Custom CAPSIM®

Developing and Adding Models to Capsim

Capsim Buffers

Creating New Buffer Types

Silicon DSP Corporation

http://www.silicondsp.com

Table of Contents

Custom Capsim®

1 Introduction

One of the most important features of Capsim is that users can add their own models, written in C code, to create a custom version of Capsim. The models can then be used within simulations just like models supplied with Capsim. In addition, users can use all of Capsim's facilities, such as parameters, probes, etc. to test and evaluate a model as it is being developed. Since Capsim is supplied with a variety of blocks, a user can easily find a block that is close to the functionality that they desire. They can then use it as a template and rapidly incorporate their model into Capsim. Capsim is also supplied with TK/TCL tools that generate block code using a graphical interface.

This manual gives a detailed description of the procedures used to create a variety of blocks. There are blocks that can have an arbitrary number of inputs and outputs. Some blocks need only one input and one output. Probe blocks pass all input samples to output buffers if they exist. Some blocks may output diagnostic data to a buffer if it is connected. The examples provided will quickly launch you into writing models for Capsim to suite your requirements.

We will also describe the various buffer types supported in Capsim including how you can add your own buffer type. For example, you may wish to send data packets instead of floating point numbers between blocks. You may send pointers to images for example. Furthermore, blocks can be written such that they can operate on selectable buffer types.

For example, the spectrum probe should be able to accept fixed point, floating point, or complex data.

Finally, we will describe issues associated with buffers and their growth. With Capsim version 3.5 you can specify the number of cells allocated to a buffer. This is called a segment. This has certain implications. If this number is too small, then simulations may run slower, yet use less memory. If it is too large, then memory is wasted. An ideal value will speedup the simulation while keeping memory usage low. Furthermore, another parameter controls the number of segments used in buffers before the simulation stops due to excessive memory use. Buffers may grow very large in some multi-rate simulations. This growth can be eliminated by following rules to be described. Capsim is one of the most efficient simulation environments for multirate signal processing. Buffer growth can be controlled and bounded using pacers described in a Capsim application note.

2 Writing Models For Capsim

2.1 Introduction

User block functions, in addition to implementing the functionality of a simulation, must do a number of things to effect the interface to Capsim, "including accepting arguments, allocating storage for state variables, checking the size of parameter storage and the number of input and output buffers, defaulting parameter values and initializing state variables, etc."[1] *precapsim.sh* is a script that provides for the adding of blocks to the Capsim library. *precapsim* takes the source code of a block and adds it to the Capsim library, creating a new personalized version of Capsim. In Capsim, we combine *precapsim* and the UNIX *make* facility along with PERL and Java to automate the whole process.

This chapter will outline what is involved in writing a block and then adding it to the library. In order to write a block the user must use the *blockgen.xsl* XSLT script which is processed by the Java program

(1) D. J. Hait and D. G. Messerschmitt, *BLOSIM Reference Manual,* UC Berkeley

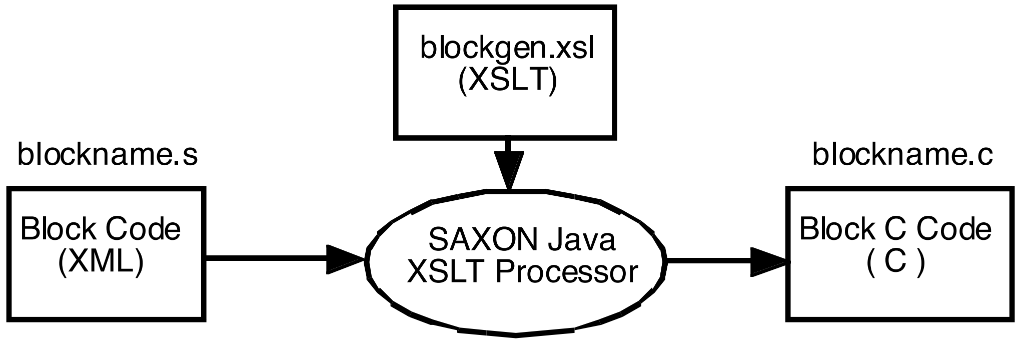
saxon.jar to produce the block C code from the block XML code. This chapter includes some sections from the original paper by Messerschmitt.

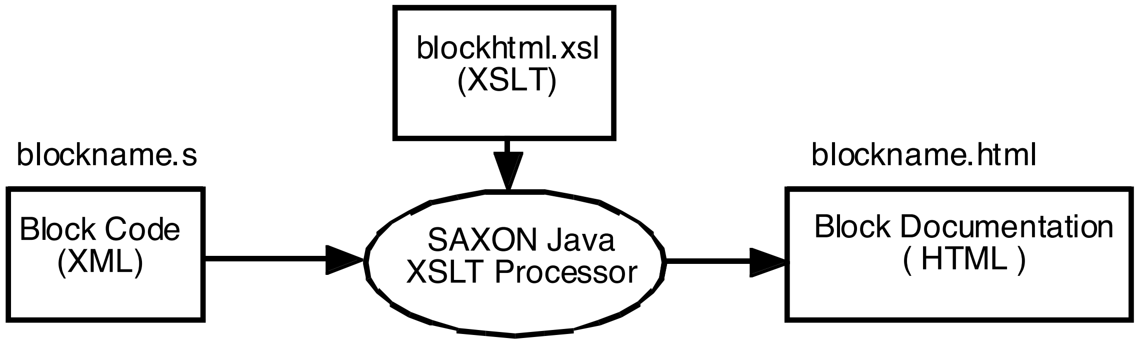
In Capsim V6 all blocks are written in XML with embedded C code. Tools are provided to generate HTML documentation and C code from the XML code.

2.2 XSLT *blockgen.xsl* Input and Output Files

Blockgen.xsl is an XSLT[2] (Extensible Stylesheet Language for Transformations) preprocessor for user block functions which allows the user to express many of the interface parameters in a form similar to and compatible with the topology specification, and then turns this specification into C code for compilation and execution. In the following sections a number of XML syntax rules will be discussed. Users do not need to be familiar with XSLT.

The file provided to *blockgen.xsl* must have a ".s" postfix; that is, be of the form "name.s". *blockgen.xsl* generates a C program in a file called "name.c" which contains the C function *name*() which is the corresponding user BLOCK function.





[2] Doug Tidwell, *XSLT*, O'Reilly, 2001

Figure 1

2.3 Block XML Structure

Blocks in Capsim are written in XML and are parsed by XSLT processor and transformed to C code. Each block has the following sections:

1- **<BLOCK>** and **</BLOCK>** which is the root element.
2- **<LICENSE>** This section contains the text of the block license. For example the GNU public license.
3- **<BLOCK_NAME>** Contains the name of the block. The generated C code function name is defined here.
4- **< DESC_SHORT >** A short description of the block. This will be used in the HTML documentation of the block.
5- **<COMMENTS>** Comments will appear in the generated C code.
6- **<INCLUDES>** Contains all #includes that are included in the generated C code.
7- **<DEFINES>** Contains all the #defines that are included in the generated C code.
8- **<PARAMETERS>** Contains the definition of the block parameters.
9- **<STATES>** Contains the definition of state variables for the block .
10- **<INPUT_BUFFERS>** The definition of input buffers if present.
11- **<OUTPUT_BUFFERS>** The definition of output buffers if present.
12- **<DECLARATIONS>** C declarations that are copied *as is* to the generated C code.
13- **<INIT_CODE>** This is the user initialization code. It is C code that is executed once. For example to allocate space, open files (with pointers that are states), to design filters base on parameters etc.
14- **<MAIN_CODE>** This is the C code that is executed as long as samples are available to process in the buffers. This is is the main processing code.
15- **<WRAPUP_CODE>** This is the C code that is executed once when the simulation is done (for example to store results in a file, close file pointers, free up memory).

Since XML is ASCII text it is easy to edit the XML block code directly to make changes to blocks. There are graphical tools available in Capsim that automatically generate the XML code for custom blocks.

The block structure which is enforced in Capsim greatly enhances the readability and understanding of the blocks. The modular structure allows common interface and re-usability. In addition XML allows for the block code to be transformed to C, HTML or even Java and SystemC code.

2.4 #Include and #Define

All includes must be with the XML tag <INCLUDES>:

```
<INCLUDES>
<![CDATA[

#include <math.h>
#include <stdio.h>
#include "someheader.h"

]]>
</INCLUDES>
```

By enclosing the includes within “<![CDATA[“ and “]]> “ the “<” and “>” do not cause a problem for the XML parser. The Tk/TCL Block generator graphical interface automatically generates the XML tags.

Defines are added to the XML tags <DEFINES>:

```
<DEFINES>

#define PI 3.1415926535898
#define PEAK_WINDOW 16

</DEFINES>
```

Note if the #defines include the “<”and “>” symbols enclose them with “<![CDATA[“ and “]]> “.

The files <*stdio.h*> and <*math.h*> are automatically included by *blockgen.xsl*, and hence do not need to be included by the user.

2.5 The C Program

The user must of course provide, in the input file (".s" file), the C program which implements the functionality of the user BLOCK function. This C program generally includes four parts:

1. Declarations of variables.

2. Initialization code which is to be executed the first time the user BLOCK routine is called by CAPSIM.

3. The main body of code which is executed every time the user BLOCK function is called. This code accesses the buffers and processes the samples.

4. Wrapup code which is executed only when the simulation has finished.

Since *blockgen.xsl* automatically generates the initialization code for checking the number of buffers, checking the number of parameters and their types, defaulting the parameters, and allocating storage for state variables and initializing them, many user BLOCK routines will not require any additional initialization code or wrapup code

blockgen.xsl recognizes the following types of code because they are imbedded between XML tags.

In particular, declarations of variables are imbedded between the lines:

```
<DECLARATIONS>
</DECLARATIONS>
```

State variables are declared with in the tags:

```
<STATES>
</STATES>
```

The parameters that are used to control the program are designated in the tags:

```
<PARAMETERS>
</PARAMETERS>
```

The buffers used to communicate between blocks are defined in the sections:

```
<INPUT_BUFFERS>
</INPUT_BUFFERS>

<OUTPUT_BUFFERS>
</OUTPUT_BUFFERS>
```

The initialization code is imbedded between the lines:

```
<INIT_CODE>
</INIT_CODE>
```

The main code is placed between the lines:

```
<MAIN_CODE>
</MAIN_CODE>
```

The wrapup code is imbedded between the lines:

```
<WRAPUP_CODE>
</WRAPUP_CODE>
```

We will now discuss the each of these sections.

2.6 Types of Variables in Blocks

CAPSIM works by calling the blocks in a given simulation one at a time, processing any data on the block's buffer and then calling on the next block. For this reason there are two types of variables available: declarations and states. The difference between these two types of variables is that a state's value is saved between calls of the block. While a declaration's value is not saved between calls.

Declarations are often used to store temporary values such as counters or dummy variables. A typical section of code would look like:

```
<DECLARATIONS>
     type variable_name
</DECLARATIONS>
```

Where *type* is any standard C type such as **int, float, FILE, char, etc**. *Variable_name* is the name of the variable. You may declare as many variables as you need in this section. The declarations look exactly as they would in a C program.

Examples of declarations are:

```
<DECLARATIONS>
      float temp, dummy;
      float zz;
      int i, j;
      char name;
</DECLARATIONS>
```

The state variables are enclosed in the following tags:

```
<STATES>
</STATES>
```

Each individual state variable is enclosed in the tags,

```
<STATE>
</STATE>
```

Each state has a *name*, a *type*, and a *default* value:

```
     <STATES>

          <STATE>
                    <TYPE> complex* </TYPE>
                    <NAME> h_P </NAME>
          </STATE>
          <STATE>
                    <TYPE> int </TYPE>
                    <NAME> limit </NAME>
                    <VALUE> 100 </VALUE>
          </STATE>
          <STATE>
                    <TYPE> FILE* </TYPE>
                    <NAME> fp </NAME>
```

```
            </STATE>
```

`</STATES>`

For each state element the state `<NAME>` is an identifier for the state variable which will be used in the program, and `<TYPE>` is the type of the variable (float, int, FILE*, float*, complex* etc.) which must not contain any blanks (note the absence of blanks between FILE and * for instance), and `<VALUE>` is the initial value the state variable is to assume, which can be either a constant or the name of a parameter.

2.7 Passing Parameters

The only parameters which can be passed are integers, floating point numbers, strings and arrays.

Parameters are passed to the BLOCK by including their names and types and default values in XML tags enclosed in the XML parameter tags:

`<PARAMETERS>`
`</PARAMETERS>`

Individual parameters are defined by the tags:

`<PARAMETER>`
`</PARAMETER>`

Each parameter must have a name and type with optional definition and default value. An example follows:

```
      <PARAMETERS>
            <PARAM>
                  <DEF>   Number   of   samples   in   sequence
      </DEF>
                  <TYPE> int </TYPE>
                  <NAME> N </NAME>
                  <VALUE> 64 </VALUE>
            </PARAM>
            <PARAM>
                  <DEF> File name containing sequence</DEF>
                  <TYPE> file </TYPE>
                  <NAME> filename </NAME>
                  <VALUE>  </VALUE>
            </PARAM>
            <PARAM>
                  <DEF> Ratio threshold </DEF>
                  <TYPE> float </TYPE>
                  <NAME> ratioThreshold </NAME>
                  <VALUE> 5.0 </VALUE>
            </PARAM>
            <PARAM>
                  <DEF> Array of weights </DEF>
                  <TYPE> array </TYPE>
                  <NAME> myarray </NAME>
            </PARAM>

      </PARAMETERS>
```

`<TYPE>` is the identification of the type of parameter. There are presently five possibilities for *type* which are specifically **int, float, file, function, string** and **array.** Parameter `<NAME>` is the identifier of the parameter as it is referred to in the initialization and main C code. Default_`<VALUE>` is a constant which is the value of the parameter if it is not set by the user in the topology description. Note that the **array** parameter type does not support default values. If you use a parameter array, the size and initial values must be set in the topology file. (This is enforced by the Capsim program.)

In the above example, for the first case the parameter is not given a default value (in this example) and hence must be specified in the topology routine. The user will be prompted with the string "Enter the number of samples in sequence" upon typing the *chp* command. In the last case with **arrays**, note that no "brackets" or dimensioning should be used, and no default values can be given. This must be done in the topology file. The array values can be changed freely within the block program, however. Also, an integer variable containing the size of the array will be **automatically** declared for use within the block program. Its name will be assigned as the array name with prefix "n_"; thus in the example above, an integer "n_myarray" will be declared and initialized with the size of "myarray".

2.8 Specifying Input and Output Buffer Names

The input and output buffers are a block's only connection with the rest of the blocks in a simulation. A block may have any number of input and output buffers, from zero on up. The buffers are specified in a special section of XML code:

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


<OUTPUT_BUFFERS>
      <BUFFER>
            <TYPE> float </TYPE>
            <NAME> y </NAME>
      </BUFFER>
      <BUFFER>
            <TYPE> float </TYPE>
            <NAME> z </NAME>
      </BUFFER>
</OUTPUT_BUFFERS>
```

In the above example the block has a floating point input buffer “x” and two floating point output buffers “y” and “z”. `<TYPE>` is the type of variable, either **float** , **int, complex** or a custom buffer type**.** `<NAME>` is the name of the buffer as referred to in the main program.

```
<INPUT_BUFFERS>
      <BUFFER>
           <TYPE> float </TYPE>
           <NAME>  x </NAME>
           <DELAY>
                  <TYPE>max</TYPE>
                  <VALUE_MAX> delay_value </VALUE_MAX>
           </DELAY>
           <DELAY>
                  <TYPE>min</TYPE>
                  <VALUE_MIN> delay_value </VALUE_MIN>
            </DELAY>
      </BUFFER>

</INPUT_BUFFERS>

<OUTPUT_BUFFERS>
     <BUFFER>
          <DELAY>
               <TYPE>max</TYPE>
               <VALUE_MAX> delay_value </VALUE_MAX>
         </DELAY>
         <TYPE> float </TYPE>
         <NAME>  y </NAME>
     </BUFFER>
```

`</OUTPUT_BUFFERS>`

In these statements, *delay_value* is either an integer constant, or an integer variable which is an input parameter. `<TYPE>` is the type of variable, either **float** or **int,** and `<NAME>` is the name of the buffer as referred to in the main program. For `<DELAY>` of `<TYPE>` **min** it is the number of samples in the past that a block considers the present. This allows the user to build a delay into the buffer structure (as opposed to using a delay block). The default is zero. For `<DELAY>` of `<TYPE>` **max** it is the largest number of samples that a block can access into the past. The default for an input buffer is **delay_min**. The default for an output buffer is zero.

An explanation of how to access the buffers in the programming of a block are given in the next section.

Note that the delay statements are unnecessary if only current buffer samples are referenced. (See next section's descriptions of *set_dmin_in* etc.) If used, these statements must be placed just prior to their respective buffer name declaration.

2.9 Available Functions in User BLOCK Code

The following functions are available in writing your own blocks. These functions involve the handling of the input and output buffers. These functions may appear in the initialization code, the main code, or the wrapup code.

AVAIl(buffer_no) returns the number of samples to be read on input buffer *buffer_no*. The buffers are numbered in the order of their declaration. The first buffer is 0, the second buffer is 1, etc.

MIN_AVAIL() returns the minimum number of samples available on all the input buffers. This statement performs an ***AVAIL(buffer_no)*** on each buffer and returns the smallest of these values.

IT_IN(buffer_no) increments time on the input buffer number *buffer_no* and returns the number of samples that were available in the buffer before the increment. If it was not possible to increment time (no more samples available), then *IT_IN(buffer_no)* does nothing and returns 0. Note that time 0 is always the present, time 1 is the previous sample, etc.

IT_OUT(buffer_no) increments time on output buffer number *buffer_no* and returns a 1 if the buffer has overflowed. Specifically, *IT_OUT()* allocates a new sample on the output buffer, which becomes the sample with delay 0. Buffer overflow indicates that the maximum number of segments (each segment 128 cells by default) has been exceeded. You should check *IT_OUT*() and if it is true you should return a non zero error code. Use the following code:

```
        if(IT_OUT(bufferNumber)) {
                KrnOverflow("blockname",bufferNumber);
                return(99);
        }
```

You can actually continue processing by returning a zero so that no more samples are placed on the buffer. In this case save the sample to be outputed and output it the next time the block is called. By then, the other blocks will have consumed the samples in the buffer and space will be available to place the previous sample. Note that this technique will only work if *all* blocks are written this way. Otherwise, just return the error code 99 along with a call to KrnOverflow to report an error message. You can either increase the maximum segments using the Capsim line command:

setmaxseg MAX_SEGMENTS

where MAX_SEGMENTS is 1000 by default. Or, you can be wise and use pacers or other techniques to prevent buffer overflow by proper design. Buffer overflow only occurs when mult-rate sampling is used.

NO_INPUT_BUFFERS() and ***NO_OUTPUT_BUFFERS()*** return respectively the number of input and output buffers connected to the BLOCK by the topology definition.

Normally the user refers to input and output buffers by name, as if they are floating point numbers. Thus, if "sample" is the name of an output buffer, then the last output sample would be referred to as *sample(0)* and the output sample *delay* in the past would be referred to as *sample(delay)*. A new output sample would be generated by calling *IT_OUT(buffer_no)* and then referring to the new sample being generated as *sample(0)*. Similarly, the output sample *delay* samples in the past would be referred to as *sample(delay)*.

However, in the case of buffers which do not store floating values, or where there are a variable number of buffers, the following routines are available to manipulate buffers.

POUT(buffer_no,delay) returns a pointer to the sample *delay* in the past for output buffer number *buffer_no*.

PIN(buffer_no,delay) returns a pointer to the sample *delay* in the past for the input buffer number *buffer_no*.

OUTF(buffer_no,delay) and ***inf(buffer_no,delay)*** return not a pointer to a sample, but rather actual floating point values. They of course assume the buffer is passing floating point numbers.

OUTI(buffer_no,delay) and ***INI(buffer_no,delay)*** return not a pointer to a sample, but rather actual integer values. They of course assume the buffer is passing integer numbers.

SET_DMIN_IN(buffer_no,delay_min) is a routine which sets the minimum delay relative to the current input sample for the input buffer number *buffer_no* to the value *delay_min* an integer constant or parameter name. The value of this minimum delay defaults to zero, so if the BLOCK will access the current sample it is unnecessary to call this function.

SET_DMAX_IN(buffer_no,delay_max) and ***SET_DMAX_OUT(buffer_no,delay_max)*** set respectively the maximum delay relative to the current sample of an input buffer and the maximum delay relative to the latest output sample for an output buffer with which the buffer is accessed to *delay_max* which is an integer constant or parameter name. The maximum delay, in the case of an input buffer, defaults to be the same as the minimum delay. In the case of an output buffer, it defaults to zero.

SET_CELLSIZE_IN(buffer_no,size) and ***SET_CELLSIZE_OUT(buffer_no,size)*** are routines which set the size of one cell in the buffer to a fixed integer This value defaults to the appropriate size for a floating value, and does not need to be set for that case. But for example if the buffer transmits integer values, then *size* would be replaced by *sizeof(int)*.

SNAME(buffer_no)
This macro returns a string which contains the signal name associated with the input buffer. This is useful in probe blocks, which may use the string to identify a plot or a legend in a plot.

Now that we have seen all the functions used in developing a block lets take a look at a few examples.

2.10 Example Block Development

2.10.1 Introduction

This section will discuss what is involved in writing your own blocks for CAPSIM. It will proceed by presenting three blocks: *convolve.s, node.s* and *add.s*. Each of these blocks will bring out different aspects of block programming.

2.10.2 A Single Input, Single Output Block (convolve.s)

If *convolve.s* does not exist in your library, it will be beneficial for you to either type it in or obtain a copy of it. The source for *convolve.s* is shown in Fig. 3. In this figure certain sections of the code are highlighted for purpose of discussion, they do not actually appear this way in the block file.

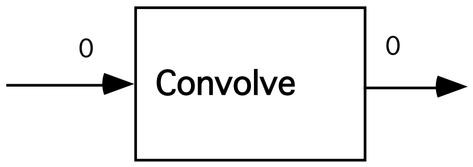


Figure 2 Convolve Block

```
<BLOCK>
<LICENSE>
/*  Capsim (r) Text Mode Kernel (TMK) Star Library (Blocks)
    Copyright (C) 1989-2002  XCAD Corporation

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later
version.

    This library is distributed in the hope that it will be
useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General
Public
    License along with this library; if not, write to the Free
Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307  USA

    http://www.xcad.com
    XCAD Corporation
    Raleigh, North Carolina */
</LICENSE>
<BLOCK_NAME> convolve  </BLOCK_NAME>

<COMMENTS>
<![CDATA[

/* convolve.s */
/*****************************************************************
******
                             convolve()
******************************************************************
******
This star convolves the input samples with the impulse response
(finite
duration, FIR ) given in a file.
Param: 1 - (file) File with the impulse response samples
       2 - (int) N  number of samples in the impulse response.

This star convolves the input samples with the impulse response
(finite
duration, FIR ) given in a file.
Param: 1 - (file) File with the impulse response samples
       2 - (int) N  number of samples in the impulse response.


Date:  September 23, 1988
Programmer: Adali Tulay
*/

]]>
</COMMENTS>

<DESC_SHORT>
This star convolves the input samples with the impulse response
(finite duration, FIR ) given in a file.
</DESC_SHORT>

<INCLUDES>
<![CDATA[

#include <math.h>
#include <stdio.h>

]]>
</INCLUDES>

<DEFINES>

#define PI 3.1415926

</DEFINES>



<STATES>
       <STATE>
              <TYPE> float* </TYPE>
              <NAME> x_P </NAME>
       </STATE>
       <STATE>
              <TYPE> float* </TYPE>
              <NAME> h_P </NAME>
       </STATE>
</STATES>

<DECLARATIONS>

       int i;
       int j;
       float tmp1,tmp2;
        float sum;
       FILE *fopen();
       FILE *imp_F;

</DECLARATIONS>



<PARAMETERS>
<PARAM>
       <DEF>File name containing impulse response samples</DEF>
       <TYPE> file </TYPE>
       <NAME> filename </NAME>
       <VALUE></VALUE>
</PARAM>
<PARAM>
       <DEF>Order of impulse response</DEF>
       <TYPE> int </TYPE>
       <NAME> N </NAME>
       <VALUE></VALUE>
</PARAM>
</PARAMETERS>
```

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



<OUTPUT_BUFFERS>
       <BUFFER>
              <TYPE> float </TYPE>
              <NAME> y </NAME>
       </BUFFER>
</OUTPUT_BUFFERS>

<INIT_CODE>
<![CDATA[

       /*
        * Allocate memory and return pointers for tapped delay
line x_P and
        * array containing impulse response samples, h_P.
        *
        */
       if( (x_P = (float*)calloc(N,sizeof(float))) == NULL ||
           (h_P = (float*)calloc(N,sizeof(float))) == NULL ) {
              fprintf(stderr,"convolve: can't allocate work
space\n");
              return(4);
       }
       /*
        * open file containing impulse response samples. Check
        * to see if it exists.
        *
        */
        if( (imp_F = fopen(filename,"r")) == NULL) {
              fprintf(stderr,"Convolve could not be opened file was
%s \n",
                             filename);
              return(4);
       }
       /*
        * Read in the impulse response samples into the array
        * and initialize the tapped delay line to zero.
        *
        */
       for (i=0; i<N; i++) {
              x_P[i]= 0.0;
              fscanf(imp_F,"%f",&h_P[i]);
       }

]]>
</INIT_CODE>


<MAIN_CODE>
<![CDATA[

       while(IT_IN(0)){
              /*
               * Shift input sample into tapped delay line
```

```
		  */
		tmp2=x(0);
		for(i=0; i<N; i++) {
			tmp1=x_P[i];
			x_P[i]=tmp2;
			tmp2=tmp1;
		}
		/*
		 * Compute inner product
		 */
		   sum = 0.0;
		for (i=0; i<N; i++) {
		     sum += x_P[i]*h_P[i];
		}
		if(IT_OUT(0)) {
			KrnOverflow("convolve",0);
			return(99);
		}
		/*
		 * set output buffer to response result
		 */
		y(0) = sum;
	}

]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[

	free(x_P); free(h_P);

]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure. 3   Source Code for *convolve.s*

We can see that code is arranged into the sections as described earlier in the chapter. Note that the similarity between this block and a C program.

We will now discuss each section of the code in detail. First several general comments must be made.

The ordering of the parts of the program is not important. BLOCKGEN.XSL interprets the ".s" file by searching the entire code for each set of delimiters, position within the file is of no significance. Now we will discuss the significance of the *convolve.s* block in particular.

The parameters and declarations are similar to the examples given previously in this chapter, however, the form of the state variables requires some discussion. The state variables, *x_P* and *h_P*, are pointers. In the **<INIT_CODE>** memory is dynamically allocated to these pointers using the *calloc* command (this is a standard C command). Dynamic memory allocation allows the block to take up as little memory as possible. This can be very important when running a big simulation. If the memory can not be allocated a **return(4)** is executed. This is a signal to CAPSIM that an error has occurred. A return with an argument different from 0 causes CAPSIM to halt execution of the simulation. Note that a user must flag their own *internal* errors. This error trapping can be seen a second time in the **<INIT_CODE>** when the file is opened. Recall that the **<INIT_CODE>** is only executed the first time the block is called, therefore memory allocation, file reading, and anything else the programmer wants to have occur only once are put in this section. Any memory pointer, file pointer or variable that will be set and used in the MAIN CODE should be declared as a STATE.

In this block, there are only one input and one output buffer, x and y respectively. Recall that the way that samples are accessed: *x(0)* is the current sample, *x(1)* is the most recent past sample, *x(2)* is the one before that, etc. The output buffer functions in an analogous manner.

Now lets discuss the **<MAIN_CODE>**. The first line in the **<MAIN_CODE>**., *while(IT_IN(0))* sets up a loop. Every time that *IT_IN(0)* is called it checks to see how many samples are on the input buffer *x*. If there are no samples then *IT_IN (0)* returns a 0 and the loop is broken. If there are samples to be read then *IT_IN (0)* returns the number of samples on the buffer and then increments the buffer pointer by 1, in effect advancing time.

Once inside the loop the present sample is first saved in a dummy variable, a **<DECLARATION>**, before being shifted into the tapped delay line. Once the tapped delay line has been updated the actual filtering of the sample takes place.

After the filtering is done the block needs to put the filtered sample on its output buffer so that other blocks can access it. This is done in two steps: First, *IT_OUT(0)* is called. This function updates time on the output buffer. Second, the line *y(0) = sum;* moves the filtered value onto the buffer in the new position.

After the block is finished running the **<WRAPUP_CODE>** will be executed once. In the **<WRAPUP_CODE>** for *convolve.s* the space that was allocated in the **<INIT_CODE>** is freed.

The above example is just one way that this block could have been implemented. As an exercise it is suggested that the reader attempt to rewrite this block using **delay_max=N** and the *x* buffer to implement the tapped delay line instead of allocating using the *x_P* array.

2.10.3 Auto Fan-in and Fan-out Blocks

Two important features available in CAPSIM blocks that are not represented in the above block are the ability to have an arbitrary number of outputs, such as in *node.s*, or an arbitrary number of inputs, such as in *add.s*.

The next example will deal with how one goes about creating a block with an arbitrary number of outputs. It will use the code in Fig. 4.

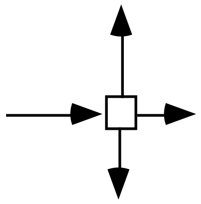


Figure 4 Node block with auto fan-out

```
<BLOCK>
<BLOCK_NAME> node </BLOCK_NAME>

<COMMENTS>
<![CDATA[

/* node.s */
/*****************************************************
                            node()
*****************************************************
Function has a single input buffer, and outputs each input
sample to
an arbitrary number of output buffers.


Function has a single input buffer, and outputs each input
sample to an arbitrary number of output buffers.

*/

]]>
</COMMENTS>
```

```
<DESC_SHORT>
Function has a single input buffer, and outputs each input
sample to an arbitrary number of output buffers.
</DESC_SHORT>

<STATES>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>obufs</NAME>
      </STATE>
</STATES>

<DECLARATIONS>

      int no_samples;
      int i;

</DECLARATIONS>




<INPUT_BUFFERS>
      <BUFFER>
            <TYPE>float</TYPE>
            <NAME>x</NAME>
      </BUFFER>
</INPUT_BUFFERS>

<INIT_CODE>
<![CDATA[

      /* note and store the number of output buffers */
      if((obufs = NO_OUTPUT_BUFFERS() ) <= 0) {
            fprintf(stdout,"node: no output buffers\n");
            return(1); /* no output buffers */
      }

]]>
</INIT_CODE>


<MAIN_CODE>
<![CDATA[



      for(no_samples=MIN_AVAIL();no_samples >0; --
no_samples) {
            IT_IN(0);
            for(i=0;i<obufs;++i) {
                  if(IT_OUT(i)) {
                        KrnOverflow("node",i);
                        return(99);
                  }
                  else
                      OUTF(i,0) = x(0);
```

```
                }
        }

        return(0);  /* input buffer empty */


]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[


]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure. 5 Source Code for *node.s*

This example is included to teach the user how to create a block that has as an arbitrary number of outputs. We see that there is no section called **<OUTPUT_BUFFERS>** in this block, this is because the number of outputs is determined at run time by the number of blocks that this block connects to.

The number of output buffers is determined in the **<INIT_CODE>**. The state variable no_buffers is set equal to the value returned by NO_OUTPUT_BUFFERS(). Recall from section 2.9 that this function returns the number of outputs connected to a block in the topology. A check is also made to see if the block is connected at all. (In a block with a fixed number of output buffers CAPSIM makes this check.)

The **<MAIN_CODE>** simply takes each input value and places it on every output buffer. This block loops through until it clears the input buffer of all available samples each time it is called by using the statement: *while(IT_IN(0))*.

The manner in which each sample is put on every output buffer is through the two statements inside the *for* loop. First, *IT_OUT(buffer_no)* advances time on the output buffer buffer_no. Second, **(float *)POUT(buffer_no,0) = x(0)* is how the output buffer is accessed. Recall from section 2.9 that *POUT()* is a pointer to output buffer number *buffer_no* 0 samples in the past (current). This statement allows us to write the current input sample *x(0)* to this location.

After putting the current input sample on each output buffer the while statement checks to see if there is another input sample. If there is the process is repeated, if not the *return(0)* statement at the bottom of the block is executed.

The last example will deal with the similar capability of having an arbitrary number of input buffers. The add.s block, Fig. 7, will be used to show this feature.

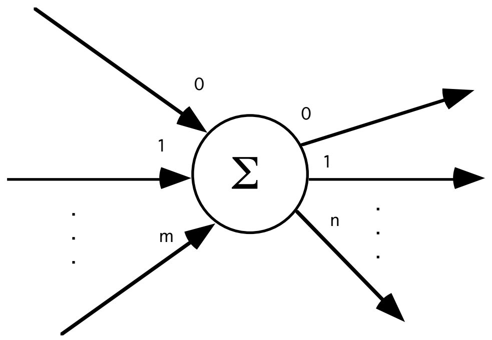


Figure 7. Add Block *add.s*

```
<BLOCK>
<BLOCK_NAME> add </BLOCK_NAME>

<COMMENTS>
<![CDATA[

/* add.s */
/**************************************************
                  add()
**************************************************
Function adds all its input samples to yield an output
sample;
the number of input buffers is arbitrary and determined at
run time.
The number of output buffers is also arbitrary (auto-
fanout).
PROGRAMMERS
Programmer: D.G.Messerschmitt March 7, 1985
Modified: 1/89 ljfaber. add auto-fanout

*/

]]>
</COMMENTS>


<DESC_SHORT>
Adds multiple floating point buffers. Auto fan-in auto fan-
out
</DESC_SHORT>

<STATES>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>ibufs</NAME>
      </STATE>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>obufs</NAME>
      </STATE>
</STATES>

<DECLARATIONS>

      int i,j;
      int samples;
      float sample_out;

</DECLARATIONS>


<INIT_CODE>
<![CDATA[

      /* store as state the number of input/output buffers
*/
```

```
      if((ibufs = NO_INPUT_BUFFERS()) < 1) {
            fprintf(stderr,"add: no input buffers\n");
            return(2);
      }
      if((obufs = NO_OUTPUT_BUFFERS()) < 1) {
            fprintf(stderr,"add: no output buffers\n");
            return(3);
      }

]]>
</INIT_CODE>


<MAIN_CODE>
<![CDATA[


      /*
       * read one sample from each input buffer and add
       them
       */
      for(samples = MIN_AVAIL(); samples >0; --samples) {

            sample_out = 0;

            for(i=0; i<ibufs; ++i) {
                  IT_IN(i);
                  sample_out += INF(i,0);
            }
            for(i=0; i<obufs; i++) {
                  if(IT_OUT(i)) {
                        fprintf(stderr,"add: Buffer %d is
full\n",i);
                        return(1);
                  }
                  OUTF(i,0) = sample_out;
            }
      }

      return(0);  /* at least one input buffer empty */


]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[


]]>
</WRAPUP_CODE>

</BLOCK>
```

Fig. 8 Source Code for *add.s*

This example will be very similar to the previous one except that the roles the input and output play are reversed. Note that in the above code there is no section entitled **<INPUT_BUFFERS>**. This is because the number of input buffers is determined at run time in the **<INIT_CODE>.**.

In the **<INIT_CODE>.**.the state variable *no_buffers* is set equal to the number of blocks connected to this block through the use of the *NO_INPUT_BUFFERS()* command. (See section 2.9.) This is done while checking to make sure that there is at least one input connection.

In the **<MAIN_CODE>** a different looping procedure is used to read the samples off the input buffers. The *MIN_AVAIL()* function (see section 2.9) is used. This is because you only want to loop while there are samples on all input buffers, as soon as one of the input buffers runs out of samples the looping must stop.

While inside this loop the current sample from each input buffer must be added together and output. The first step is to advance time on the output buffer: *IT_OUT(0)*. Then the add block must add together all the current input values, this is done in a *for* loop. The loop advances time on the input buffer buffer_no and then adds this into the output. The key command is: *sample_out(0) += INF(buffer_no,0)*. Recall from section 2.9 that the *INF()* command accesses the **value** on the input buffer *buffer_no 0* samples in the past (current sample). This loop traverses all the input buffers.

2.10.4 The Null Block

The null block has the important characteristic that it can replace any block no matter how many input or output buffers the block has. It is a very useful block. In particular, if Capsim runs across a block in a topology that does not exist in the library, it will be replaced with a null block. A warning will be issued.

```
<BLOCK>
<BLOCK_NAME>  null </BLOCK_NAME>

<COMMENTS>
<![CDATA[

      /* null.s */
      /****************************************************
      *

                                null()

      *****************************************************
      *
      This star is useful as a temporary substitute for
      another star.
      The data at any input buffer is passed to the
      matching (same numbered)
      output buffer.  Any unmatched input buffers have
      their data absorbed.
      Any unmatched output buffers output zeroes at a rate
      matching input 0.
      */
]]>
</COMMENTS>

<DESC_SHORT>
This star does nothing, simply puts its input samples on its
output buffer. It is useful as a temporary substitute for a star.
</DESC_SHORT>



<STATES>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>ibufs</NAME>
      </STATE>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>obufs</NAME>
      </STATE>
      <STATE>
            <TYPE>int</TYPE>
```

```
                <NAME> extraOBufs </NAME>
        </STATE>

 </STATES>

 <DECLARATIONS>
                 int i,j;

 </DECLARATIONS>
        <INIT_CODE>
<![CDATA[

        if((ibufs = no_input_buffers()) <= 0) {
                 fprintf(stdout,"null: no input buffers\n");
                          return(1);
        }
                  obufs = no_output_buffers();
        if((extraOBufs = obufs - ibufs) < 0) extraOBufs = 0;

]]>
</INIT_CODE>


<MAIN_CODE>
 <![CDATA[
        for(i=0; i<ibufs; i++) {
                  if(i == 0 && extraOBufs > 0) {
                          while(it_in(0)) {
                                  if(it_out(0)) {
                            KrnOverflow("convolve",0);
                            return(99);
                      }

                                  outf(0,0) = inf(0,0);
                                  for(j=0; j < extraOBufs; j++)
        {

        if(it_out(ibufs+j)){
                            KrnOverflow("null",ibufs);
                            return(99);
                      }

                                          outf(ibufs+j,0) = 0;
                                  }
                          }
                  }
                  else if(i < obufs) {
                          while(it_in(i)) {
                                  it_out(i);
                                  outf(i,0) = inf(i,0);
                          }
                  }
                  else
                          while(it_in(i));
        }

]]>
</MAIN_CODE>
```

```
<WRAPUP_CODE>
<![CDATA[


]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure. 9 Source Code for *null.s*

2.10.5 Probe Blocks

The following block illustrates how to write a probe. This type of block has the feature that all input buffer samples flow through the block to the output buffers. The block performs an operation on the data samples as they flow through. In this case, the samples are printed to a file.

```
<BLOCK>
<BLOCK_NAME> prfile </BLOCK_NAME>

<COMMENTS>
<![CDATA[

/*********************************************************
                  prfile()
**********************************************************
Prints samples from an arbitrary number of input buffers
to a file, which is named as a parameter.  If the file name is
set to "stdout", or "stderr" the output goes to the terminal.
- A sample from each input is printed in columns on a single
line.
      If printing to stdout, these are labeled with signal names.
- The printing function can be disabled without removing the
star,
      via a control parameter.
- Data "flow-through" is implemented: if any outputs are
connected,
      their values come from the correspondingly numbered input.
      (This feature is not affected by the control parameter.)
      (There cannot be more outputs than inputs.)

Programmer:  L.J.Faber

*/

]]>
</COMMENTS>

<DESC_SHORT>
Prints samples from an arbitrary number of input buffers to a
file, which is named as a parameter.
</DESC_SHORT>


<DEFINES>

#define FLOAT_BUFFER 0
#define COMPLEX_BUFFER 1
#define INTEGER_BUFFER 2

</DEFINES>
```

```
<STATES>
      <STATE>
            <TYPE>FILE*</TYPE>
            <NAME>fp</NAME>
      </STATE>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>numberInputBuffers</NAME>
      </STATE>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>numberOutputBuffers</NAME>
      </STATE>
      <STATE>
            <TYPE>int</TYPE>
            <NAME>displayFlag</NAME>
            <VALUE>0</VALUE>
      </STATE>
</STATES>

<DECLARATIONS>

      int i,j,k;
      complex val;

</DECLARATIONS>



<PARAMETERS>
<PARAM>
      <DEF>Name of output file</DEF>
      <TYPE>file</TYPE>
      <NAME>file_name</NAME>
      <VALUE>stdout</VALUE>
</PARAM>
<PARAM>
      <DEF>Print output control (0/Off, 1/On)</DEF>
      <TYPE>int</TYPE>
      <NAME>control</NAME>
      <VALUE>1</VALUE>
</PARAM>
<PARAM>
      <DEF>Buffer type:0= Float,1= Complex, 2=Integer</DEF>
      <TYPE>int</TYPE>
      <NAME>bufferType</NAME>
      <VALUE>0</VALUE>
</PARAM>
</PARAMETERS>

<INIT_CODE>
<![CDATA[

if((numberInputBuffers = NO_INPUT_BUFFERS()) <= 0) {
      fprintf(stdout,"prfile: no input buffers\n");
```

```
        return(1);
}
if((numberOutputBuffers = NO_OUTPUT_BUFFERS()) >
numberInputBuffers) {
        fprintf(stdout,"prfile: more output than input buffers\n");
        return(2);
}
if(strcmp(file_name,"stdout") == 0) {
        fp = stdout;
        displayFlag = 1;
}
else if(strcmp(file_name,"stderr") == 0) {
        fp = stderr;
        displayFlag = 1;
}
else if((fp = fopen(file_name,"w")) == NULL) {
        fprintf(stdout,"prfile: can't open output file '%s'\n",
                file_name);
        return(3);
}
switch(bufferType) {
        case COMPLEX_BUFFER:
                for(i=0; i< numberInputBuffers; i++)
                        SET_CELL_SIZE_IN(i,sizeof(complex));
                for(i=0; i< numberOutputBuffers; i++)
                        SET_CELL_SIZE_OUT(0,sizeof(complex));
                break;
        case FLOAT_BUFFER:
                for(i=0; i< numberInputBuffers; i++)
                        SET_CELL_SIZE_IN(i,sizeof(float));
                for(i=0; i< numberOutputBuffers; i++)
                        SET_CELL_SIZE_OUT(0,sizeof(float));
                break;
        case INTEGER_BUFFER:
                for(i=0; i< numberInputBuffers; i++)
                        SET_CELL_SIZE_IN(i,sizeof(int));
                for(i=0; i< numberOutputBuffers; i++)
                        SET_CELL_SIZE_OUT(0,sizeof(int));
                break;
        default:
                fprintf(stderr,"Bad buffer type specified in prfile
\n");
                return(4);
                break;
}

]]>
</INIT_CODE>


<MAIN_CODE>
<![CDATA[


if(control) {
        if(displayFlag && MIN_AVAIL() > 0) {
                fprintf(fp,"\n");
```

```
            for(j=0; j<(numberInputBuffers-2); j++)
                  fprintf(fp,"%-6s","");
            fprintf(fp,"Output From Prfile '%s'\n",block_P-
>name);
            for(j=0; j<numberInputBuffers; ++j)
                  fprintf(fp,"%-10s  ", SNAME(j));
            fprintf(fp,"\n");
      }
      /* This mode synchronizes all input buffers */
      for(i = MIN_AVAIL(); i>0; i--) {
            for(j=0; j<numberInputBuffers; ++j) {
                  IT_IN(j);
                  if(j < numberOutputBuffers) {
                        if(IT_OUT(j)) {
                              KrnOverflow("prfile",j);
                              return(99);
                        }
                        switch(bufferType) {
                              case COMPLEX_BUFFER:
                                 OUTCX(j,0) = INCX(j,0);
                                 break;
                              case FLOAT_BUFFER:
                                 OUTF(j,0) = INF(j,0);
                                 break;
                                case INTEGER_BUFFER:
                                 OUTI(j,0) = INI(j,0);
                                 break;
                        }

                  }
                  switch(bufferType) {
                        case COMPLEX_BUFFER:
                           val=INCX(j,0);
                           if(fp!= stdout)
                               fprintf(fp,"%-10g %-10g ",
                                         val.re,val.im);
                           else {
                               fprintf(stderr,"%-10g %-10g ",
                                         val.re,val.im);

                           }
                           break;
                        case FLOAT_BUFFER:
                            if(fp!= stdout)
                                  fprintf(fp,"%-10g  ",
                                             INF(j,0));
                           else {
                               fprintf(stderr,"%-10g  ",
                                             INF(j,0));

                           }
                           break;
                          case INTEGER_BUFFER:
                            if(fp!= stdout)
                                  fprintf(fp,"%-d  ",
                                              INI(j,0));
                              else {
                               fprintf(stderr,"%-d  ",
                                              INI(j,0));



                              }
                           break;
                  }

            }
            if(fp!= stdout)
               fprintf(fp,"\n");
            else  {
                  fprintf(stderr," \n ");


            }

      }
}
else {
      /* This mode empties all input buffers */
      for(j=0; j<numberInputBuffers; ++j) {
            if(j < numberOutputBuffers) {
                  while(IT_IN(j)) {
                        if(IT_OUT(j) ){
                              KrnOverflow("prfile",j);
                              return(99);
                        }
                        switch(bufferType) {
                              case COMPLEX_BUFFER:
                                 OUTCX(j,0) = INCX(j,0);
                                 break;
                              case FLOAT_BUFFER:
                                 OUTF(j,0) = INF(j,0);
                                 break;
                                case INTEGER_BUFFER:
                                 OUTI(j,0) = INI(j,0);
                                 break;
                        }

                  }
            }
            else
                  while(IT_IN(j));
      }
}
return(0);


]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[

      if(fp != stdout && fp != stderr)
```

```
            fclose(fp);

]]>
</WRAPUP_CODE>

</BLOCK>
```

Figure 10 Source Code for *prfile.s*

2.11 Template BLOCK

One trick in making the writing of your own blocks easier is to keep a *template* block. The BLOCK "*template.s*" should have all the control structures and comments that direct you as to what needs to be added. Editing this BLOCK, deleting the comments and adding your own customized code, is an easy way to develop a new BLOCK definition. An alternate approach is to edit another block that performs a similar function.

Capsim provides TK/TCL tools to generate block XML code using a graphical interface based on templates for sources,processing blocks, terminators and probes and for a variety of input/output buffer types (floating point, integer, complex, and image).

2.12 Tips and Hints in Writing Blocks

The following are a number of helpful hints in writing blocks;

(1) Do not use reserved names such as buffer, file, int, char, state, etc for variable names. Avoid using names which contain the state variables, parameters or buffer names subsets of the names. Some errors during compilation are related to the fact that *blockgen.xsl* creates *#define* statements for states, buffers, and parameters. The C preprocessor will then substitute these into the pattern creating an error during compilation.

(2) Blockgen.xsl will completely ignore all code not between parameters, states, buffers, initialization code, main_code, wrapup, etc. Hence, if you have global variables to the block, or define constants, put them in an include file. Then use #include "user.h" for example. This works because blockgen.xsl copies the #include line directly to the generated C code.

(3) It is better to keep the block code to a minimum and call subroutines from the block. Let the block code handle parameters, states and buffers.

(4) You don't have to integrate a filter design package etc. into Capsim to use it. All you need to do is write a block that executes the design program using the system() call during initialization. Then you can extract the filter parameters for example, from a file and use the filter during run time (in main_code). In the initialization code segment, you can use the parameters to create an input file to the package, execute it with the system call, extract the results, allocate space for the filter and run it in the main_code. This is efficient since the majority of a simulation time is spent in the main_code. Also, during initialization, memory usage is at a minimum level. Using this tip you can even use FORTRAN design packages with Capsim.

(5) Make sure that there are no spaces after *<INIT_CODE>*, *<MAIN_CODE>*,etc. You may get an unnerving error during blockgen.xsl or compilation. The reason is that blockgen.xsl does a pattern match and a blank becomes part of the pattern.

(6) To define a pointer use the following:

```
int*      samples;
char*     strBuff;
float**   matrix;
```

blockgen.xsl has a problem with *int* **samples,* for example.

(7) blockgen.xsl does not recognize two dimensional arrays such as

```
float     matrix[10][100];
```

However, it does recognize double pointers:

```
float**   matrix;
```

In this case allocate matrix as follows:

```
matrix = (float**)calloc(10,sizeof(float));
for(i=0; i< 10; i++)

matrix[i]=(float*)calloc(100,sizeof(float));
```

This method is good since usually it is done during initialization code, the matrix is allocated to the size needed as determined by say the parameters, and finally, the memory may be freed up during wrapup_code. This is especially useful for images.

(8) Make sure that you consume all samples (cells) on input buffers for efficiency. Also, make sure that no more samples are requested than are available on the input buffer. For *single* input buffers use the following while loop in main code:

```
while(IT_IN(0)){
          ...
}
```

In this case **IT_IN(0)** points to the new sample and is TRUE as long as a sample is available.

When more than one buffer is connected to the input use the following:

```
for(no_samples = MIN_AVAIL();
            no_samples > 0;  --no_samples){
                     ...
}
```

The function **MIN_AVAIL ()** returns the minimum number of samples from among all input connections. While one buffer may have 128 samples another may have 256. Thus the number 128 will be returned. Input beyond the buffer will not be requested and the program will not crash.

(9) A block may increase or decrease the sampling rate. For example for every input sample you may create 10 output samples or vice versa. This is a major advantage of Capsim. So no problem!

(10) When documenting a block provide useful information at the top of the code, since with the view source code facility in Capsim, users will be presented with a window with this information. The information should include comments about parameters and valid ranges in addition to the block's functionality.

3 Adding Your Blocks to CAPSIM

CAPSIM provides the utility to add blocks to the library through the use of the bash shell program *precapsim.sh*. This is a separate program from CAPSIM and is run from the UNIX/LINUX shell not from inside CAPSIM. Another program, *blockmaint.pl*, is also available for library maintenance. When using the MSYS/MINGW environment use *source precapsim.sh* instead of *bash precapsim.sh* to execute the script.

3.1 Precapsim and Makefile

precapsim.sh and the Capsim Makefile provides all the functionality necessary to create your own personalized version of CAPSIM.

Create a new directory any where on your system. For example *WORK*. You will be creating blocks and subroutines to incorporate into Capsim in this directory.

Suppose that CapsimTMK (or Capsim) is installed in the following directory:

/usr/local/CapsimTMK

Then setup the environment variable CAPSIM as follows:

```
export  CAPSIM=/usr/local/CapsimTMK
```

Next make the directory you just created (WORK) the current directory and execute the following:

```
%bash $CAPSIM/TOOLS/precapsim.sh –l
```

The *precapsim.sh* shell command will create all the necessary directories and copy all files including Makefiles and scripts from the $CAPSIM directory. It will also create the *capsim* executable in the current directory.

After executing the script *precapsim.sh* type `make`:

```
%make
```

To execute capsim type:

```
./capsim –b
```

You will be in the Capsim TMK interactive environment. Go ahead and quit.

To start Capsim with TCL scripting support, type

```
./capsim –c
```

To exit, type `exit`.

The shell command *precapsim.sh* creates the directories BLOCKS, SUBS, and *include*. A dummy subroutine and block are also created.

A key point is that the *precapsim.sh* also copies a *Makefile* into the current directory. With this *Makefile* it is very easy to build capsim using your blocks and subroutines. All you need to do is place a block in the BLOCKS directory and, in the main directory (WORK), type *make*.

View the Makefile to see how this happens. The BLOCKS directory has a file called *blocks.mak* . This Makefile has all the necessary dependencies to create the C code from the Block "*.s*" XML code. It adds the block to the block database *blockdatabase.dat* and creates the file *krn_blocklib.c* and the library *libblock.a* with the object files for the blocks. This is all done automatically. The file *blocks.mak* itself is created by a perl script *blockmake.pl* in the $CAPSIM/TOOLS directory. This perl script, given the name of the blocks or **.s*, creates a make file for the blocks: *blocks.mak*.

For more info on the block database and its maintenance see the sections below.

The subroutine directory SUBS also has a *Makefile* and any C subroutine will be compiled and added to the *libsubs.a* library and linked to *capsim*.

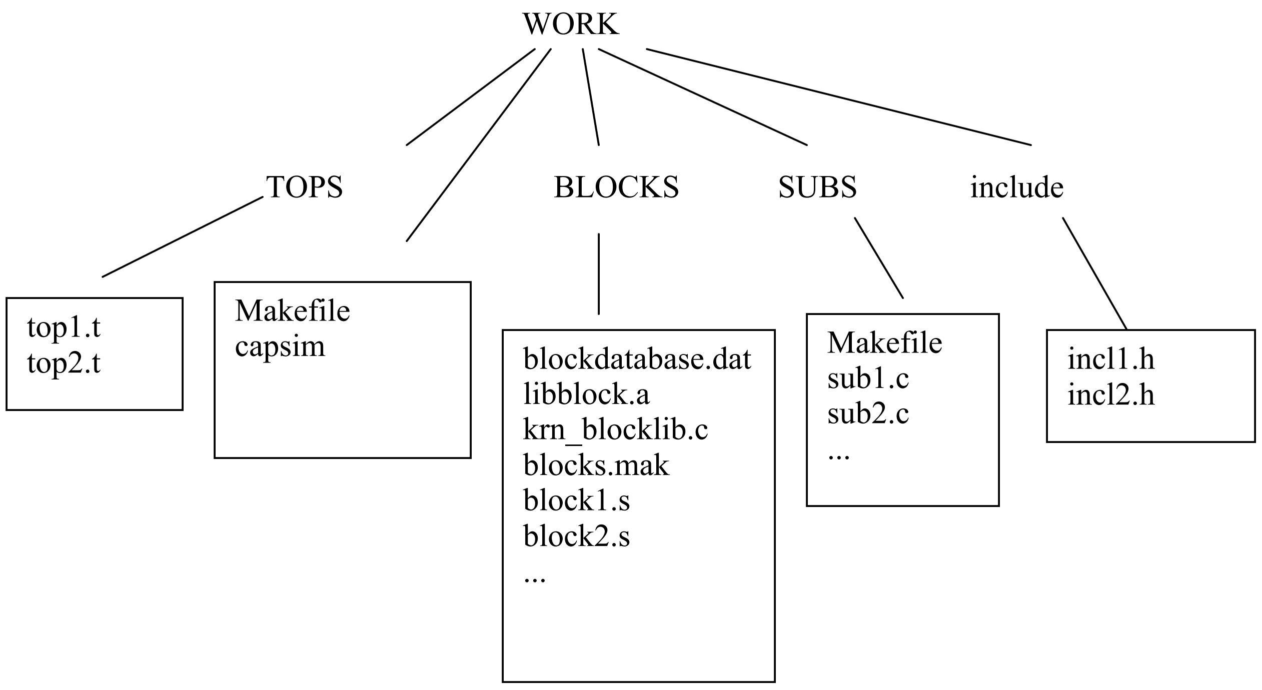
The *include* directory is where common include files for both the blocks in the BLOCKS and C code in the SUBS directory should be placed. The Makefiles use this directory to search for include files when compiling the blocks and subroutines.

So in a nutshell it is real easy to get started. For the first time just execute *precapsim.sh –l* . Afterwords just type *make* to update. Just place blocks in the BLOCKS directory and the subroutine C code in the SUBS directory.

When running *make*, check for errors and correct them. For block compile errors, refer to C code. Make sure you fix the corresponding *.s* code.

You cans also run make in the BLOCKS directory to check and correct errors by typing “make –f blocks.mak”.

You can use *gdb* and *ddd* to debug the block source code.



3.2 BLOCK MAINTENANCE

This is the program that is called by PRECAPSIM and in Makefiles to handle the adding or deleting of a block to/from the library (libblock.a). It may be run by itself to update or examine the library without creating a new version of *capsim*.

BLOCKMAINT is called in one of five ways:

```
%perl $CAPSIM/TOOLS/blockmaint.pl a[dd] blockname

% perl $CAPSIM/TOOLS/blockmaint.pl d[elete] blockname

% perl blockmaint.pl l[ist]

%perl  blockmaint.pl u[sage]

%perl  blockmaint.pl g[enerate]

% perl blockmaint.pl h[elp]
```

The **a** option is used to add a block to *blockdatabase.dat* and krn_blocklib.c is updated.

The **d** option is used to delete a block from *blockdatabase.dat*.

The **l** option gives a list of the blocks in *blockdatabase.dat*.

The **g** option is used to generate the C code krn_blocklib.c from *blockdatabase.dat*. This is useful when blocks are deleted or added to the database and this C code needs to reflect the changes in *blockdatabase.dat*.

```
% perl blockmaint.pl l

* * * Blocks in Main Library * * *

add         delay       node        impulse
gain        time        readfile    prfile
eye         plot        null        zero
lconv       data        linecode    filtnyq
random      sink
```

The **u** option gives the usage for BLOCKMAINT.

```
% perl blockmaint.pl u

usage:
command ->
      a[dd] blockname
      d[elete] blockname
      l[ist]
      u[sage]
      h[elp]
```

Finally, the **h** option gives a brief description as to the function of BLOCKMAINT.

This chapter contains all the information necessary to write your own blocks and maintain a personalized version of *capsim*. The power of CAPSIM comes from the ability to share code with other users. Once a block has been developed by one user it is easily shared with other users.

4 CAPSIM Buffers

4.1 Introduction

In this section we will illustrate how Capsim supports buffers for integer, float,double, complex, double precision fixed point (64 bit) or any other data structure. In Fig. 11, the source block places "samples" on the buffer. By "samples" we mean any data structure. The Consumer Block takes samples from the buffer and processes them. The buffer is output buffer 0 for the Source Block and input buffer 0 for the Consumer Block. Thus both blocks are accessing the same data buffer structure. The mechanism by which "samples" are placed on a buffer and accessed by the receiving block are explained below.

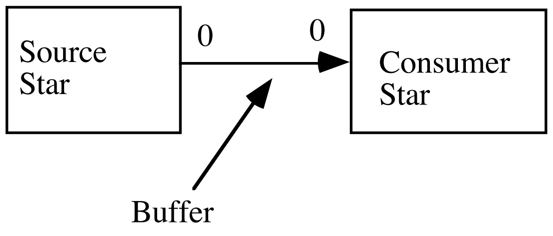


Figure 11.

Although, the explanation may be involved, the insights gained are very valuable for proper block coding and understanding of Capsim's capabilities and limitations. You can actually skip section 4.2 and proceed to section 4.3 if you want to understand how to create your own buffer type.

4.2 Buffer Implementation

The interconnection of blocks within Capsim is accomplished through buffers. These buffers are actually random access buffers. In Capsim, certain enhancements were made to buffer management but essentially the random access buffers are the same implementation as in BLOSIM. For a discussion and history on the development and implementation of random access buffers see Messerschmitt[3], (also visit http://capsimtmk.sourceforge.net/blosim.htm) . Random access buffers are implemented as a circular double-linked list data structure. The buffers are made up of cells. Each cell contains a pointer to user data and two pointers which point to adjacent cells.

The pointer to user data is what gives buffers in Capsim the flexibility to pass floating point, double precision fixed point, complex, character, or any other general data structure between blocks. The user allocates memory for the data structure, places valid data in the structure and the pointer to the structure is what a cell refers to.

The buffer data structure contains the size of the cells, the maximum and minimum number of samples to retain for delays, the current number of allocated cells, pointers to the last cells accessed by the user, for both input and output, the current number of cells stored in the buffer, and other variables and pointers.

The circular double-linked list data structure for a buffer is illustrated in Fig. 11.

[3] David Messerschmitt, *Structured Interconnection of Simulation Programs*, IEEE GLOBECOM 1984

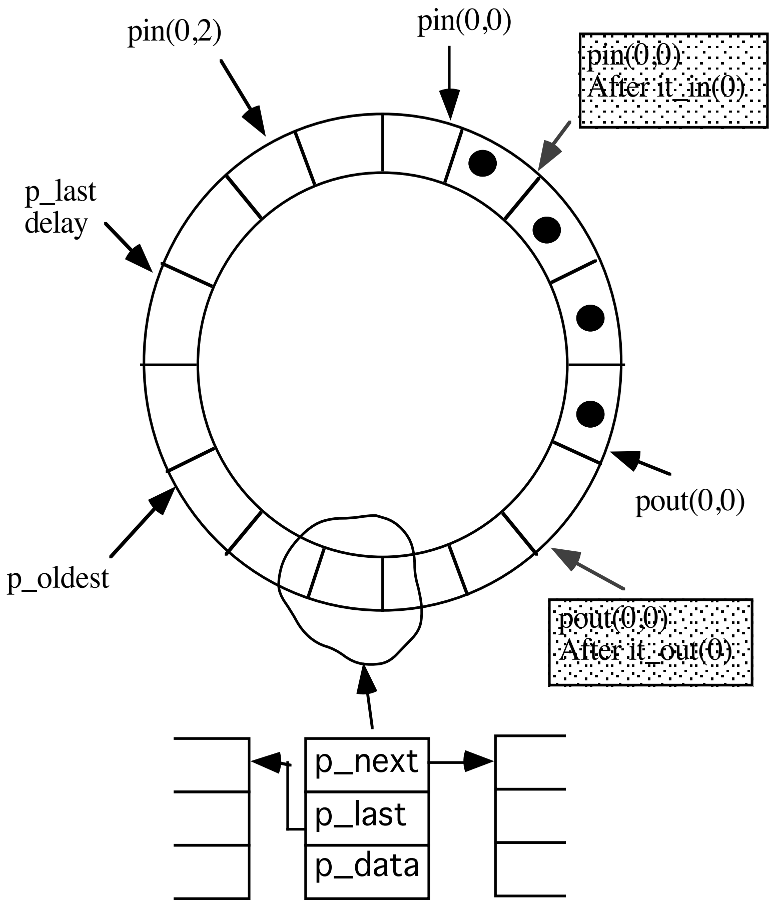


Figure 11 Implementation of a random access buffer

The first time a "sample" is placed on a buffer, 128 cells are allocated for the buffer. This number is a default and may be changed. So, at first there will be 128 available cells in the buffer. The next time a sample is placed in the buffer, a cell will be available for accepting data. All that is required is to point to an empty cell. In the figure, *PIN(0,0)* is a pointer to the current cell in the 0th input buffer. *POUT(0,0)* points to the most current cell available for placement of data in the output buffer. As time is incremented in the blocks, through calls to *IT_IN(0)* and *IT_OUT(0)*, only the pointers move while the cells remain stationary. When time is incremented, by the *IT_IN(0)* function in the receiving block, *PIN(0,0)* will point to the next cell. When *PIN(0,0)* is equal to *POUT(0,0)* then all samples have been accessed from the buffer and the block must return (*IT_IN(0)* will return a 0). In the mean time, when the output block increments time using the function *IT_OUT(0)*, the *POUT(0,0)* pointer is incremented. As long as cells are available and have not been read, or, are not cells that require access due to a set delay requirement on the buffer, there is no problem. However, if the block inputting the data has not processed enough cells, when *POUT(0,0)* equals *p_oldest*, then the buffer is full and another 128 cells are allocated. *p_oldest* is the oldest cell which is necessary to retain in order to satisfy the maximum input and output delay requirements.

In the figure, the cells marked with a dot are forbidden access by the input buffer unless time is incremented.

A number of conclusions can be drawn from the above description on the implementation of buffers in Capsim.

(1) The first time a block calls *IT_OUT(bufferNumber)*, 128 cells are allocated for that buffer. *POUT(bufferNumber,0)* will refer to a cell ready to accept output data.

(2) No further allocation of extra cells will happen as long as time is incremented by the input block and samples accessed from the buffer thus moving *p_oldest* around the circle and ahead of *POUT(bufferNumber,0)*. This is normally the case since a block will process all of its input samples when called for efficiency.

(3) In order to limit the size of buffers and avoid memory reallocations, blocks producing outputs should not exceed placing 128 samples on the output buffer per call. If more samples are outputted then the buffer will need to grow.

(4) As discussed earlier, a block should consume all available input samples before returning to the kernel.

(5) A buffer data structure is not created unless a connection is made to a block. Thus, you should first check to see if a buffer exists before you access it. Otherwise, the program may crash. The correct procedure will be explained below.

4.3 Using and Creating Buffer Types

4.3.1 Simple Buffers

The simplest way to specify input and output buffers when creating a block is to use the code,

```
<INPUT_BUFFERS>
      <BUFFER>
             <TYPE> float </TYPE>
             <NAME> xin </NAME>
     </BUFFER>
</INPUT_BUFFERS>

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

In this case an input buffer called *xin* is specified which uses floating point samples. An output buffer called *yout* is also specified with floating point samples.

When you use this technique, Capsim will check to see if a buffer is connected and will issue an error message prior to running the simulation if it is not. A check is made as to whether the source block buffer type (therefore, its cell size) is compatible with the receiving block and vice versa for output buffers. It is up to you to keep your buffers consistent.

As explained earlier in creating new blocks, the buffer is accessed by first incrementing time (using the *IT_IN()* or *IT_OUT()* routines), and then, for input buffers, assigning a variable to the data in the cell pointed to by the input buffer pointer. As an example, you do the following for input buffers in the main code (assuming one input buffer),

```
<MAIN_CODE>
     while(IT_IN(0)) {
          inputSample = xin(0);
          ...
     }
     ...
```

```
        </MAIN_CODE>
```

The routine *IT_IN(0)* will return a 0 if the buffer is empty. We then return back to the kernel until the block is executed again.

The buffer access *xin(delay)* is actually defined to be *(*((float*)PIN(0,delay)))* for those readers who have read section 4.2.

You can specify a delay associated with the buffer so that previous samples can be obtained from the buffer. The buffer will retain the delayed samples for use. This avoids the necessity to create an array and maintaining it through the use of state variables. Capsim provides this facility for your use. Here is an example:

```
        <INPUT_BUFFERS>
              <BUFFER>
                     <TYPE> float </TYPE>
                     <NAME> xin </NAME>
                     <DELAY>
                          <TYPE>max</TYPE>
                          <VALUE_MAX> 5 </VALUE_MAX>
                     </DELAY>
             </BUFFER>
        </INPUT_BUFFERS>


        ...
        <MAIN_CODE>
              while(IT_IN(0)) {
                    delayedSample=xin(4);
                    currentSample = xin(0);
                    ...
              }
              ...
        </MAIN_CODE>
```

Note that *xin(4)* points to the input sample delayed in time by 4. That is it is equivalent to the z^{-4} operator.

For output buffers, time is simply incremented and samples are placed on the buffer. Here is an example,

```
        <OUTPUT_BUFFERS>
              <BUFFER>
                     <TYPE> float </TYPE>
                     <NAME> yout </NAME>
                     <DELAY>
                          <TYPE>max</TYPE>
                          <VALUE_MAX> 2 </VALUE_MAX>
                     </DELAY>
```

```
     </BUFFER>
</OUTPUT_BUFFERS>
...
<MAIN_CODE>
      for(i=0; i<128; i++) {


            if(IT_OUT(buffer_no)) {
                  KrnOverflow("blockname",buffer_no);
                  return(99);
            }

            value = yout(2) + 0.5*yout(1);
            yout(0)= value;
      }
</MAIN_CODE>
```

If the buffer is full, *IT_OUT(0)* will automatically allocate more cells to the buffer. Note that buffers have a preset maximum length. If too many samples are placed on a buffer, the program will be suspended in an awkward state. In Capsim a block developer can check to see if by placing the sample on the buffer it will exceed its maximum length and can let the block return, so that other blocks have a chance to consume the outputted samples. This problem is avoided by proper use of Capsim and only occurs in huge simulations with multi-rate sampling. It is easily avoided by using pacers described in a separate Capsim application note. In fact, most Capsim source blocks are supplied with pacing capability built in.

4.3.2 Buffer Types and How to Use Them

Capsim is supplied with built in support for the following buffer types. Other types can be added by the user. The procedure will be explained in the next section.

Buffer Type	**Input**	**Output**	**Comments**
float	INF(#buff,delay)	OUTF(#buff,delay)	floating point (32 bit)
double	IND(#buff,delay)	OUTD(#buff,delay)	double precision(64 bit)
char	INC(#buff,delay)	OUTC(#buff,delay)	character (8bit)
int	INI(#buff,delay)	OUTI(#buff,delay)	integer (32 bit)
doublePrecInt	INDI(#buff,delay)	OUTDI(#buff,delay)	typedef struct { long int lowWord; long int highWord; } doublePrecInt;
complex	INCX(#buff,delay)	OUTCX(#buff,dela y)	typedef struct { float re; float im; } complex;

The following examples show how these buffers are used. The first example is the complex conjugate block. This blocks accepts complex input samples, conjugates them and outputs them to as many output buffers that are connected to the block (auto fan-out).

Note that for all buffers except for the default buffer *float*, the cell size must be set to the size of the buffer type. This is accomplished using the function:

```
SET_CELLSIZE_IN(inputBufferNumber,sizeof(complex));
```

and

```
SET_CELLSIZE_OUT(outputBufferNumber,sizeof(complex));
```

```
<BLOCK>
<BLOCK_NAME> cxconj </BLOCK_NAME>

<COMMENTS>
<![CDATA[
/* cxconj.s */
/*****************************************************************
*****
                            cxconj()
******************************************************************
*****
Function has a single complex input buffer, and outputs
the conjugate of each complex input sample to
an arbitrary number of complex output buffers.

DESCRIPTION
Function has a single complex input buffer, and outputs
the conjugate of each complex input sample to
an arbitrary number of complex output buffers.
*/

]]>
</COMMENTS>


<DESC_SHORT>
Function has a single complex input buffer, and outputs the
conjugate of each complex input sample to an arbitrary number of
complex output buffers.
</DESC_SHORT>

<STATES>
       <STATE>
              <TYPE>int</TYPE>
              <NAME>numOutBuffers</NAME>
       </STATE>
</STATES>

<DECLARATIONS>

       int numberOfSamples;
       int i;
       complex val;

</DECLARATIONS>


<INPUT_BUFFERS>
       <BUFFER>
              <TYPE>complex</TYPE>
              <NAME>x</NAME>
       </BUFFER>
</INPUT_BUFFERS>

<INIT_CODE>
<![CDATA[

       /* note and store the number of output buffers */
       if((numOutBuffers = NO_OUTPUT_BUFFERS() ) <= 0) {
              fprintf(stderr,"node: no output buffers\n");
              return(1); /* no output buffers */
       }
       SET_CELL_SIZE_IN(0,sizeof(complex));
       for (i=0; i<numOutBuffers; i++)
              SET_CELL_SIZE_OUT(i,sizeof(complex));

]]>
</INIT_CODE>
```

```
<MAIN_CODE>
<![CDATA[
       for(numberOfSamples=MIN_AVAIL();
                numberOfSamples >0; --numberOfSamples) {
              IT_IN(0);
              for(i=0;i<numOutBuffers;++i) {
                     if(IT_OUT(i)) {
                            KrnOverflow("cxconj",i);
                            return(99);
                     }
                     val=x(0);
                     val.im = - val.im;
                     OUTCX(i,0) = val;
              }
       }

       return(0);  /* input buffer empty */

]]>
</MAIN_CODE>

<WRAPUP_CODE>
</WRAPUP_CODE>

</BLOCK>
```

The next example shows how a single block can support a number of input buffers. The type of buffer is specified as a parameter, and the block adjusts its behavior depending on the buffer type. A good example of this is the spectrum probe. We would like to use the same probe for floating point, double precision, integer, or complex buffers. The following code fragments illustrate the technique:

```
<PARAMETERS>
...
   <PARAM>
      <DEF>Buffer type:0= Float,1= Complex, 2=Integer</DEF>
      <TYPE> int </TYPE>
      <NAME> bufferType </NAME>
      <VALUE> 0 </VALUE>
   </PARAM>

</PARAMETERS>

<INIT_CODE>
<![CDATA[
       ...
switch(bufferType) {
       case COMPLEX_BUFFER:
              SET_CELLSIZE_IN(0,sizeof(complex));
              if(numberOutputBuffers == 1)
                     SET_CELLSIZE_OUT (0,sizeof(complex));
              break;
       case FLOAT_BUFFER:
              SET_CELLSIZE_IN(0,sizeof(float));
              if(numberOutputBuffers == 1)
```

```
                  SET_CELLSIZE_OUT (0,sizeof(float));
            break;
      case INTEGER_BUFFER:
            SET_CELLSIZE_IN(0,sizeof(int));
            if(numberOutputBuffers == 1)
                  SET_CELLSIZE_OUT(0,sizeof(int));
            break;
      default:
            fprintf(stderr,"Bad buffer type specified in
spectrum \n");
            return(4);
            break;
}

      ...
]]>
</INIT_CODE>

<MAIN_CODE>
<![CDATA[
for(samples = MIN_AVAIL(); samples > 0; --samples) {
      /*
       * feed all input buffer samples  to the output
buffers
       */
      for(i=0; i<numberInputBuffers; ++i) {
            IT_IN(i);
            if(numberOutputBuffers > i) {
                  if(IT_OUT(i)) {
                        KrnOverflow("cxconj",i);
                        return(99);
                  }


                  switch(bufferType) {
                        case COMPLEX_BUFFER:
                              OUTCX(i,0) = INCX(i,0);
                              break;
                        case INTEGER_BUFFER:
                              OUTI(i,0) = INI(i,0);
                              break;
                        case FLOAT_BUFFER:
                              OUTF(i,0) = INF(i,0);
                              break;
                  }

            }
        }
        ...
}
]]>
</MAIN_CODE>
```

In the above code, note that we first check to see if an output buffer is connected before we change its cell size. Otherwise we will crash the program since we are accessing invalid memory (your favorite segmentation error message). The constants COMPLEX_BUFFER, INTEGER_BUFFER, and FLOAT_BUFFER are defined in the krn_blocks.h header file which is included in the generated C code by blockgen.xsl .

Blocks can have mixed buffers connected to them. For example you can combine two real channels to form a single complex output channel. Output buffers can also be mixed.

The above examples serve to pave the way for you to write blocks that use the built in Capsim buffer types. In the next section, we will show you how to add your own buffer types to Capsim.

4.3.3 Defining New Buffer Types

You can create new buffer types by writing an include file for blocks that you develop. Later, as the buffers have been tested, you can make them globally accessible to all block developers by adding the contents of the include file to the *krn_blocks.h* include file in the $CAPSIM/include directory. This include file is automatically placed in the C code generated by *blockgen.xsl*.

To illustrate the procedure we will create a complex buffer type. First you need to define a structure for the complex data type:

```
typedef struct {
        float    re;
        float    im;
} complex;
```

Next we define the macro for inputting a complex value from the buffer.

```
#define INCX(BUFFER_NO,DELAY) \
        *(complex *)buffer_access(pblock->pin_buffer[BUFFER_NO],
                                  1,DELAY)
```

Finally we define the macro used to place a complex value on the output buffer.

```
#define OUTCX(BUFFER_NO,DELAY) \
        *(complex *)buffer_access(pblock->pout_buffer[BUFFER_NO],
                                          0,DELAY)
```

That's all folks! What you need to do is make sure that you set the cell size to the size of the new data structure. See section 4.3.1.

Complex buffers may be defined in the input_buffer section or output_buffer section. Below we provide an example for using the complex buffer. Suppose that the include file we created is called "complex.h".

```
<BLOCK>
<COMMENT>
/*************************************************

            cxmakecx()

*************************************************

      Inputs:           one or two real channels

      Outputs:    the complex channel
      Parameters:       None

**************************************************

This block creates a complex buffer from one or two input
buffers.
If one input buffer(buffer 0) is connected, it is assumed
to be the real part.
The imaginary part of the complex output is set to zero.
If two input channels exist then the second channel (buffer
1) is assumed to be
the imaginary part of the complex output sample.

Programmer:       Sasan Ardalan
Date:       September 4, 1991

*/
</COMMENT>

<INCLUDES>
<![CDATA[
#include <math.h>
#include "complex.h"

]]>
</INCLUDES>

<STATES>
      <STATE>
            <TYPE> int </TYPE>
            <NAME> numOutBuffers  </NAME>
      </STATE>
      <STATE>
            <TYPE> int </TYPE>
            <NAME> numberInBuffers </NAME>
      </STATE>
</STATES>



<DECLARATIONS>
      int no_samples;
      float a,b;
      int   i;
      complex calc;
</DECLARATIONS>

<INIT_CODE>
<![CDATA[

       /* store as state the number of input/output buffers
*/
        if((numberInBuffers = NO_INPUT_BUFFERS()) < 1) {
                fprintf(stderr,"cxmakecx: no input
buffers\n");
                return(2);
        }
        if(numberInBuffers >2 ) {
                fprintf(stderr,"cxmakecx: too many inputs
connected\n");
                return(3);
        }
        if((numOutBuffers = no_output_buffers()) < 1) {
                fprintf(stderr,"cxmakecx: no output
buffers\n");
                return(4);
        }
        /*
         * Note that we set the cell size for the output
buffer since it is complex
         */
        for (i=0; i<numOutBuffers; i++)
                SET_CELLSIZE_OUT(i,sizeof(complex));

]]>
</INIT_CODE>

<MAIN_CODE>
<![CDATA[

      /* note the minimum number of samples on the    */
      /* input buffers and iterate that many times    */

      for(no_samples=(MIN_AVAIL());no_samples >0; --
no_samples)

      {
            if(numberInBuffers == 1) {
                  /*
                   * only one input buffer connected
                   * get the sample and set imaginary part
to zero
                   */
                  IT_IN(0);
                  a = INF(0,0);
                  b=0.0;

            } else {
                  /*
                   * two input  buffers  connected
                   */
                  IT_IN(0);
                  a = INF(0,0);
                  IT_IN1);
                  b = INF(1,0);
            }



               for(i=0; i<numOutBuffers; i++) {
                  /*
                   * form complex sample and output on all
connected
```

```
                         * output buffers
                         */
                                     if(IT_OUT(i)) {
                                     KrnOverflow("cxconj",i);
                                     return(99);
                        }

                        calc.re = a;
                        calc.im = b;
                              OUTCX(i,0) = calc;
                      }

          }
          return(0);
]]>
</MAIN_CODE>
</BLOCK>
```

Once you are ready to make your buffer types available to all users, edit the *krn_stars.h* file in the $CAPSIM/include directory and incorporate your definitions.

Using this feature of Capsim, you can create buffers for passing frames of images, or packets. For example, you can create the data structure,

```
typedef struct {
        char    **image_PP;
        int     width;
        int     height;
} image;
```

where image_PP is a double pointer to the 8 bit per pixel image. By passing a pointer, you avoid sending the whole image over the buffer pixel by pixel.

4.4 Buffer Size Management

The main mechanism for placing samples on a buffer is to first call *IT_OUT()*. If *IT_OUT(bufferNumber)* returns a 1, then the maximum number of cells has been exceeded on that buffer. That is, an overflow has happened. You have two choices. One is to return from the block with a non zero error code. This will cause the kernel to stop the simulation and display an error message.(use a call to *KrnOverflow("blockname", bufferNumber)* prior to returning. This will print an understandable error message. The second choice, is to return with a zero. This does not cause the kernel to stop. It will call the other blocks which will hopefully consume samples on the buffer. This in turn will free some space. So if the block stored the sample that was to be outputted, it can safely place it on the buffer prior to the new samples. If a block is written in this way, then all blocks have to use this method. In this way buffers will never exceed the maximum. In fact you can set the maximum number of segments to 4 the minimum value and it will not be exceeded. That is CAPSIM can control and bound the size of buffers. Further more, you can limit blocks to outputting one sample at a time. That is you can also reduce the segment size. Use the following CAPSIM line commands:

```
setmaxseg      MAXIMUM_NUMBER_SEGMENTS
```

by default MAXIMUM_NUMBER_SEGMENTS is 1000. Each segment is 128 cells be default. The size of the segment can be changed by:

```
setcellinc     CELL_INCREMENT
```

Note that you should make sure that blocks output a maximum of CELL_INCREMENT samples each time they are called so that it will be unnecessary for the kernel to increase the buffer size. Use NUMBER_SAMPLES for the value (this is defined to be the cell increment through a global variable). For compatibility you can also use NOSAMPLES. See the source blocks.

5 Appendix A

For reference, the include file "stars.h" in the $CAPSIM/include directory is shown in this appendix. This file is included with all blocks by *blockgen.xsl* .

```
/*
    Capsim (r) Text Mode Kernel (TMK)
    Copyright (C) 1989-2002  XCAD Corporation

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later
version.

    This library is distributed in the hope that it will be
useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General
Public
    License along with this library; if not, write to the Free
Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307  USA

    http://www.xcad.com
    XCAD Corporation
    Raleigh, North Carolina
*/

/* blocks.h */
/******************************************************************
*****

             INCLUDE FILE FOR BLOCKS

******************************************************************
****

This file should be included in all CAPSIM user block routines --
It defines macro substitutions
*/

#define IN 0
#define OUT 1


#define COMPLEX
```

```
typedef struct {
       float  re;
       float  im;
} complex;

typedef struct {
       long int     lowWord;
       long int     highWord;
} doublePrecInt;

typedef struct {
       int          width;
       int          height;
       float**      image_PP;
} image_t, *image_Pt;

POINTER BufferAdd(),BufferAccess();

#define AVAIL(BUFFER_NO) \
       BufferLength(block_P->inBuffer_P[BUFFER_NO])

#define MIN_AVAIL() MinimumSamples(block_P)

#define IT_IN(BUFFER_NO) \
        IncRdPtr(block_P->inBuffer_P[BUFFER_NO])

#define PIN(BUFFER_NO,DELAY) \
        BufferAccess(block_P->inBuffer_P[BUFFER_NO],1,DELAY)

#define INF(BUFFER_NO,DELAY) \
        *(float                              *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define INI(BUFFER_NO,DELAY) \
        *(int                                *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define INC(BUFFER_NO,DELAY) \
        *(char                               *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define IND(BUFFER_NO,DELAY) \
        *(double                             *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define INCX(BUFFER_NO,DELAY) \
        *(complex                            *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define INDI(BUFFER_NO,DELAY) \
        *(doublePrecInt                      *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define INIMAGE(BUFFER_NO,DELAY) \
        *(image_t                            *)BufferAccess(block_P-
>inBuffer_P[BUFFER_NO],1,DELAY)

#define IT_OUT(BUFFER_NO) \
       IncWrPtr(block_P->outBuffer_P[BUFFER_NO])

#define POUT(BUFFER_NO,DELAY) \
        BufferAccess(block_P->outBuffer_P[BUFFER_NO],0,DELAY)
```

```
#define OUTF(BUFFER_NO,DELAY) \
       *(float                             *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define OUTI(BUFFER_NO,DELAY) \
       *(int                               *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define OUTC(BUFFER_NO,DELAY) \
       *(char                              *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define OUTD(BUFFER_NO,DELAY) \
       *(double                            *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define OUTCX(BUFFER_NO,DELAY) \
       *(complex                           *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define OUTDI(BUFFER_NO,DELAY) \
       *(doublePrecInt                     *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define OUTIMAGE(BUFFER_NO,DELAY) \
       *(image_t                           *)BufferAccess(block_P-
>outBuffer_P[BUFFER_NO],0,DELAY)

#define SNAME(BUFFER_NO)  (block_P->signalName[BUFFER_NO])

#define BLOCK_NAME  (block_P->name)

#define SET_DMIN_IN(BUFFER_NO,DELAY) \
       delay_min(block_P->inBuffer_P[BUFFER_NO],DELAY)

#define SET_DMAX_IN(BUFFER_NO,DELAY) \
       delay_max(block_P->inBuffer_P[BUFFER_NO],DELAY)

#define SET_DMAX_OUT(BUFFER_NO,DELAY) \
       delay_max(block_P->outBuffer_P[BUFFER_NO],DELAY)

#define SET_CELL_SIZE_IN(BUFFER_NO,SIZE) \
       CellSize(block_P->inBuffer_P[BUFFER_NO],SIZE)

#define SET_CELL_SIZE_OUT(BUFFER_NO,SIZE) \
       CellSize(block_P->outBuffer_P[BUFFER_NO],SIZE)

#define NO_OUTPUT_BUFFERS()      (block_P->numberOutBuffers)

#define NO_INPUT_BUFFERS() (block_P->numberInBuffers)


/* the following are for examination of the buffers during
debugging */
#define LOOK_OUT(BUFFER_NO) \
      ExamineBuffer(block_P->outBuffer_P[BUFFER_NO])

#define LOOK_IN(BUFFER_NO) \
      ExamineBuffer(block_P->inBuffer_P[BUFFER_NO])

extern double drand48();
```

6 Appendix B

The full source code for convolve.s.

```
<BLOCK>
<LICENSE>
/*  Capsim (r) Text Mode Kernel (TMK) Star Library (Blocks)
    Copyright (C) 1989-2002  XCAD Corporation

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

    http://www.xcad.com
    XCAD Corporation
    Raleigh, North Carolina */
</LICENSE>
<BLOCK_NAME> convolve  </BLOCK_NAME>

<COMMENTS>
<![CDATA[

/* convolve.s */
/***********************************************************************
                             convolve()
************************************************************************
This star convolves the input samples with the impulse response (finite
duration, FIR ) given in a file.
Param: 1 - (file) File with the impulse response samples
       2 - (int) N  number of samples in the impulse response.

This star convolves the input samples with the impulse response (finite
duration, FIR ) given in a file.
Param: 1 - (file) File with the impulse response samples
       2 - (int) N  number of samples in the impulse response.


Date:  September 23, 1988
Programmer: Adali Tulay

*/

]]>
</COMMENTS>

<DESC_SHORT>
This star convolves the input samples with the impulse response (finite
duration, FIR ) given in a file.
</DESC_SHORT>

<INCLUDES>
<![CDATA[
#include <math.h>
#include <stdio.h>

]]>
</INCLUDES>

<DEFINES>

#define PI 3.1415926

</DEFINES>



<STATES>
       <STATE>
              <TYPE> float* </TYPE>
              <NAME> x_P </NAME>
       </STATE>
       <STATE>
              <TYPE> float* </TYPE>
              <NAME> h_P </NAME>
       </STATE>
</STATES>

<DECLARATIONS>

       int i;
       int j;
       float tmp1,tmp2;
        float sum;
       FILE *fopen();
       FILE *imp_F;

</DECLARATIONS>



<PARAMETERS>
<PARAM>
       <DEF>File name containing impulse response samples</DEF>
       <TYPE> file </TYPE>
       <NAME> filename </NAME>
       <VALUE></VALUE>
</PARAM>
<PARAM>
       <DEF>Order of impulse response</DEF>
       <TYPE> int </TYPE>
       <NAME> N </NAME>
       <VALUE></VALUE>
</PARAM>
</PARAMETERS>



<INPUT_BUFFERS>
       <BUFFER>
              <TYPE> float </TYPE>
              <NAME> x </NAME>
       </BUFFER>
</INPUT_BUFFERS>



<OUTPUT_BUFFERS>
```

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

<INIT_CODE>
<![CDATA[

        /*
         * Allocate memory and return pointers for tapped delay line x_P and
         * array containing impulse response samples, h_P.
         *
         */
        if( (x_P = (float*)calloc(N,sizeof(float))) == NULL ||
            (h_P = (float*)calloc(N,sizeof(float))) == NULL ) {
                fprintf(stderr,"convolve: can't allocate work space\n");
                return(4);
        }
        /*
         * open file containing impulse response samples. Check
         * to see if it exists.
         *
         */
         if( (imp_F = fopen(filename,"r")) == NULL) {
                fprintf(stderr,"Convolve could not be opened file was %s \n",
                                filename);
                return(4);
        }
        /*
         * Read in the impulse response samples into the array
         * and initialize the tapped delay line to zero.
         *
         */
        for (i=0; i<N; i++) {
                x_P[i]= 0.0;
                fscanf(imp_F,"%f",&h_P[i]);
        }

]]>
</INIT_CODE>


<MAIN_CODE>
<![CDATA[

        while(IT_IN(0)){
                /*
                 * Shift input sample into tapped delay line
                 */
                tmp2=x(0);
                for(i=0; i<N; i++) {
                        tmp1=x_P[i];
                        x_P[i]=tmp2;
                        tmp2=tmp1;
                }
                /*
                 * Compute inner product
                 */
                   sum = 0.0;
                for (i=0; i<N; i++) {
                     sum += x_P[i]*h_P[i];
                }
                if(IT_OUT(0)) {
                        KrnOverflow("convolve",0);
                return(99);
            }
            /*
             * set output buffer to response result
             */
            y(0) = sum;
        }

]]>
</MAIN_CODE>

<WRAPUP_CODE>
<![CDATA[

        free(x_P); free(h_P);

]]>
</WRAPUP_CODE>

</BLOCK>
```

7 Appendix C

Here is an example initialization and use for the *function* parameter:

```
<PARAMETERS>

            <PARAM>
                  <DEF> Impulse Response Function</DEF>
                  <TYPE> function </TYPE>
                  <NAME> impulse_response </NAME>
                  <VALUE>  "hfilt" </VALUE>
            </PARAM>

</PARAMETERS>

<DECLARATIONS>
/*
 * PFI is a typedef for function returning an integer
 * defined in "krn_capsim.h"
 */
      PFI function,funct_list();
      int hfilt();
</DECLARATIONS>




<INIT_CODE>
      if(strcmp(impulse_response,"hfilt") !=  0)  {
          if((function=funct_list(impulse_response))==NULL)
                 return(1);
      }
      else function = hfilt;

</INIT_CODE>

<MAIN_CODE>
      /*  call  the  function  returning  the  impulse  response
*/
      ... = (*function)(...);
</MAIN_CODE>
```

In this example, the default function hfilt() is declared right inside the BLOCK routine to be a function pointer, but any other function name passed as a parameter is converted to a function pointer by BLOCKGEN.XSL internally.