Real-Time Block Processing Environment

V 1.0, October 6, 2005, Winter term 2005/06
Christian Feldbauer, feldbauer@tugraz.at,
Signal Processing and Speech Communication Laboratory, www.spsc.tugraz.at
Infeldgasse 16c/II

Abstract

This document describes different buffering concepts implemented in the block processing environment used in most of the DSP programs of this laboratory course.

1 Why Block Processing?

Often, the application requires a processing scheme on a block-by-block basis. Such applications are block transforms or block filters. The main reason, in general, to do block processing is to facilitate optimization for execution speed. This reason should be clear when keeping in mind that more data available at a time reduces data dependencies and, thus, allows efficient pipelining and parallelization of the execution on the eight functional units of a C6000 DSP. Longer blocks reduce the (not optimizable) overhead (e.g., call of interrupt service routines). On the other hand, block processing brings an inevitable drawback: a delay proportional to the block size.

2 Source Code Components of the Real-Time Block Processing Environment

A complete project consists of the following source code files (the files of the block processing environment can be downloaded from www.spsc.tugraz.at/courses/dsplab/; feel free to explore them):

main.c does the whole initialization (board, serial ports, codec, DMA controller) and remains in an endless loop. You can change the sampling rate here. Before the loop, the function init() is called, where you can do additional initializations (such as defining ‘dynamic objects’).

dsk6713*.* contain modules needed for the initialization.

blocks.h provides constants for the block processing environment, such as size and overlap of the working buffer.
isr_tttt.c contains the Interrupt Service Routine (ISR) which is triggered by the DMA controller’s hardware interrupt. It manages the circular buffers needed by the DMA controller to transfer data between the memory and the codec, and it calls the function processing() where the actual signal processing happens. The tttt is a substitution for the type of the actual working buffer (e.g., float or short_cmplx).

vectors.asm is the interrupt vector table. There must be an entry for the ISR described above.

lnk.cmd contains the memory map and arguments for the linker command (e.g., libraries, stack size, heap size, ...)

userfiles The init() and processing() functions must be defined here and all the necessary (global) objects (e.g., additional buffers, filter coefficients, structures of ‘objects’). These two functions are the interface between the block processing environment and your DSP application.

algorithm modules contain definitions and implementations (e.g., structures and functions) for filters, transforms, etc. Such modules can also be provided by libraries.

3 The DMA Circular Buffers

The implemented circular buffer scheme ensures that the DMA controller works continuously in the background (‘shadow buffering’), i.e., that it reads/writes samples from/to the codec sample-by-sample. There is one circular buffer for incoming samples (from the ADC) and one for outgoing samples (to the DAC). Whenever a block (in general, a part of the circular buffer) is completely filled by incoming samples, the DMA controller causes a hardware interrupt. The interrupt service routine (ISR) is called, which prepares the working buffer (see later) and calls the function processing(address of working buffer).

The interval between two interrupts, which corresponds to the size of a block in samples, is set by the constant HOPSZ defined in the header file blocks.h. As it has been already mentioned, each of the two circular buffers consists of several (at least two) blocks. The number of blocks can be (indirectly) set by the constant OVERLAP defined in the header file blocks.h. Note, the number of blocks is defined as NBLOCKS=OVERLAP+2. As an example, for signal processing on an ordinary block-by-block basis (‘double buffering’ or ‘ping-pong buffering’), we have to set OVERLAP=0 to get one block for the DMA controller and another block for the application. The latter block is then called the working buffer. After the next interrupt event, the two buffers are exchanged automatically.

4 The Working Buffer

In general, the working buffer is the part of the DMA circular buffer that is currently not used by the DMA controller. To ensure a universal usage\(^1\), the mentioned part of the circular buffer is ‘unwrapped’ by the ISR to obtain a linear working buffer before the function processing() is called.

\(^1\)In the meantime, also the end of an outgoing block has been reached.

\(^2\)We do not want to require the function processing() to be able to deal with a circular buffer. However, this point constitutes a potential for further optimization.
Real-Time Block Processing Environment

Figure 1: Unwrapping of the DMA circular buffer to obtain a linear working buffer. Note that the most recent data block is at the end of the working buffer (important to keep in mind when implementing overlap-and-add (OLA) block filters).

Figure 2: Overlapping (incoming) consecutive working buffers \( \text{OVERLAP}=3 \).

is called. This unwrapping is visualized in Fig. 1 for a scenario where \( \text{OVERLAP}>0 \). The number of blocks in the working buffer is \( \text{NWBLOCKS}=\text{NBLOCKS}-1=\text{OVERLAP}+1 \).

5 Overlapping Working Buffers

When the constant \( \text{OVERLAP}>0 \), the working buffer contains \( \text{OVERLAP} \) previous data blocks in addition to the most recent data block. In this way, consecutive working buffers do overlap. The actual overlap factor is \( \frac{\text{OVERLAP}}{\text{OVERLAP}+1} \). Fig. 2 shows a scenario of consecutive working buffers with an overlap factor of \( \frac{3}{4} \) \( \text{OVERLAP}=3 \).

The availability of past data blocks simplifies the implementation of delay-line based signal processing functions (such as FIR filters), i.e., the functions do not need to care about storing past data. Other applications that require overlapping input buffers are overlap-and-save (OLS) fast convolution block filters or so-called lapped transforms, e.g., Short-Term Fourier
Figure 3: Overlap-and-add for consecutive outgoing working buffers (OVERLAP=3). The summation is performed by the ISR when the constant OLA=1 is set in the header file blocks.h.

Transform (STFT), Gabor transform, Modified Discrete Cosine Transform (MDCT), or Modulated Lapped Transform (MLT), which are based on a sliding window.  

6 Built-In Overlap-and-Add

The function processing(address of working buffer) is called with a single argument: the pointer to the current working buffer. This implies that the working buffer is used for the new incoming samples as well as for the processed samples for the outgoing signal.

Depending on your application, the ISR can sum the overlapping parts of consecutive working buffers to build the final outgoing signal. This overlap-and-add is enabled by setting OLA=1 in the header file blocks.h and should be preferred for OLA block filters and (inverse) lapped transforms. Note, the summation has no effect as long as OVERLAP=0. In Fig. 3 an overlap-and-add scenario is shown for 3/4 overlap.

However, there are applications that do not require the OLA scheme for the outgoing buffers (e.g., OLS block filters). Setting OLA=0 disables the built-in summation of overlapping blocks. The outgoing signal is then simply built by concatenation of the first HOPSZ samples (i.e., the first block) of the processed working buffers, as illustrated in Fig. 4.

---

3The naming HOPSZ originates from the advance of the sliding window (= hop size).
7 Summary

In this document the used block processing environment has been described. Special attention has been paid to overlapping buffer schemes, which simplify the implementation of block filters or lapped transforms. The discussed buffer schemes are not only important for the DMA-based circular buffers provided by the interrupt-driven environment, which represent the interface to the codec, but also for the signal flow between different signal processing objects. Similar concepts can be found in many other real-time signal processing environments, such as ‘(j)Max/FTS’, ‘MAX/MSP’, or ‘pd’.