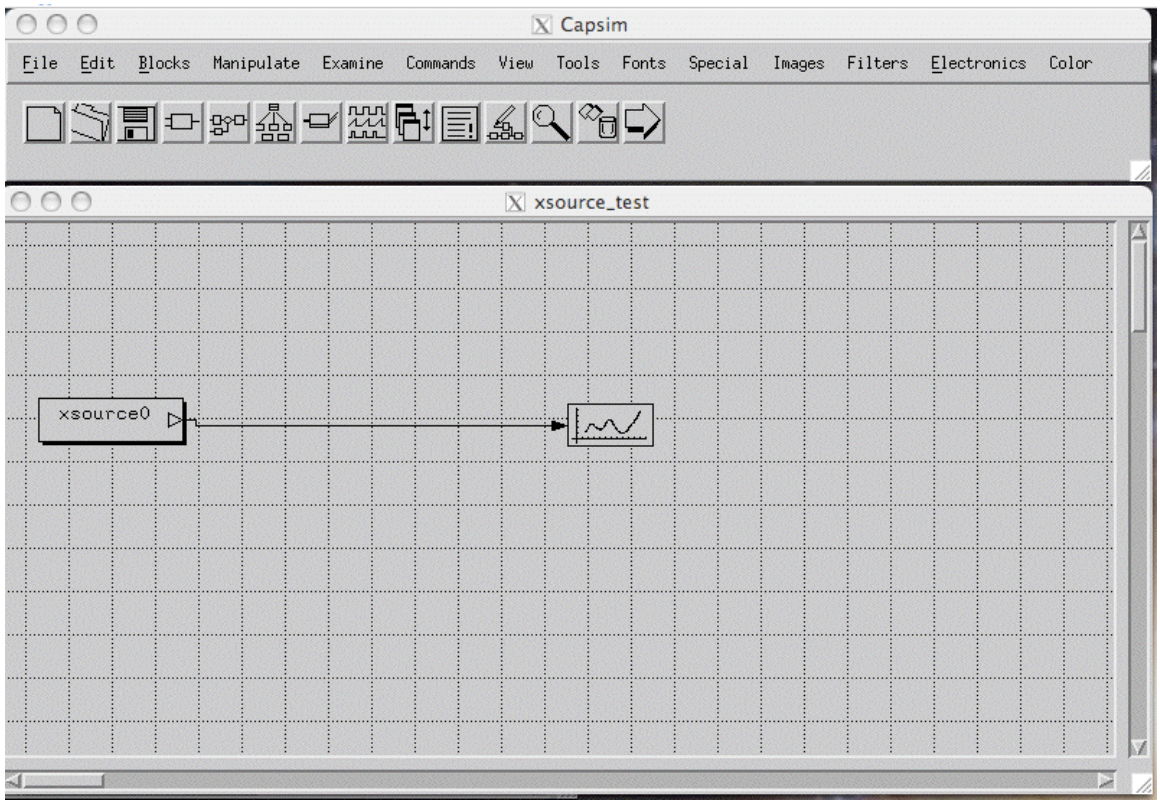

Figure 21 Moving the plot Block

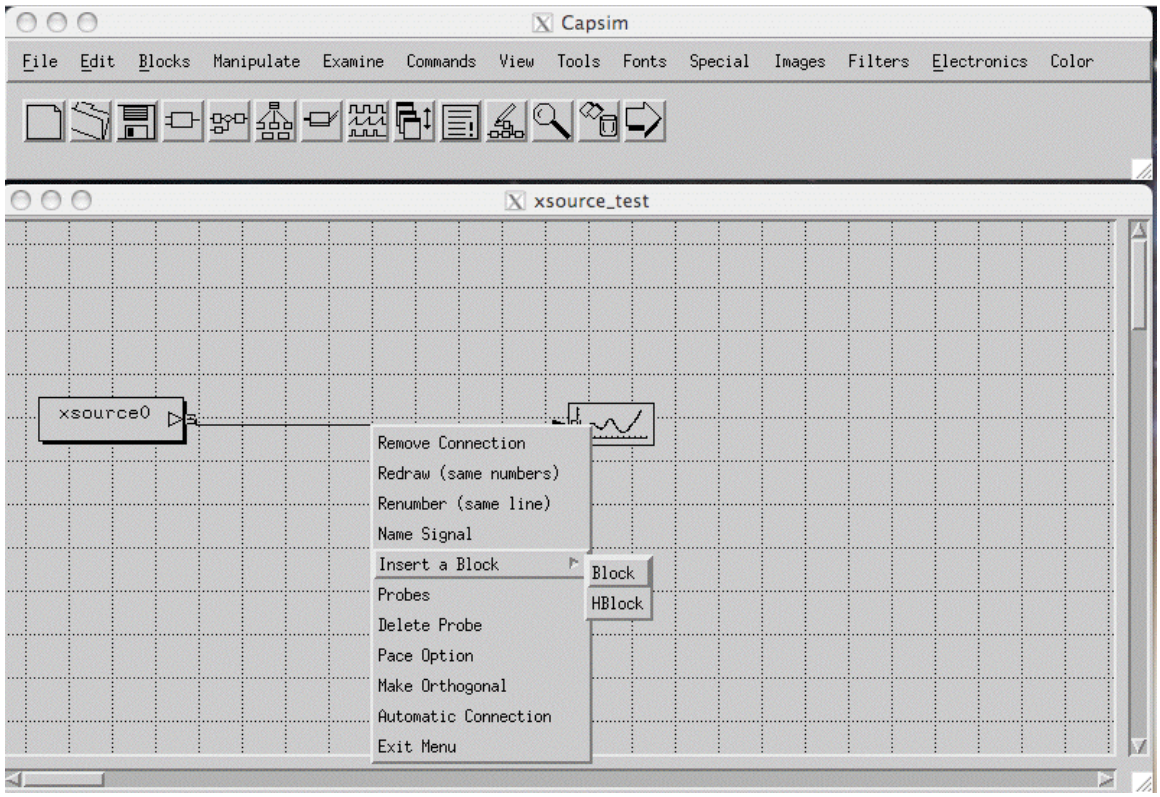

Figure 22 Selecting Connection and Selecting Insert from Popup Menu

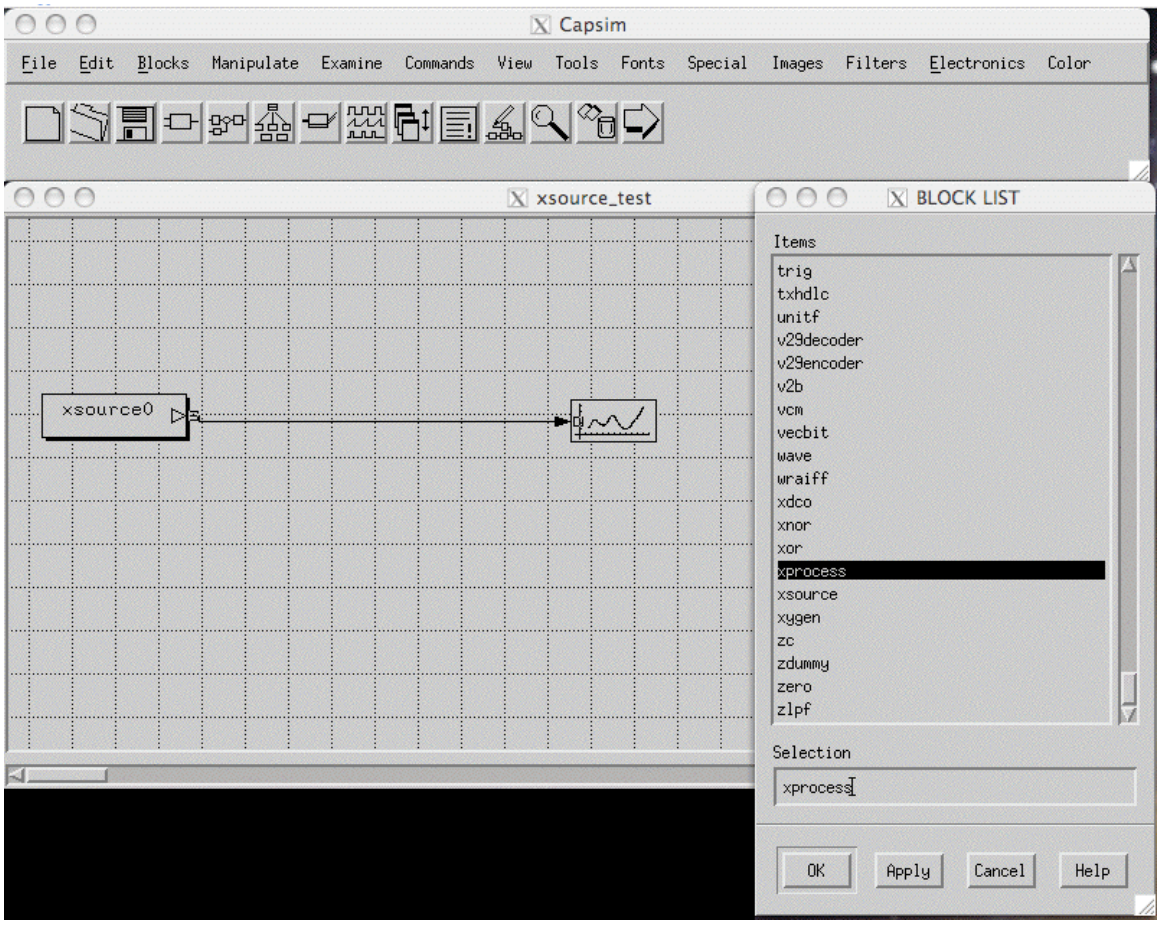

Figure 23 Insert Block List with xprocess Selected

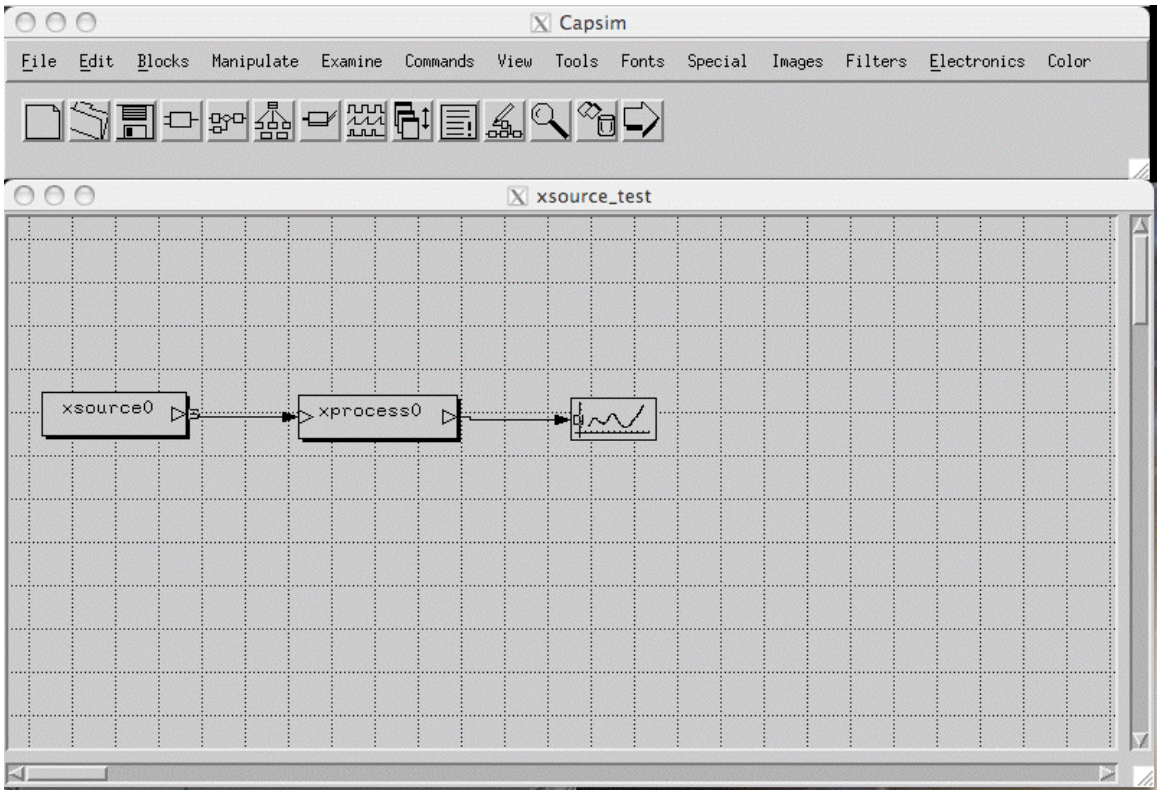

Figure 24 Block Inserted

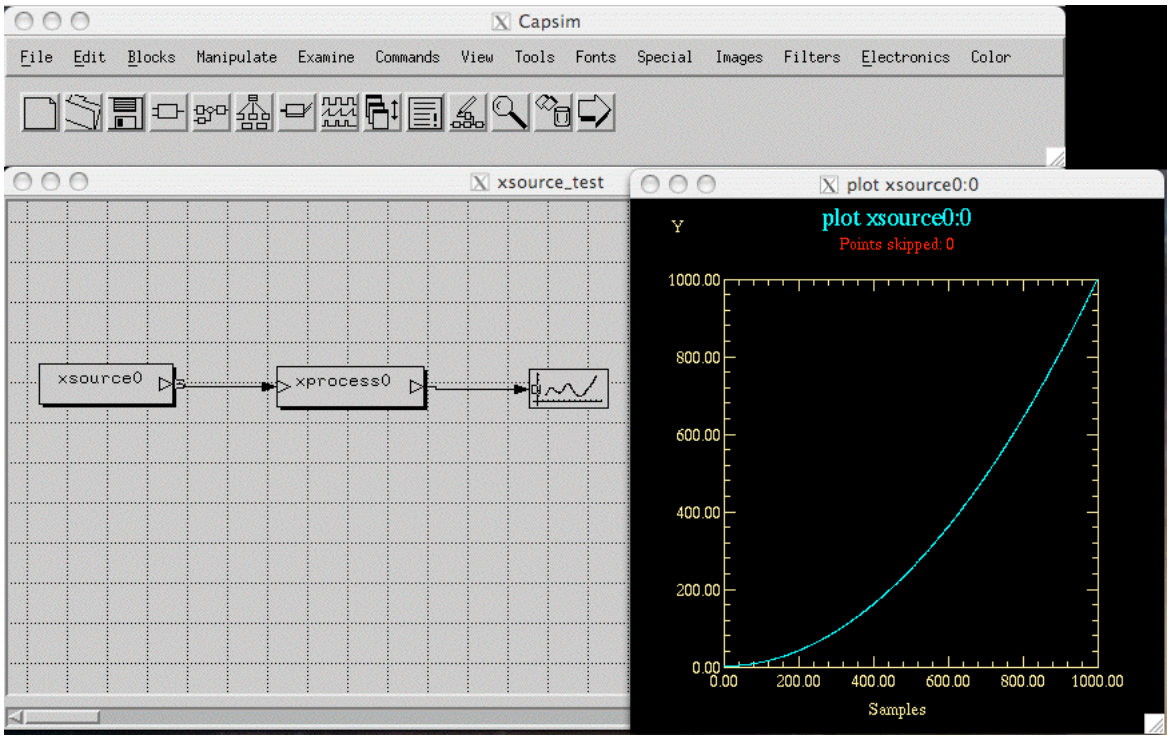

Figure 25 Running Simulation with xsource and xprocess

Compilation Errors

In this section we will add an undefined variable ZZZ to the MAIN_CODE section of *xprocess.s* and examine the compilation errors that result. Figure 26 shows the MAIN_CODE of *xprocess.s* with the undefined variable ZZZ inserted. If we *cd* .. to WORK and type *make* in the WORK directory we get compilation errors as shown in Figure 27.

```
while(IT_IN(0)) {
        sample=x(0);

          /*
           * ready output buffer for sample
           * check for overflow
           */
          if(IT_OUT(0)) {
                          KrnOverflow("xprocess",0);
                          return(99);
          }
          /*
           * output the sample
           */
          y(0)=sample*sample/10.0*ZZZ;


}

]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[

]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure 26 *xprocess.s* with Undefined Variable *ZZZ* Inserted

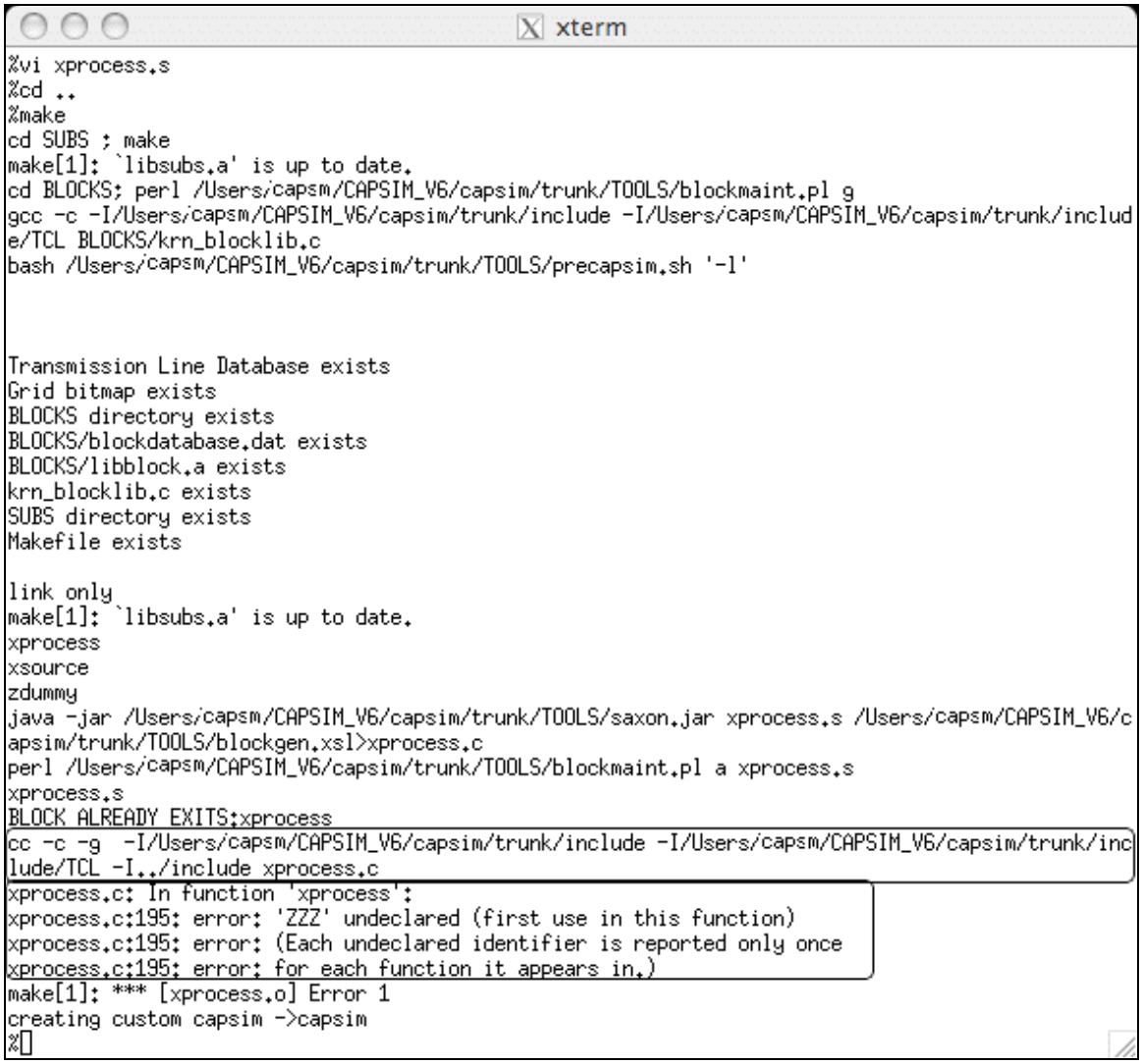

Figure 27 Compile with Error Messages using Make in WORK Directory

We can check for block compilation errors in the BLOCKS directory by typing:

```
make –f blocks.mak
```

in the BLOCKS directory. Thus it is straight forward to detect and fix compilation errors in the BLOCKS directory. After all updates to blocks type make in the WORK directory.

Block Database Utility

A useful utility is the Block Database Management Utility. In this section we will only describe a few of its capabilities. Change directory to the BLOCKS directory and type the following command:

```
wish $CAPSIM/BLOCKS/blocksmanage.tcl
```

The Block Management Utility will appear as shown in Figure 28.

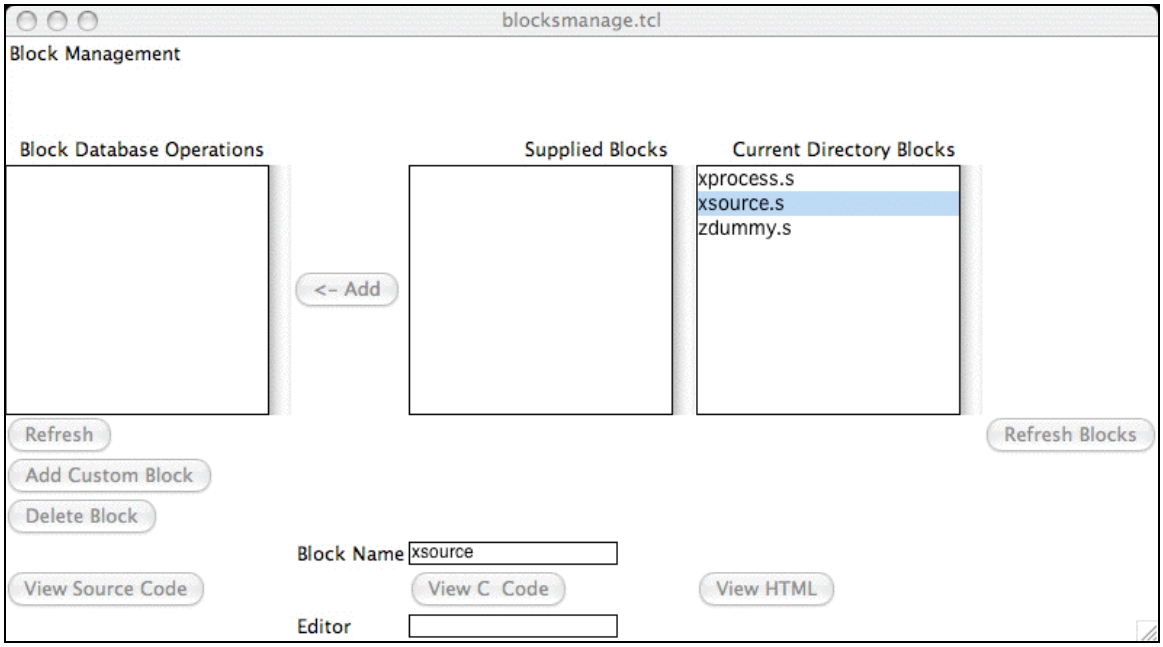

Figure 28 Block Management Utility

For now, only the blocks in the current BLOCKS directory are shown. The Supplied Blocks are blank. For more advanced operation this list will be populated with the supplied blocks. The three buttons:

1- View Source Code
2- View C Code
3- View HTML

are of interest. To view the source select the block and click on View Source Code. You can change the viewing editor by modifying the TCL source code for *blocksmanage.tcl.* For example you can use JEDIT etc. Keep a backup of *blocksmanage.tcl* if you make this modification.

If you click on ViewHTML the HTML code for the selected block will be generated. For example, *xsource.htm.* To view it, open the BLOCKS directory from Windows Explorer and drop *xsource.htm* on FIREFOX or another Browser.

It is instructive to view the generated C code for the block. Note the STATES and PARAMETER data structures and the defines for parameters and states.

Adding Subroutines

Add any subroutine C code to the SUBS directory. The Makefile will be run from the Makefile in the WORK directory and the subroutine will be compiled and added to *libsub.a* and linked into CAPSIM. Or just type make in the SUBS directory.

If you need to link into a library (for example a numerical package) you need to add the library to the *precapsim.sh* bash shell in the $CAPSIM/TOOLS directory. Make a backup of *precapsim.sh*. Then go to the end and insert the name and location of the library to the link path.

Thus you can develop blocks that interact with acquisition and playback instruments.