C++ LIBRARY FOR DIGITAL SIGNAL PROCESSING - slib

Darko Pekar, Radovan Obradoviæ
School of Engineering, University of Novi Sad, Yugoslavia

I INTRODUCTION
Basic idea of this approach is that the whole DSP system should be divided into processing blocks. Each of the blocks performs a relatively simple operation and has appropriate inputs and outputs. Both inputs and outputs are realized as FIFO ring buffers, filled by the previous and read by the next block. Very simple example of this organization is the sample recording system for a microphone, which performs some filtering, and saves the results into a file (Figure 1).

![Figure 1. An example of a simple system](image)

Such organization has several advantages:
- modularity and uniformity in projecting and programming which greatly eases programmers job and standardizes every module look
- simple usage and extremely little code in main program
- good performance due to an efficient usage of cache and modern CPUs characteristics
- possibility of memory savings if the buffer sizes are dimensioned properly
- great possibilities for work division in programming teams due to expressed modularity and uniformity in the code structure.

II LIBRARY OVERVIEW
Considering the described organization, slib is based on the following tree classes:
- SSystem - a framework that represents a discrete system for the signal processing
- SBlock - processing blocks for various operations
- SBuffer - buffers that contain information and the connect blocks.

Buffers overview
Buffers represent parts of the system which function as FIFO organized ring buffers, and connect blocks. Each buffer has the following internal structure:

![Buffer internal structure](image)

The "samples" part contains valid samples that have not yet been processed by the next block. So, the next block should begin reading from the "read_position", can read maximum "n_samples" samples, while previous block should begin writing from "write_position" and can write maximum "free space" samples. Obviously, only three variables are independent, while the remaining two are calculated from the existing. Those three fixed variables are "size", "n_samples" and "write_position", and remaining two are calculated as:

$$\text{free space} = \text{size} - \text{n_samples}$$
$$\text{read_position} = (\text{write_position} - \text{n_samples}) \mod \text{size}$$

Term "samples" doesn’t mean that buffers can contain only scalar (number) values. Buffer can also be an array of vectors, matrices etc. The base class for buffers (SBuffer) does not restrict what kind of data will be stored in the buffer.

A Key feature that makes the buffer usage simple is operator[] defined in each derived class. It should be defined as follows:

```c++
float &operator[](int i) { return data[i % size]; }
```

Desired information index is thus obtained by calculating remainder of index "i" divided by buffer size. Thereby index is always held in the buffer size scope, and programmer can reference any buffer index as if he is operating with a buffer of an infinite size. But it is obvious that this additional operation adds significant overhead to data access. It is true if we want the buffer size to be an arbitrary value and the "%" operator must be used. But if we restrict the buffer size values to exponents of 2, the operation can be reduced to a simple masking of the lowest N bits, and would become:

```c++
float &operator[](int i) {
    return data[i & (size-1)];
}
```
This simple organization of buffer has one drawback. Since only one value for n_samples variable is possible, it means that one buffer can be used only by one block. If we want to proceed it to several blocks we would have to make several copies of the same buffer (with some copy block). In order to avoid such overhead n_samples_map structure is invoked, and every block that uses this buffer has it's own value for n_samples. It requires some additional initializing operations at every block's Reset() function. Also n_samples(...) is now function and this value should be used as a parameter in order to get certain block's n_samples value.

This change preserves modularity and does not compell programmer to insert many different functions into the same block, or to make unnecessary overhead by copying buffers. All this is accomplished with minimal price and unobstruced elegance of the code.

SBuffer class is abstract and represents the base class for a concrete buffer structures realization. All buffer classes must be derived from it. Derived classes should define the following: structure for data storage, operator[] as well as virtual functions GetType() and Reset(). GetType() function should return information about the buffer type. For the time being, it returns three important information: one sample character (scalar, vector, ...), value type (int, float, ...) and buffer size type (binary or common size). These information are mainly used in visualization blocks.

Reset() function sets all important parameters in "ready to go" state. These parameters include buffer basic information (size, n_samples, write position). It also allocates the specified amount of memory.

Blocks overview

For system realization, besides the buffers, implementation of process blocks is also necessary. They usually have input and output buffers, although any of the two may be omitted (in the case of block on the beginning or ending of the system). Base class for these structures is SBlock. Both SSystem and SBlock are inherited from abstract class SSYSTEMBase. This is so because it offers possibility for one (sub)system to be a part of another, i.e. only one block in it. There are three basic functions that should be overloaded in every block:

```cpp
virtual void Reset();
virtual void Process();
virtual void Finish();
```

Reset() function is called by the owner system at the beginning of the processing procedure. It should set initial internal parameters of the block, and output buffers according to the input buffers and internal information. If it succeeds, it should set block's state, and output buffer's state, to appropriate values, as information to the system.

Process() is the function that processes samples from input buffers, and stores the results into output buffers. Therefore it is the main procedure in the block, and actually does the whole work.

Finish() is an optional function called by the system when the processing is finished. Some blocks have to be called at that point to finish some work (e.g. to close an opened file, close the channel, etc.).

In the main program besides the creation of instances of blocks and buffers, and their insertion into the system it is also necessary to connect the blocks. Two blocks are connected via same buffer. This is done by pointers, i.e. output buffer pointer of one block should point to the same buffer as the input buffer pointer of another block. Of course, the buffers of an appropriate type should be used.

System overview

System is the framework that encapsulates all the buffers and blocks, and ensures that they function properly. It has the same virtual functions as the SBlock class but user rarely has a need to overload them. Basically the only function user should use is Add(...) which serves for inserting both blocks and buffers into the system. After inserting all the parts into the system and connecting them, Run() member function should be called for the system to perform all the tasks set.

Visualization blocks

As mentioned above, this library provides an easy way of observing signals in any phase of the processing. All the user has to do is to create an instance of SVis class and add buffers he wants to observe. Due to the buffer internal information provided by GetType() function, this class automatically determines the default view. Both scalar and vector buffers are supported. For scalar buffers the default view is 2D graph, and for vector buffers the map plot in which the brightness of the point represents its value. Some utility options are available, such as zooming and scrolling by any of the axes. An example of a visualization window is given on Figure 3.
The first method advantage is a real time processing with the minimal delay, but it shows some drawbacks with multirate systems, and also in the mentioned block organization idle state would prevail and cause significant time overhead, especially with the blocks that require certain amount of samples to begin processing (e.g. windowing).

The second method neutralizes these drawbacks, but inducts severe delay, that can be inadmissible. This method can also have significant memory requirements. What is not so obvious, and will be explained is that this method demonstrates worse performance on modern processors, than the third method that is a compromise between the two extremes.

Compared to the second method this compromise has smaller requirements for data memory at one point of processing. This enables processor to store all the data, required in one cycle of processing, into the cash and access them far more efficiently.

There are several advantages to the first method. We have already mentioned the idle state reduction. Second advantage is again tied to cash memory, only this time it is program storage cash. In the first method one sample must go through the entire processing algorithm, which excludes the possibility for the program to be stored in the cash if it's rather large. With the third method, certain amount of samples are processed by one part of the program, and when processed, they are passed on to another part, which enables storing currently active part of the program into the cash, resulting in significant speed up. This course of processing is also the cause for the third advantage of the last method. It is known that modern processors have internal pipeline structure, and that some phases of the instructions to come are done in advance in order to speed up the execution part.

Main problem in this sort of processing are jumps, especially conditional. That is why all modern processors apply so-called jump-prediction techniques whose goal is to "learn" where the next jump will go, and from where to fill the pipe. When these algorithms miss, great time delay occurs because of pipe crash and taking another path. These techniques are based on assumption that the program will execute the loop similar amount of times as in the previous pass, which implies that the processor will execute the same code several times. If we organize our program in such manner that all its smaller parts are executed only once, and immediately followed by another part of program (first approach), jump-prediction techniques won't be able to express their advantage, and system performance will be worse.

Finally, the conclusion is that neither of the extremes is good, each for its own reasons. What is the optimal value for the buffer size, that will give system the best performance, depends on the processor and mostly on the cash size.

**IV slab SCRIPT LANGUAGE**

slab script language is designed for those who don't have or do not know how to use C++ compiler, or code compilation each time the programmer changes something in the system. Script language has its own syntax that is similar to that used in C++ programming, and is written as a plain text file. Program slab_script uses this text file as an input parameter, interprets it, creates and runs the system as requested. The system performance is not degraded in any way, since it is first created (as if done directly in C++ code), and then started.

**Script Language Syntax**

slab script language syntax is pretty simple. It consists of functions that require certain number of parameters. Each function call should be enclosed in {} braces. Parameters are divided by blank symbols. The number of parameters is often not fixed, but can vary from minimum to maximum value. There are few built-in functions, and others are practically copied from slab class index. More information can be found on the web, and here we give an example of the script file that describes the system shown in Figure 1:

```plaintext
( Set "input" { SRecorder } )
( Set "buff1" { SBufferFloat } )
( Set "preemphasis" { SFir } )
( Set "buff2" { SBufferFloat } )
( Set "output" { SFileOutput "test.wav" } )
( SRecorder.SetOutput $input $buff1 )
( SFir.SetH $preemphasis { Vector 1 -0.95 } )
( SFir.SetInput $preemphasis $buff1 )
( SFir.SetOutput $preemphasis $buff2 )
( SFileOutput.SetInput $output $buff2 )
( Set "sys" { SSystem } )
( SSystem.AddBlock $sys $input )
( SSystem.AddBuffer $sys $buff1 )
( SSystem.AddBlock $sys $preemphasis )
( SSystem.AddBuffer $sys $buff2 )
( SSystem.AddBlock $sys $output )
( SSystem.Run $sys )
```

**V TEST RESULTS**

In the experiment we observed the time required for a spectrum calculation in the windows with the duration of 32 samples, shifted by 8 samples. Total signal duration was 28.6 seconds, and sampling frequency was 8 kHz. Signal samples, values in the vectors representing the windowed signal fragments, as well as values in the vectors representing the spectrum are of type float (4 bytes). Scheme of the tested system is given in Figure 4.

![Figure 4. System used in experiment](image-url)
Read block in this system represents a so-called dummy block whose only purpose is to read samples form the input buffer in order to enable the system functioning. Main program that describes this system would look like this:

```cpp
SFileInput* file_input = new SFileInput("test.wav");
SHammingWindow* window = new SHammingWindow(32,8);
SFFT* fft = new SFFT;
SReadVectorBlock* read_block = new SReadVectorBlock;
SBufferFloat* buff1 = new SBufferFloat;
SBufferVectorFloat* buff2 = new SBufferVectorFloat;
SBufferVectorFloat* buff3 = new SBufferVectorFloat;
file_input->output = buff1;
window->input = buff1;
window->output = buff2;
fft->input = buff2;
fft->output = buff3;
read_block->input = buff3;
SSystem sys;
sys.Add(file_input);
sys.Add(window);
sys.Add(fft);
sys.Add(read_block);
sys.Add(buff1);
sys.Add(buff2);
sys.Add(buff3);
sys.Run();
```

Graph in Figure 5 shows the test results. On the x-axis is the first buffer size, in samples. On the y-axis is time in seconds. It can be calculated that in the last experiment the whole signal fit into the buffer (second approach). Tests were conducted on a Celeron "A" 450 MHz processor.

Figure 4. Calculation time depending on the buffer size

Optimal performance is reached for the buffer size of 512 samples. For smaller and greater values, the effects we mentioned earlier slowly take place. Pounce for the buffer greatest size could be explained by percentually largest participation of memory allocation in the whole processing, because it is a time consuming task.

VI APPLICATION IN ASR I TTS

Modular organization and pipeline character of signal processing make this library very suitable for application in ASR and TTS algorithms. The Front-end processing requires such organization, but without a tool like this, a programer is forced to implement every step of the algorithm from scratch. With this library, the feature extraction is reduced to a mere selection and connection of the blocks which extract the features for the ASR system. Numerous blocks for feature extraction and other purposes have already been created. If the user wants any new feature to be included in his ASR system, he should build that block first, and connect it to the rest of the system.

VII CONCLUSION

A C++ library for DSP is briefly described in this paper. Besides being simple, modular and easy to use, the library has advantages when performance, i.e. the processing speed, is in question as confirmed experimentally. The tests showed how much we can speed up the system if the modern processors characteristics are taken into account in the right way. Being an open source, and platform independent, this library is very useful tool when when dealing with any sort of DSP. Till now, over 60 processing blocks for various tasks in DSP were implemented, and the library has constantly been improved and expanded. This library is already in use in the ASR systems developed by the AlfaNum team at the University of Novi Sad.

LITERATURE


Abstract: This paper describes a C++ library for digital signal processing (DSP), named slib. It is simple, efficient, lightweight and easy to use, to extend and customize. It can be used for standard DSP operations like FIR, IIR, FFT, but also for somewhat more complicated tasks such as speech recognition, voice recognition, text to speech synthesis etc. It is suitable for off-line and some on-line applications. Since it is highly efficient it can be used in time critical tasks that require speed and no overhead. The library is independent on the operating system or the target platform. It offers an easy way to observe signal in any phase of the processing, by using included visual tools. Only those tools are platform dependent and now available for Windows 9x/NT. A script language for this library is also developed, that enables its usage without knowing the C++ language and without the presence of a compiler. The code is written in the open source manner and can be downloaded and used freely. The source code, as well as more detailed documentation can be found at www.alfanum.com.