Texas Instrument C6748 DMA Tutorial

Julien Osmalskyj

October 24, 2015

1 DMA Ping Pong Buffering

For real-time processing, a ping-pong buffering organization is chosen. Two sets of input and output buffers are used.

- PING buffers (in and out) are being filled with samples by the ADC (IN) and emptied to the DAC (OUT).
- At the same time, the CPU processes samples stored in PONG IN buffer and stores the resulting new samples in the PONG OUT buffer.
- Once the PING buffers have been filled and emptied, interrupt4 is triggered and the buffers are swapped so that the PONG buffers are sent to the codec and the PING buffers are sent to the CPU for processing.

DMA and Ping-Pong buffering are configured in the `isr.c` file. The size of the buffers is `BUFCOUNT x 2 = BUFLENGTH`. The ping and pong buffers contains stereo samples, that is a serie of left and right samples concatenated, as shown in Figure 1.

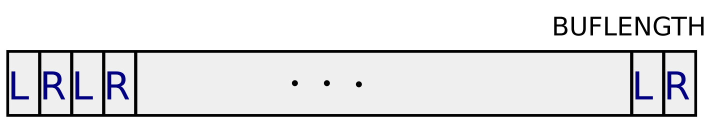

Figure 1: Ping/Pong buffers structure. Left and right samples are concatenated.

When an input buffer is completely filled with new samples from the outside world, and the corresponding output buffer has been emptied to the DAC, an interruption (INT4) is triggered. INT4 reads which buffers should be used for processing samples and which ones should be used by the codecs to be read and output samples.

The interruption sets a global flag `proc_buffer` to either 0 or 1, which corresponds to the PING or PONG buffers. It also sets a flac `buffer_full` to notify that a buffer is ready for processing. Figure 2 shows the code for `interrupt4`.

```c
interrupt void interrupt4(void) { // interrupt service routine

  switch(EDMA_3CC_IPR) {

    case 1:                          // if register TCC = 0
      procBuffer = PING;             // process ping buffers
      EDMA_3CC_ICR = 0x0001;         // clear EDMA3 IPR bit TCC
      break;

    case 2:                          // if register TCC = 1
      procBuffer = PONG;             // process pong buffers
      EDMA_3CC_ICR = 0x0002;         // clear EDMA3 IPR bit TCC
      break;

    default:                         // may have missed an interrupt
      EDMA_3CC_ICR = 0x0003;         // clear EDMA3 IPR bits 0 and 1
      break;
  }

  EVTCLR0 = 0x00000100;
  buffer_full = 1;                   // notify that buffers are ready
  return;
}
```

Figure 2: DMA Interrupt routine in `isr.c`.

When a buffer is ready, flag `buffer_full = 1`. It means that a pair of buffers contains samples to be processed by the CPU. The processing of the samples must be done in the `process_buffer()` function. The **maximum time** available for processing the samples is the time needed to fill and empty input and output buffers by the EDMA controller. Consequently, it depens on the size of the buffers and the sampling rate:

$$FS = 48000\,Hz \;\rightarrow 48000\,\text{samples/sec}$$

$$BUFLENGTH = 2048\,\text{samples}$$

$$\text{Available time} = \frac{2048}{48000} = 0.0427s = 42ms$$

If $FS = 8000\,Hz$, then the available processing time is $256ms$.

The `process_buffer()` function starts by setting two pointers `*inBuf` and `*outBuf` either to PONG IN and PONG OUT or to PING IN and PING OUT by testing the `proc_buffer` flag set by the `interrupt4` interruption. This gives a direct access to the data acquired by the DMA controller, as shown in Figure 3.

Note that `inBuff` and `outBuff` are pointers and therefore correspond to addresses in the input and output buffers. To actually access the data stored

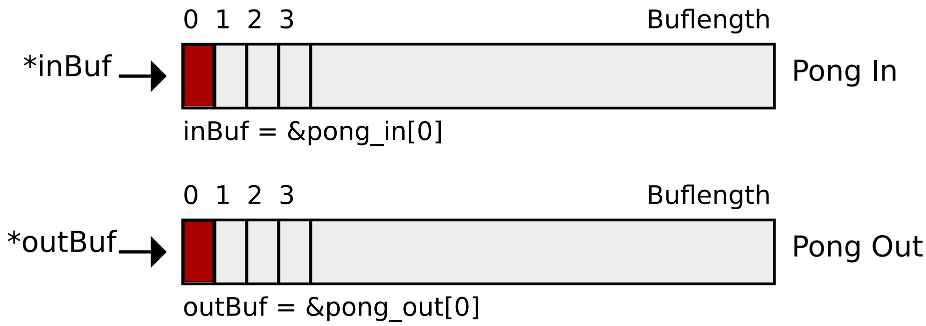

Figure 3: Pointers to Ping and Pong buffers in the `process_buffer` function.

at these addresses, the pointers need to be dereferenced using the $*$ operator. Therefore, `inBuf` corresponds initially to the address of `PongIN[0]` and `*inBuf` corresponds to the content of `PongIN[0]`.

Once the processing is finished, the process function sets the `buffer_full` flag to 0 to indicate that the processing is done. This last process is needed in order to avoid timing errors and therefore undefined behaviour. If the processing is finished *before* the EDMA transfler of the second pair of buffers is finished, this ensures that the `process_buffer` function is not executed twice, or more. Figure 4 shows the code of the `process_buffer` function. Here there is no processing at all, as the input samples are simply copied to the output buffer. Once the processing is done, the `buffer_full` flag is set to 0. The program then waits for the EDMA controller to finish filling samples in the input buffers. Once it is done, the interrupt will be triggered, the `buffer_full` flag will be set to 1 and the `process_buffer` function will be executed again.

The execution of the processing function is controlled by the main program, as shown in Figure 5. The main function starts an infinite loop in which is constantly tests whether the `buffer_full` flag is set to 1 or not.

Files organization

The files are organized as follows. The main program file `main.c` runs the main loop. It only initializes the DSP and runs the main infinite loop. It must include the `prototypes.h`, which declares all the functions available in `isr.c`. That files contains all the functions related to DMA configuration and processing of the data buffers. Other functions can be added to `isr.c`, but they have to be also declared in `prototypes.h`.

2 Frame-based convolution

Using frame-based signal processing, convolutions can be performed by adapting the convolution algorithm so that the signal is processed by blocks of samples.

```c
void process_buffer(void) {
  int16_t *inBuf, *outBuf;    // pointers to process buffers
  int16_t left_sample, right_sample;
  int i;

  if (procBuffer == PING) {   // if buffers to process are PING
    inBuf = pingIN;           // set the pointers to the PING buffers
    outBuf = pingOUT;
  }

  if (procBuffer == PONG) { // if buffers to process are PONG
    inBuf = pongIN;           // set the pointers fo the PONG buffers
    outBuf = pongOUT;
  }

  /* process buffer here */
  for (i = 0; i < (BUFCOUNT) ; i++) { // simple pass through

    left_sample = *inBuf++;  // read pingIN[0] and increment address
    right_sample = *inBuf++; // read pingIN[1] and increment address

    *outBuf++ = left_sample; // copy the input samples to the output buffer
    *outBuf++ = right_sample;
  }

  buffer_full = 0; // indicate that buffer has been processed
  return;
}
```

Figure 4: Process buffer function in `isr.c`.

```c
int main(void) {
  L138_initialise_edma(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB);
  zero_buffers();

  while(1) {
    if (is_buffer_full()) {
      process_buffer();
  }
  }
}
```

Figure 5: Main function in `main.c`.

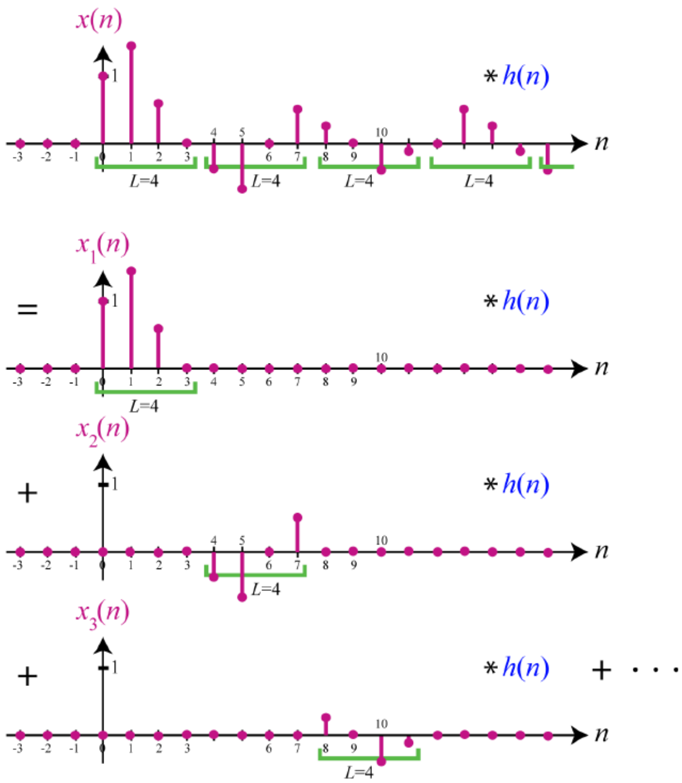

Figure 6: Basic idea of the Overlap-Add method.

This allows to process more than one sample at a time, and gives more processing time as the DMA controller needs some time to fill some buffers while the other ones are being processed.

2.1 Overlap-Add

The Overlap-Add method is based on the observation that when we consider two discrete-time signals $x_k[n]$ and $h[n]$, with lengths L and M respectively, the resulting convolution $y_k[n] = x_k[n] * h[n]$ has a length of $L + M - 1$. Using this idea, we can divide the input stream $x[n]$ into L-length blocks and convolve each block with $h[n]$, and then sum all the convolution outputs along the L-boundaries, as shown in Figure 6.

Using the EDMA controller, we have an easy way of splitting the input stream into fixed-length input blocks. Therefore, we can process each block separately using the overlap-add algorithm.

In Figure 6, the operation of convolving a very long signal $x[n]$ with $h[n]$ is equivalent to the operation of convolving each L-block denoted $x_k[n]$ with $h[n]$

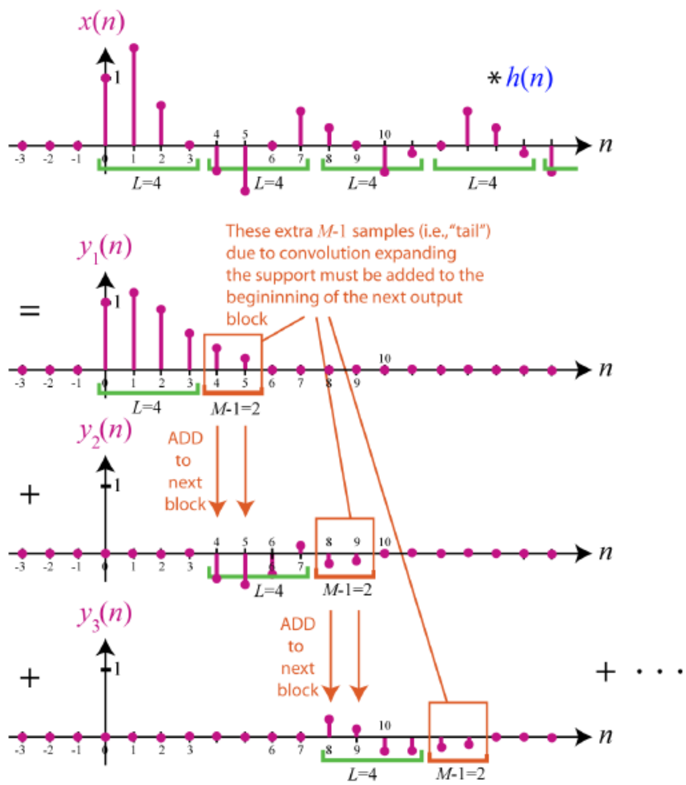

Figure 7: Tail resulting from convolution must be added to the next convolved block.

and then conducting addition judiciously to deal with the "tail" region from each block convolution, as it will be explained next. An important aspect is that after convolving each block with $h[n]$, the resulting intermediate signal is $L + M - 1$ samples in length, and therefore, we have $M - 1$ extra samples at the end due to the convolution. These $M - 1$ extra samples must be added to the first $M - 1$ samples of the *next* convolved block, as shown in Figure 7.

One main challenge in a real-time processing scenario is that the timing of completing a block convolution needs to be approximately synchronized with the overall output speed so that tail region may be added to the next block at the right time. If the process of convolving each block is slower than outputting the samples of blocks already convolved, then the tail region will not have the opportunity to be added to the next block, resulting in an erroneous output. One way to deal with this is to slow down the rate of the output. However, another much more attractive way is to make use of the Fast Fourier Transform for convolutions.

2.2 Overlap-Add Convolution with FFT

Computing convolution using the standard equation results in a lot of computational effort. The process can be improved through the use of the FFT in order to speed up the algorithm. Indeed, performing a convolution in the time domain results in a simple *multiplication* in the frequency domain. The procedure to perform a convolution with the FFT is described below.

1. Zero-pad the filter $h[n]$ with $K - M$ zeros, where K is the first power of two greater than $L + M - 1$, where L is the length of one block of data, and M is the length of the filter h.
2. Compute the K-point FFT of the zero-padded filter $h[n]$ and save it.
3. Zero-pad each input segment $x_k[n]$ of length L with $K - L$ zeros to make it the same size as the filter.
4. Compute the K-point FFT of the segment.
5. Multiply sample-by-sample the two FFT results from Steps 2 and 4.
6. Take the inverse FFT of the resulting product to produce $y_k[n]$.

Next, we need to use the same overlap-add algorithm than for the classical convolution. Indeed, the output blocks $y_k[n]$ need to only be of length L. Therefore, we need to save the $K-L$ remaining samples in order to add them to the output of the next processed block. The process is illustrated by Figure 8.

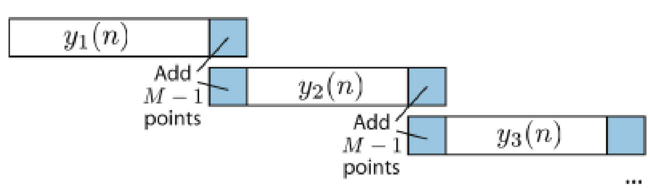

Figure 8: FFT overlap-add method.