3.3.1 Circular Buffer

The signals use a circular buffer to store data. Because standard C++ do not have a circular buffer container (at least up to ISO C++17) one was developed. The circular buffer was developed using the same principles and style of the other STL containers, trying to make sure that the future integration of a standard circular buffer in our code will as easy as possible. In this development we use the following references [1, 2, 3]. In [1] a simple circular buffer implementation is presented, in [2] a standard like version of a circular buffer is discussed, and in [3] a comparative assessment is presented considering different implementation strategies. We try to follow [2] as possible.

A circular buffer is a fixed-size container that works in a circular way, the default buffer size is 512. A circular buffer uses a begin and a end pointer to control where data is going to be retrieved (consumed) or added. A full buffer flag is also used to signal the full buffer situation.

Initially, the begin and the end are made to coincide and the full flag is set to false. This is the empty buffer state. When data is added, the end pointer advances. After adding data if the end and the begin pointer coincide the buffer is full.

When data is retrieved, the begin pointer advances. After retrieving data if the begin and end pointer coincide the buffer is empty.

3.4 Log File

3.4.1 Introduction

The Log File allows for a detailed analysis of a simulation. It will output a file containing the timestamp when a block is initialized, the number of samples in the buffer ready to be processed for each input signal, the signal buffer space for each output signal and the amount of time in seconds that took to run each block. Log File is enabled by default, so no change is required. If you want to turn it off, you must call the set method for the logValue and pass $false$ as argument. This must be done before method $run()$ is called, as shown in line 125 of Figure 3.1.

```
115
116     // #####################################################################################################
117     // ########################### System Declaration and Inicialization ###################################
118     // #####################################################################################################
119
120     System MainSystem{ vector<Block*> { &B1, &B2, &B3, &B4, &B5, &B6, &B7, &B8} };
121
122     // #####################################################################################################
123     // #################################### System Run #####################################################
124     // #####################################################################################################
125     MainSystem.setLogValue(false);
126     MainSystem.run();
```

Figure 3.1: Disabling Log File

3.4.2 Parameters

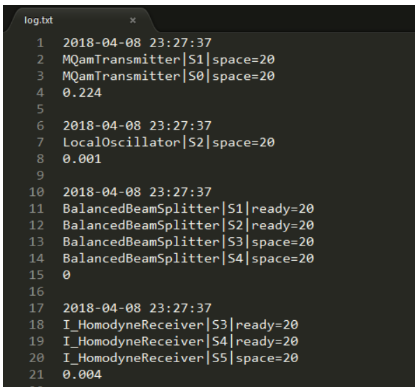
The Log File accepts two parameters: $logFileName$ which correspond to the name of the output file, i.e., the file that will contain all the information listed above and $logValue$ which will enable the Log File if $true$ and will disable it if $false$.

Log File Parameters		
Parameter	**Type**	**Default Value**
logFileName	string	"log.txt"
logValue	bool	true

Available Set Methods		
Parameter	**Type**	**Comments**
setLogFileName(string newName)	void	Sets the name of the output file to the name given as argument
setLogValue(bool value)	void	Sets the value of logValue to the value given as argument

3.4.3 Output File

The output file will contain information about each block. From top to bottom, the output file shows the timestamp (time when the block was started), the number of samples in the buffer ready to be processed for each input signal and the signal buffer space for each output signal. This information is taken before the block has been executed. The amount of time, in seconds, that each block took to run, is also registered. Figure 3.2 shows a portion of an output file. In this example, 4 blocks have been run: MQamTransmitter, LocalOscillator, BalancedBeamSplitter and I_HomodyneReceiver. In the case of the I_HomodyneReceiver block we can see that the block started being ran at 23:27:37 and finished running 0.004 seconds later.


log.txt

```
2018-04-08 23:27:37
MQamTransmitter|S1|space=20
MQamTransmitter|S0|space=20
0.224

2018-04-08 23:27:37
LocalOscillator|S2|space=20
0.001

2018-04-08 23:27:37
BalancedBeamSplitter|S1|ready=20
BalancedBeamSplitter|S2|ready=20
BalancedBeamSplitter|S3|space=20
BalancedBeamSplitter|S4|space=20
0

2018-04-08 23:27:37
I_HomodyneReceiver|S3|ready=20
I_HomodyneReceiver|S4|ready=20
I_HomodyneReceiver|S5|space=20
0.004
```

Figure 3.2: Output File Example

Figure 3.3 shows a portion of code that consists in the declaration and inicialization of the I_HomodyneReceiver block. In line 97, we can see that the block has 2 input signals, $S3$ and $S4$, and is assigned 1 output signal, $S5$. Going back to Figure 3.2 we can observe that $S3$ and $S4$ have 20 samples ready to be processed and the buffer of $S5$ is empty.

```
97      I_HomodyneReceiver B4{ vector<Signal*> {&S3, &S4}, vector<Signal*> {&S5} };
98      B4.useShotNoise(true);
99      B4.setElectricalNoiseSpectralDensity(electricalNoiseAmplitude);
100     B4.setGain(amplification);
101     B4.setResponsivity(responsivity);
102     B4.setSaveInternalSignals(true);
103
```

Figure 3.3: I-Homodyne Receiver Block Declaration

The list of the input parameters loaded from a file is presented at the top of the output file, as shown in Figure 3.4.

```
log.txt
The following input parameters were loaded from a file:
pLength
numberOfBitsGenerated
bitPeriod
shotNoise
-----------------------------------------------------
2018-05-15 00:28:42
MQamTransmitter|S1|space=20
MQamTransmitter|S0|space=20
0.107

2018-05-15 00:28:42
LocalOscillator|S2|space=20
0.004
```

Figure 3.4: Four input parameters where loaded from a file

3.4.4 Testing Log File

In directory *doc/tex/chapter/simulator_structure/test_log_file/bpsk_system/* there is a copy of the BPSK system. You may use it to test the Log File. The main method is located in file *bpsk_system_sdf.cpp*

References

[1] URL: https://embeddedartistry.com/blog/2017/4/6/circular-buffers-in-cc.

[2] URL: https://www.boost.org/doc/libs/1_68_0/doc/html/circular_buffer.html.

[3] URL: https://www.codeproject.com/Articles/1185449/Performance-of-a-Circular-Buffer-vs-Vector-Deque-a.

3.5 Input Parameters System

3.5.1 Introduction

With the Input Parameters System (IPS) it is possible to read the input parameters from a file.

Format of the Input File

We are going to explain the use of the IPS using as an example the PBSK system. In Figure 3.5, it is possible to observe the contents of the file **input_parameters_0.txt** used to load the values of some of the BPSK system's input parameters. The input file must respect the following properties:

1. Input parameter values can be changed by adding a line in the following format: **paramName=newValue**, where **paramName** is the name of the input parameter and **newValue** is the value to be assigned.

2. IPS supports scientific notation. This notation works for the lower case character **e** and the upper case character **E**.

3. If an input parameters is assigned the wrong type of value, method **readSystemInputParameters()** will throw an exception.

4. Not all input parameters need to be changed.

5. The IPS supports comments in the form of the characters **//**. The comments will only be recognized if placed at the beginning of a line.

```
//----------BPSK PARAMETERS--------------
//Changes the value of parameter pLength
pLength=5
//Is able to change the value of different types of parameters
//Of type int
numberOfBitsGenerated=1e3
//Of type double
bitPeriod=20e-12
//Of type bool
shotNoise=false
```

Figure 3.5: Content of file input_parameters_0.txt

Loading Input Parameters From A File

Execute the following command in the Command Line:

some_system.exe <input_file_path> <output_directory>

where **some_system.exe** is the name of the executable generated after compiling the project, **<input_file_path>** is the path to the file containing the new input parameters; **<output_directory>** is the directory where the output signals will be written into.

3.5.2 How To Include The IPS In Your System

In this illustrative example, the code of the BPSK System will be used. To implement the IPS the following requirements must be met:

1. Your system must include **netxpto_20180418.h** or later.

2. A class that will contain the system input parameters must be created. This class must be a derived class of **SystemInputParameters**. In this case the created class is called **BPSKInputParameters**.

3. The created class must have 2 constructors. The implementation of these constructors is the same as **BPSKInputParameters**.

```
BPSKInputParameters();
BPSKInputParameters(int argc, char*argv[]);
```

4. The created class must contain the method **initializeInputParameterMap()**. For every input parameter **addInputParameter(paramName,paramAddress)** must be called, where **paramName** is a string that represents the name of your input parameter and **paramAddress** is the address of your input parameter.

```
void initializeInputParameterMap(){
  //Add parameters
}
```

5. All signals must be instantiated using the constructor that takes as argument, the file name and the folder name, according to the type of signal.

```
Binary S0("S0.sgn", param.getOutputFolderName()) //S0 is a Binary signal
```

6. Method **main** must receive the following arguments.

```
int main(int argc, char*argv[]){...}
```

7. The MainSystem must be instantiated using the following line of code. The ... represent the list of blocks.

```
System MainSystem{ vector<Block*>
     {...},param.getOutputFolderName(),param.getLoadedInputParameters()};
```

The following code represents the input parameters class, **BPSKInputParameters**, and must be changed according to the system you are working on.

```
class BPSKInputParameters : public SystemInputParameters {
public:
  //INPUT PARAMETERS
  int numberOfBitsReceived{ -1 };
  int numberOfBitsGenerated{ 1000 };
  int samplesPerSymbol = 16;
   (...)

  /* Initializes default input parameters */
  BPSKInputParameters() : SystemInputParameters() {
    initializeInputParameterMap();
  }

  /* Initializes input parameters according to the program arguments */
   /* Usage: .\bpsk_system.exe <input_parameters.txt> <output_directory> */
  BPSKInputParameters(int argc, char*argv[]) : SystemInputParameters(argc,argv) {
    initializeInputParameterMap();
    readSystemInputParameters();
  }

  //Each parameter must be added to the parameter map by calling addInputParameter(string,param*)
  void initializeInputParameterMap(){
    addInputParameter("numberOfBitsReceived", &numberOfBitsReceived);
    addInputParameter("numberOfBitsGenerated", &numberOfBitsGenerated);
    addInputParameter("samplesPerSymbol", &samplesPerSymbol);
      (...)
  }
};
```

The method **main** should look similar to the following code.

```
int main(int argc, char*argv[]){

   BPSKInputParameters param(argc, argv);

   //Signal Declaration and Initialization
   Binary S0("S0.sgn", param.getOutputFolderName());
  S0.setBufferLength(param.bufferLength);

  OpticalSignal S1("S1.sgn", param.getOutputFolderName());
  S1.setBufferLength(param.bufferLength);
   (...)

   //System Declaration and Initialization
  System MainSystem{ vector<Block*> { &B1, &B2, &B3, &B4, &B5, &B6, &B7,
       &B8},param.getOutputFolderName(),param.getLoadedInputParameters()};

   //System Run
  MainSystem.run();

  return 0;
}
```

The class **SystemInputParameters**, has the following constructors and methods available:

SystemInputParameters - Constructors	
Constructors	**Comments**
SystemInputParameters()	Creates an object of SystemInputParameters with the default input parameters' values
SystemInputParameters(int argc, char*argv[])	Creates an object of SystemInputParameters and loads the values according to the program arguments passed in the command line

SystemInputParameters - Methods		
Method	**Type**	**Comments**
addInputParameter(string name, int* variable)	void	Adds an input parameter whose value is of type int
addInputParameter(string name, double* variable)	void	Adds an input parameter whose value is of type double
addInputParameter(string name, bool* variable)	void	Adds an input parameter whose value is of type bool
readSystemInputParameters()	void	Reads the input parameters from a file.

3.6 Documentation

As in any large software system documentation it is critical. The documentation is going to be developed in Latex using WinEdt as the recommend editor. The bibliography is per section, for this to work replace the bibtex by biber, go to the WinEdt Options->Execution Modes->Bibtex and replace bibtex.exe by biber.exe.