Application Report
SPRA645A - April 2001

Circular Buffering on TMS320C6000

Dipa Rao *DSP West Applications*

ABSTRACT

This application report explains how circular buffering is implemented on the TMS320C6000™ devices. Circular buffering helps to implement finite impulse response (FIR) filters efficiently. Filters require delay lines or buffers of past (and current) samples. Circular addressing simplifies the manipulation of pointers in accessing the data samples.

This application report addresses the following:

- Circular buffer concept
- Circular buffer setup and manipulation
- Circular addressing example for block FIR

Contents

List of Figures

List of Tables

1 Circular Buffer Concept

Finite impulse response (FIR) filters are commonly found in a lot of digital signal processing applications. Finite impulse response filters are implemented based on the following equation

$$y_n = \sum_{i=0}^{N-1} x_{n-i} * a_i \qquad (1)$$

where a_i is a filter coefficient , x_k is a data sample, k is the time index and N is the number of taps. As seen from equation (1), in order to calculate an output y_n, we need to maintain a buffer of previous values (also called a delay line) along with the current sample. Typically, a pointer is set up at the beginning of the sample array (oldest sample) and then manipulated to access the consecutive values. Whenever a new sample needs to be added to the delay line either all the values need to be shifted down (Figure 1) or the oldest value need to be overwritten (Figure 2). The second technique can be implemented by using circular mode for pointer access.

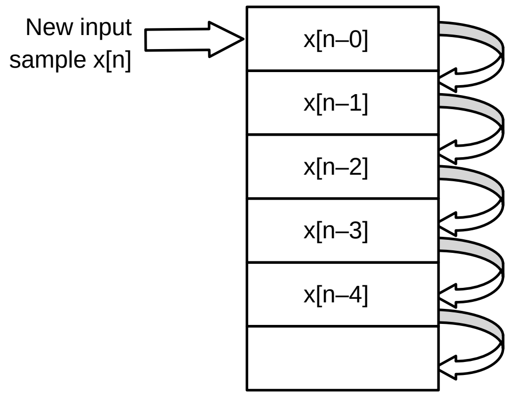

Figure 1. Delay Line Implemented With Shifting of Samples

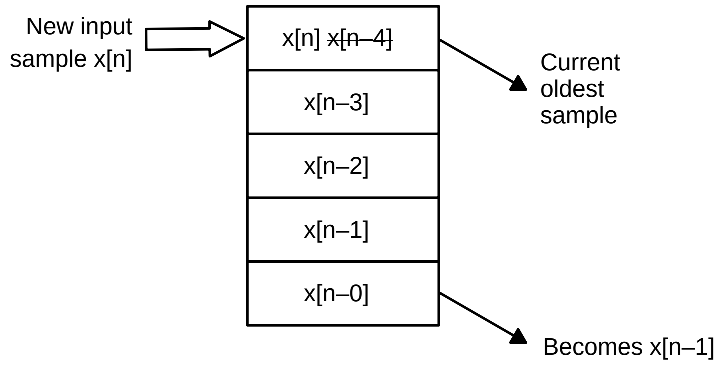

Figure 2. Delay Line With Pointer Manipulation Using Circular Addressing

Circular addressing uses pointer manipulation to add the new samples to the buffer by overwriting the oldest available samples hence reusing the memory buffer. When the pointer reaches the last location of the delay line it needs to wrap back to the beginning of the line. This would normally involve some amount of software overhead. When circular addressing is used, the pointer automatically wraps back to the top whenever the bottom of the buffer is reached. As a consequence, the memory locations appear to be tied together in a circular manner hence the name ‘circular buffer’. Most digital signal processors implement circular buffering in hardware in order to conserve memory and minimize software overhead.

2 Circular Buffer Implementation

On the C6000™ processor data values in memory are accessed by setting up a register as a pointer to memory and then loading values using indirect or pointer addressing. The pointers to memory locations can be manipulated in linear or circular mode. Of the 32 registers available (A0–A15 and B0–B15), circular addressing can be implemented using any of eight registers (A4–A7,B4–B7) as a pointer. A 32 bit address mode register (AMR), shown in Figure 3, is used to indicate whether registers A4–A7 and B4–B7 are enabled in linear or circular mode.

31 26	25 21	20 16	15 14	13 12	11 10	9 8
Reserved	BKO	BK1	B7 mode	B6 mode	B5 mode	B4 mode

7 6	5 4	3 2	1 0
A7 mode	A6 mode	A5 mode	A4 mode

Figure 3. Address Mode Register (AMR)

The block size fields (BK0 and BK1) in AMR can be used in conjunction with any of the registers to specify the length of the buffer as shown in Table 1. A value N in the register BK0/BK1 corresponds to a block size of $2^{(N+1)}$ bytes. With the 5 bit field in BK0 and BK1 it is possible to specify 32 different block lengths for circular addressing from 2 bytes to 4G bytes (which incidentally is also the maximum address reach using a 32 bit pointer register). This makes it convenient to implement filters with very large number of taps. The circular buffer is always aligned on the block size byte boundary. This is necessary for the hardware to correctly locate the start and end of the circular buffer. For instance, if the register A4 contains an address 0x80000005 and the block size is 64 bytes then the start address of the defined circular buffer is strapped to 0x80000000 and the end address is 0x8000003F.

Table 1. AMR Mode Field Encoding

Mode	Addressing Option
00	Linear Mode
01	Circular Mode using BK0 size
10	Circular Mode using BK1 size
11	Reserved

To understand the circular addressing, let us consider a case where we require a pointer A4 to be set up in circular mode to point to a buffer of length 16 bytes. Firstly, the pointer A4 needs to be setup in circular mode by initializing bit field [1:0] of register AMR with '01' (if using BK0) or '10' (if using BK1). The field BK0 set to 00011 resulting in a circular buffer size of 2^{3+1} or 16 bytes. So finally, AMR register will have bit field [1:0] set to 01 and field [25:21] set to 00011. Let us assume that pointer A4 contains the address 0x8000000E. Now if the pointer A4 is post incremented by using an instruction such as,

```
LDH            ; *A4++,   A8
```

This instruction loads the contents of 0x8000000E into A8 and then increments the pointer A4 by 2 bytes, then the pointer will end up at location 0x80000000 due to circular addressing. Circular addressing hardware automatically defines address 0x80000000 as the top of the buffer and 0x8000000F as the end of the 16 byte long buffer as shown in Figure 4. In fact, if the pointer A4 were to point to any shaded location in the buffer shown in Figure 4, the start and end addresses of the circular buffer would be the same for the given definition of circular buffer.

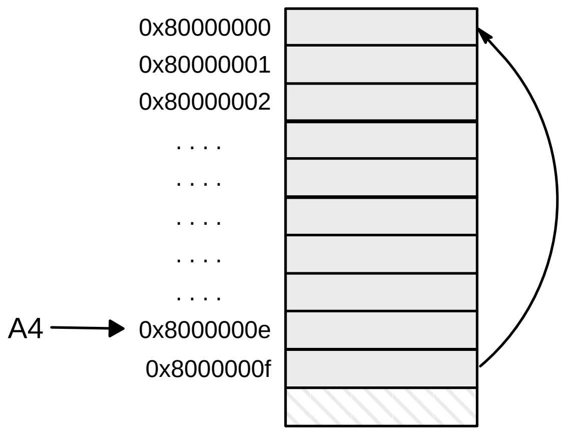

Figure 4. Circular Buffer Pointer Modification

3 Block FIR Circular Buffer Example

The following example shows how circular addressing is used when performing block FIR filtering. Block FIR filter implementation involves the computation of multiple outputs in the same invocation of the filtering routine. During a sampling period multiple inputs are added to the circular buffer and multiple outputs are generated. This implies that the calculation shown in equation (1) is repeated for the number of outputs desired. This requires tracking the pointer to the data samples and manipulating them to access the correct data in different iterations through the loop. In this application note, circular addressing is used to simplify the access of data samples from the buffer. The buffer is defined to be greater than the block size (number of filter outputs) and to be a power-of-2 (2^N). To understand this we consider a simple case with a block of 6 input samples (16bits each) and a 4 tap (16bits) FIR filter. The data buffer needs to be large enough to contain the samples and needs to be a power-of-2. Let us assume the buffer to be 16 half words (2^5 or 32bytes long). Figure 5 illustrates the state of the data samples in the buffer on 4 consecutive calls to the filtering algorithm. The first time the function is called after 6 new samples are added to the buffer. To calculate the first set of outputs y[0]–y[5],

$$\begin{aligned} y_0 &= a_0x_0 + a_1x_{-1} + a_2x_{-2} + a_3x_{-3} \\ y_1 &= a_0x_1 + a_1x_0 + a_2x_{-1} + a_3x_{-2} \\ &\ldots. \\ y_5 &= a_0x_5 + a_1x_4 + a_2x_3 + a_3x_2 \end{aligned} \tag{2}$$

We require 9 data samples x[–3]– x[5] from the buffer as seen in equation (2). Thus, to calculate each output we require the current sample as well (N–1) previous samples where N is the number of taps in the filter. For the first output y_0, the current sample is x[0] while the old samples are x[–3],x[–2] and x[–1]. The calling routine uses index to indicate the first location that needs to be accessed for the current iteration through the filtering loop. Every time the function is called the calling algorithm passes not only the pointers to the data samples but also the index into the buffer as well. Hence, the first time the algorithm is called the index into the data sample buffer is 0 or x[–3]. After the outputs are computed the routine returns to the main calling program. Before the filtering routine is invoked next, a new set (block size=6) of samples is added to the buffer. Hence, the second block of data x[6]–x[11] is added to the delay line and then the function is called a second time. To calculate the outputs y[6]–y[11] we require the data samples x[3]–x[11] which would imply that the index into the delay line is 6 as illustrated in Figure 5. The index can be easily calculated as

$$\text{Index} = [\text{Previous\_Index} + \text{Block\_Size}]\ \%\ \text{Buffer\_Size} \tag{3}$$

In our example during the second call to the algorithm, the previous index was 0, block size is 6 and buffer size is 16 hence the index is 6 (pointing to location x[3]).

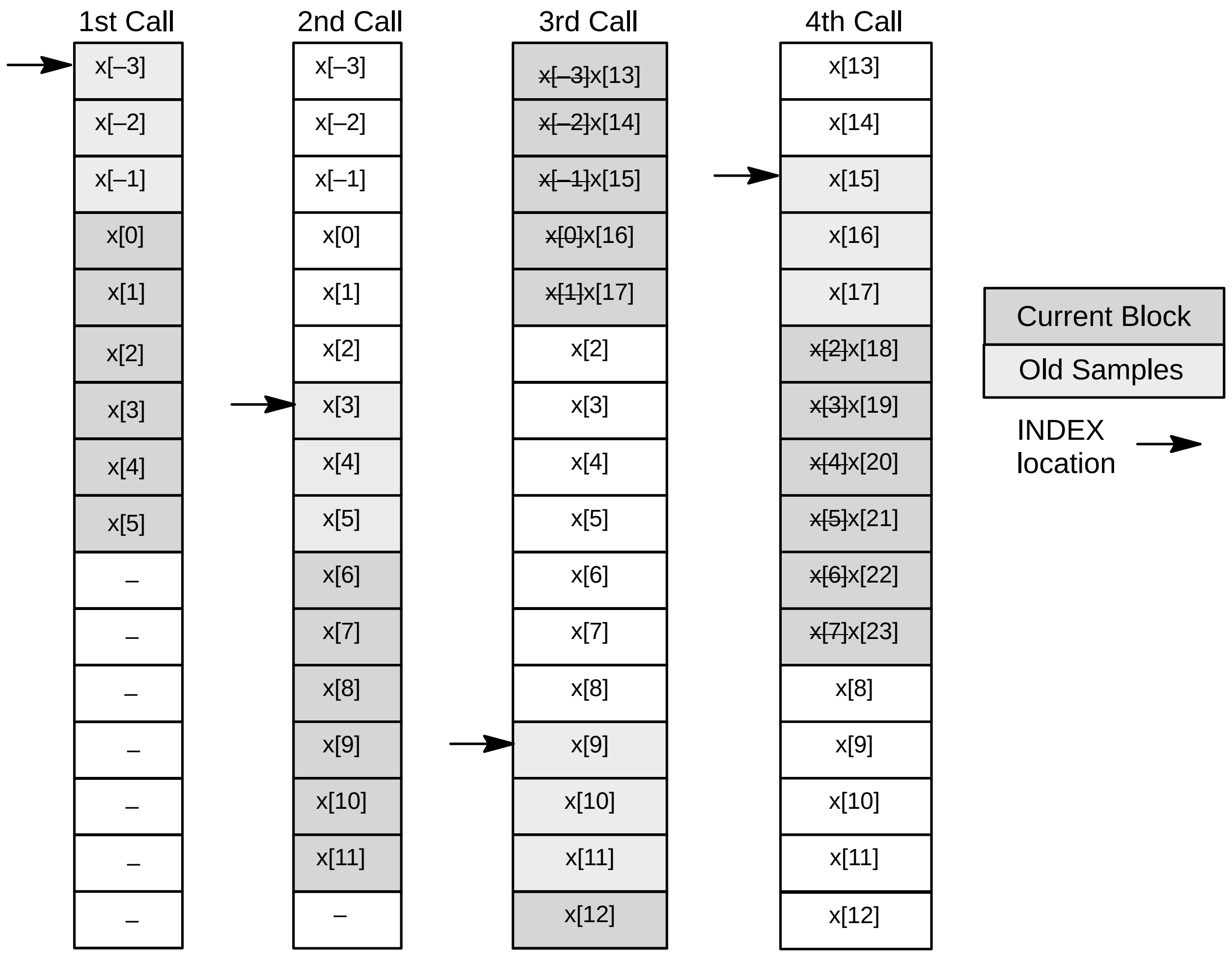

Figure 5. Delay Line State

Now let us consider how the pointer needs to be manipulated within the filtering algorithm. The third block of 6 samples x[12]–x[17] needs to go into the delay line right after sample x[11]. However, since the buffer is only 16 half words long, some of the new samples will be added at the top of the buffer overwriting the old values like x[–3], x[–2], etc. Within the filtering algorithm the pointer used to access the data needs to be manipulated so that new samples are accessed correctly. In our example, during the third call to the filtering algorithm, as shown in Figure 5, the pointer needs to be wrapped around to the top of the buffer to access value x[13]. Setting up the pointer in circular mode ensures that the pointer will fold back to the beginning of the buffer automatically. By using the circular buffering technique, the filtering algorithm can be called multiple times without using up data memory space.

The hand assembly code to implement this technique is presented in Appendix A. In the code, the number of taps is set to be 16, the block size for filtering is 20 and the buffer size is set to 128 bytes or 64 half words (16bits). The instructions that setup the circular pointer mode are highlighted. The code presented uses the technique of software pipelining to achieve maximum efficiency and performance. As a consequence there are certain restrictions on the size of the block as well as number of taps. In order to make the algorithm efficient, the FIR filter algorithm has been unrolled such that four multiply-accumulates (as shown in equation (2)) happen in the same iteration, hence the number of taps must be a multiple of 4. In addition, two output values are computed in the same loop kernel as a result the block size must be a multiple of 2. This is explained in the header of the function. Two pointers A7 and B4 access the data samples in the fir routine so both the pointers are set up in circular mode with the appropriate block size indicated in BK0 bit field. The sample code also includes a main function that calls the filtering algorithm multiple times, each time with a newly calculated index value as well as newly loaded data samples.

4 Conclusion

This application note discussed the basic benefits of using circular addressing and illustrated it with an example of block FIR filtering. The C6000 code generation tools will offer support for circular buffering from the C environment in a future release.

Appendix A Block FIR With Circular Buffering

A.1 C Code: Main.c

```
// Define Circular Block Size (BUF_LEN), Number of Coefficients (NUM_TAPS) and //
Block FIR Size (NUM_SAMP). BUF_LEN is defined in bytes
#define BUF_LEN   128
#define TOT_SAMP  100
#define NUM_TAPS   16
#define NUM_SAMP   20
#define Q15     32768
// Define output array and input array to filtering routine
short out_array[TOT_SAMP];
short in_array[BUF_LEN/2];

// Align Input Array (Circular Buffer) at appropriate byte boundary
#pragma DATA_ALIGN(in_array,BUF_LEN)

// Initialized data sample values for testing
short inp_samp[TOT_SAMP+NUM_TAPS–1]=
{0xee4d,0xc000,0x08bb,0x3709,0xdc9a,0xae4d,0xf7b2,0x2600,0xcca9,0x9e5b,
0xe93f,0x178c,0xc010,0x91c2,0xded1,0x0d1e,0xb80f,0x89c2,0xd971,0x07bf,
0xb573,0x8726,0xd9a9,0x07f7,0xb87e,0x8a30,0xdf73,0x0dc0,0xc0e1,0x9294,
0xea3a,0x1888,0xcdc8,0x9f7a,0xf8ed,0x273b,0xdde9,0xaf9c,0x0a16,0x3864,
0xefac,0xc15e,0x1c00,0x4a4e,0x014a,0xd2fd,0x2ce2,0x5b30,0x1104,0xe2b6,
0x3b0f,0x695c,0x1d49,0xeefb,0x451e,0x736b,0x24e1,0xf694,0x4a0f,0x785c,
0x270b,0xf8bd,0x4963,0x77b1,0x238f,0xf541,0x432d,0x717a,0x1ac6,0xec78,
0x380a,0x6657,0x0d90,0xdf42,0x2915,0x5763,0xfd3d,0xceef,0x17cc,0x461a,
0xeb6c,0xbd1f,0x05e6,0x3434,0xd9e4,0xab96,0xf52c,0x2379,0xca62,0x9c15,
0xe747,0x1594,0xbe73,0x9025,0xdd99,0x0be7,0xb745,0x88f8,0xd91a,0x0767,
0xb591,0x8743,0xda3b,0x0888,0xb980,0x8b32,0xe0df,0x0f2c,0xc2ae,0x9461,
0xec5d,0x1aaa,0xd032,0xa1e5,0xfb90    };

// Define low pass filter coefficients
short coeff_array[NUM_TAPS] = {  0*Q15/10000,  37*Q15/10000,
                               113*Q15/10000, 163*Q15/10000,
                                 0*Q15/10000, 535*Q15/10000,
                              1371*Q15/10000,2163*Q15/10000,
                              2163*Q15/10000,1371*Q15/10000,
                               535*Q15/10000,   0*Q15/10000,
                              –163*Q15/10000,–113*Q15/10000,
                               –37*Q15/10000,   0*Q15/10000};
```

```
// Prototype filter routine
extern void fir_circ_asm(short *y,short *x,int n,short *h, int s, int m, int size,
int indexex);
// Define function that fills input array with data
void readdata(short *y,short *x, int n, int m);
main()
{
  // call assembly FIR routine
  int scale_factor=15;
  int index=0;
//Fill input array with samples
readdata(inp_samp,in_array,0,35);
//Call Block FIR algorithm
fir_circ_asm(out_array,in_array,NUM_TAPS,coeff_array,scale_factor,NUM_SAMP,BUF_LEN,
index);
// Compute next INDEX value based on the old INDEX and Block FIR (not circular
buffer block) size BUFLEN/2 is used since the pointer points to 16 bit data
index= (index+NUM_SAMP)%(BUF_LEN/2);


readdata(inp_samp,in_array,35,55);
fir_circ_asm(&out_array[NUM_SAMP],in_array,NUM_TAPS,coeff_array,scale_factor,NUM_SA
MP,BUF_LEN,index);
index= (index+NUM_SAMP)%(BUF_LEN/2);
readdata(inp_samp,in_array,55,75);
fir_circ_asm(&out_array[2*NUM_SAMP],in_array,NUM_TAPS,coeff_array,scale_factor,NUM_
SAMP,BUF_LEN,index);
index= (index+NUM_SAMP)%(BUF_LEN/2);
readdata(inp_samp,in_array,75,95);
fir_circ_asm(&out_array[3*NUM_SAMP],in_array,NUM_TAPS,coeff_array,scale_factor,NUM_
SAMP,BUF_LEN,index);


}
void readdata(short init_values[],short array[],int n,int m)
{
  int i,temp;
  for(i=n;i<m;i++)
  {
    temp=i%(BUF_LEN/2);
   array[temp]=init_values[i];
   }
}
```

A.2 Hand Coded Assembly File: FIRCIRC.ASM

```
*======================================================================*
*                                                                      *
*     TEXAS INSTRUMENTS, INC.                                          *
*                                                                      *
*     NAME                                                             *
*           fir_circ_asm                                               *
*                                                                      *
*     USAGE   This routine is C Callable and can be called as:         *
*          void fir_circ_asm(short r[], short x[], int nh, short h[],
*               int s, int nr, int size, int index)
*
*          r       = output array
*          x       = input array, must fill the Block Size
*          nh      = number of coefficients (MULTIPLE of 4 >= 4)
*          h       = coefficient array
*          s       = output scaling factor
*          nr      = number of outputs to calculate (MULTIPLE of 2
*                    >= 2)
*          size    = Block Size (in Bytes) for Circular Addressing
*                   (Power of 2)
*          index   = Initial Index into input array
*
*          (See the C compiler reference guide.)
*
*     DESCRIPTION
*           The routine fir_circ_asm() performs a Finite Impulse
*           Response filter using circular addressing on the input
*           array alone w/ inital index and output scaling.  It
*           operates on 16–bit data with a 40–bit accumulate.  The
*           final accumulated output is scaled down by the scaling
*           factor s. This scaling factor determines the balance
*           between precision of the output and room for overflow
*           of the output. Q15 x Q15 = Q30, so s set to 15 will
*           produce Q15 output. However, if the outputs were to
*           overflow, a larger s could be used to allow more headroom.
*           For example, s set to 17 would produce Q13 output, but
*           allow for two bits of overflow at the cost of precision.
*           The FIR assumes that the number of filter coefficients is a
*           multiple of 4 and the number of output samples is a
*           multiple of 2. This routine has no memory hits regardless
*           of where x,h and r are located in memory. The filter has nx
*           input samples are nh coefficients. The assembly routine
*           generates 2 output samples at a time. The block size of the
*           circular buffer is passed in as the argument “size” in
*           bytes.
*
*           On subsequent calls to this routine, index should be
*           updated to indicate where in the input array the last
*           call finished. The new index is (index + nr) % (size/2)
*           An example of how this indexing and circular buffering
*           works is shown below:
*
*           First call, nh=4, nr=40, size=128 (64 shorts), index=0
*
*           |<–         Input array, x[64], 128 bytes              –>|
*           |<–  nr+nh–1 (43) inputs used –>|<– unused inputs (21) –>|
*           |1111111111111111111111111111111|xxxxxxxxxxxxxxxxxxxxxxxx|
*           | 1st input needed to restore  state for next call (40)  |
*
*           Second call, nh=4, nr=40, size=128, index=(0+40)%(128/2)=40
*
*           |<–             Input array, x[64], 128 bytes          –>|
*           |<–last 19 inputs–>|<–unused (21)–>|<– first 24 inputs –>|
*           |222222222222222222|xxxxxxxxxxxxxxx|111111111111111111111|
*
*           | first input needed to restore state for next call (16) |
*
*
*           Third call, nh=4, nr=40, size=128, index=(40+40)%(128/2)=16
*
*           |<–            Input array, x[64], 128 bytes           –>|
*           |<–unused (16)–>|<–    inputs used (43)  –>|<–unused(5)–>|
*           |xxxxxxxxxxxxxxx|11111111111111111111111111|xxxxxxxxxxxxx|
*
*           | first input needed to restore state for next call (56) |
*
*
*           void fir_circ_asm(short r[], short x[], int nh, short h[],
*                                int s, int nr, int size, int index)
*               {
*               int             i, j;
*               Long40          y0;
*               Long40          round = (Long40) 1 << (s – 1);
*               for (j = 0; j < nr; j++) {
*                    y0 = round;
*                    for (i = 0; i < nh; i++)
*                    y0 += x[(i + j + index) % (size/sizeof(short))]
*                          * h[i];
*                    r[j] = y0 >> s;
*                    }
*               }
*
*
*     ASSUMPTIONS:
*           x fills Block Size, aligned on a Block Size boudary
*           e.g. size=128, x[64] aligned on 128 byte boundary
*           nh MULTIPLE of 4 >= 4
*           nx EVEN >= 2
*           index <= (size/2) – 2
*           size is a power of 2
*
*     MEMORY NOTE:
*           This code has no memory hits regardless of where x and h
*           are located in memory.
*
*
*     TECHNIQUES
*           The inner loop is unrolled four times thus the number of
*           filter coefficients must be a multiple of four.  The outer
*           loop is unrolled twice so the number of output samples must
*           be a multiple of 2.
*
*           If an odd number of output samples is needed or possible,
*           the final store can either be removed or conditionally
*           executed depending on whether nx is even or odd.  This code
*           would have to be added to the existing code.
*
*           The outer loop, like the inner loop, is software pipelined
*           as well.  e, o, and p in the comments of the individual
*            instructions correspond to the epilogue, outer loop, and
*            prologue respectively.
*
*            Refer to FIR example in the optimizing assembly chapter of
*            the programmer’s guide for more information.
*
*     CYCLES                                                             *
*            nr*(nh + 11)/2 + 17                                          *
*=======================================================================*
*      Copyright (C) 1997–1999 Texas Instruments Incorporated.           *
*                       All Rights Reserved                              *
*=======================================================================*
*=============== SYMBOLIC REGISTER ASSIGNMENTS ========================
               .asg     B15,         B_SP
*=======================================================================*
*      Copyright (C) 1997–1999 Texas Instruments Incorporated.           *
*                             All Rights Reserved                        *
*=======================================================================*
              .sect    ”.data:copyright_h”
_Copyright:   .string  ”Copyright (C) 1999 Texas Instruments Incorporated. ”
              .string  ”All Rights Reserved.”
              .sect    ”.text:hand”
*             .include ”fir_circ_h.h62”
              .def          _fir_circ_asm
_fir_circ_asm:
;Disable interrupts in code since code is software pipelined
        MVC     .S2     CSR,B0          ; Load CSR to change GIE bit
        AND     .L2     –2,B0,B0        ; Disable interrupts
||      STW     .D2     B0,*B15––       ; Save old CSR value on stack
        MVC     .S2     B0,CSR

        MVC     .S2     AMR, B0
||      LMBD    .L1     1, A10, A0      ; set circular block size

        STW     .D2     B0, *B15––      ; push AMR on the stack
||      SUB     .L1     15, A0, A0      ; set circualr block size

        STW     .D2     B10, *B15––     ; push B10 on the stack
||      MV      .L1X    B15, A1         ; copy stack pointer
||      ADD     .S1     15, A0, A0      ; set circualr block size
```

```
        STW     .D2     A10, *B15--[2]  ; push A10 on the stack
||      STW     .D1     B11, *--A1[2]   ; push B11 on the stack

        STW     .D2     A11, *B15--[2]  ; push A11 on the stack
||      STW     .D1     B12, *--A1[2]   ; push B12 on the stack

*** BEGIN Benchmark Timing ***
B_START
        B       .S1     OUTLOOP
||      ADD     .D1     6, A6, A10      ; nh + 6 half array reset
||      SHL     .S2X    A0, 16, B0      ; set circular block size
||      ADDAH   .D2     B4, B10, B4     ; x += index
||      MV      .L1X    B3, A0          ; copy return address
||      MV      .L2     B8, B1          ; move m

        SHR     .S1     A6, 2, A3       ; nh / 4
||      MV      .L2X    A10, B10        ; copy array reset
||      SET     .S2     B0, 8, 8, B0    ; set B4 (x) in circular mode
||      ADD     .L1X    2, B6, A5       ; copy h
||      STW     .D2     A12, *B15--[2]  ; push A12 on the stack
||      STW     .D1     B13, *--A1[2]   ; push B13 on the stack

        ADD     .L1X    2, B4, A7       ; copy x
||      ADD     .L2     B10, 2, B14     ; array reset
||      SET     .S2     B0, 6, 6, B0    ; set A7 (x) in circular mode
||      STW     .D2     A13, *B15--[2]  ; push A13 on the stack
||      STW     .D1     B14, *--A1[2]   ; push B15 on the stack

        ADDAH   .D1     A5, A10, A5     ; compensate for first pass
||      ADDAH   .D2     B6, B10, B5     ; compensate for first pass
||      MVC     .S2     B0, AMR         ; setup AMR

        ADDAH   .D1     A7, A10, A7     ; compensate for first pass
||      ADDAH   .D2     B4, B14, B4     ; compensate for first pass
||      MVK     .S2     1, B2           ; setup j loop priming

        ADD     .L2X    2, A4, B11      ; copy y
||      STW     .D2     A15, *B15--     ; push A15 on the stack
```

```
LOOP:   ; LOOP BEGINS HERE
  [!A1] ADD     .L2X    A9, B13:B12,B13:B12 ; y1 += p00,            i=0
||[!A1] ADD     .L1X    B9, A13:A12,A13:A12 ; y0 += p01,            i=0
||      MPY     .M2     B3, B7, B6      ; p11 = x1 * h1,        i=1
||      MPY     .M1     A6, A11, A11    ; p00 = x0 * h0,        i=1
||      LDH     .D1     *++A5[2], B9    ;** h1 = *h++,          i=0
||      LDH     .D2     *++B5[2], A9    ;** h0 = *h++,          i=0

  [!A1] ADD     .L2     B6, B13:B12,B13:B12 ; y1 += p11,            i=0
||[!A1] ADD     .L1     A6, A13:A12,A13:A12 ; y0 += p10,            i=1
||      MPY     .M1X    B3, A9, A15       ;* p10 = x1 * h0,       i=0
||      MPY     .M2X    A15, B9, B9       ;* p01 = x0 * h1,       i=0
||      LDH     .D2     *++B4[2], B3      ;** x1 = *x++,          i=1
||      LDH     .D1     *++A7[2], A6      ;** x0 = *x++,          i=1
||[A2]  SUB     .S1     A2, 1, A2         ; i++

  [A2]  B       .S1     LOOP              ;* for i
||[!A1] ADD     .L2X    A11, B13:B12,B13:B12 ; y1 += p00, i=1
||[!A1] ADD     .L1X    B7,  A13:A12,A13:A12 ; y0 += p01,            i=1
||      MPY     .M2     B8, B9, B6        ;* p11 = x1 * h1,       i=0
||      MPY     .M1     A15, A9, A9       ;* p00 = x0 * h0,       i=0
||      LDH     .D1     *++A5[2], B7      ;** h1 = *h++,          i=1
||      LDH     .D2     *++B5[2], A11     ;** h0 = *h++           i=1
||[B0]  SUB     .S2     B0, 1, B0         ; decrement flushing count

  [!A1] ADD     .L2     B6, B13:B12,B13:B12 ; y1 += p11,            i=1
||[B0]  ADD     .L1     A15, A13:A12,A13:A12 ;* y0 += p10,           i=0
||      MPY     .M2X    A6, B7, B7        ;* p01 = x0 * h1,       i=1
||      MPY     .M1X    B8, A11, A6       ;* p10 = x1 * h0,       i=1
||      LDH     .D2     *++B4[2], B8      ;*** x1 = *x++,         i=0
||      LDH     .D1     *++A7[2], A15     ;*** x0 = *x++,         i=0
||[A1]  SUB     .S1     A1, 1, A1         ; decrement priming

        ; inner loop branch occurs here
OUTLOOP:
        LDH     .D2     *––B4[B14], B3    ;p x1 = *x++,           i=1
  [B1]  B       .S2     LOOP              ;p for i
||      LDH     .D2     *++B4[2], B8      ;p x1 = *x++,           i=0
||      LDH     .D1     *––A7[A10], A15   ;p x0 = *x++,           i=0
||      MV      .L2X    A8, B6            ;o copy s
||      SUB     .L1     A8, 1, A1          ;o s – 1
||      SHR     .S1     A13:A12,A8,A13:A12 ;e y0 >>= s

        SHR     .S2     B13:B12,B6,B13:B12 ;e y1 >>= s
||      LDH     .D1     *––A5[A10], B9     ;p h1 = *h++,           i=0
||      LDH     .D2     *––B5[B10], A9     ;p h0 = *h++,           i=0

  [!B2] STH     .D1     A12, *A4++[2]      ;e r[0] = y0
||[!B2] STH     .D2     B12, *B11++[2]     ;e r[1] = y1
||      MVK     .S1     1, A12             ;o \ round = (Long40) 1
||      ZERO    .L1     A13                ;o /
||      ZERO    .L2     B2                 ;o clear j loop priming

        LDH     .D2     *++B4[2], B3       ;p x1 = *x++,           i=1
||      LDH     .D1     *++A7[2], A6       ;p x0 = *x++,           i=1
||      SHL     .S1     A13:A12,A1, A13:A12 ;o y0 = round = (Long40) 1<<(s–1)
||      ADD     .L2X    1,A3, B0           ;p setup flushing count

  [B1]  B       .S2     LOOP               ;p for i
||      LDH     .D1     *++A5[2], B7       ;p h1 = *h++,           i=1
||      LDH     .D2     *++B5[2], A11      ;p h0 = *h++            i=1
||      MV      .L2X    A13, B13           ;o y1 = round

        LDH     .D2     *++B4[2], B8       ;p* x1 = *x++,          i=0
||      LDH     .D1     *++A7[2], A15      ;p* x0 = *x++,          i=0
||      MV      .L2X    A12, B12           ;o y1 = round
||[B1]  SUB     .S2     B1, 2, B1          ;p j++
||      MV      .L1     A3, A2             ;p i < n
||      MVK     .S1     1, A1              ;p i loop priming

B_END:
*** END Benchmark Timing ***
END:    LDW     .D2     *++B15, A15        ; pop A15 off the stack
||      MV      .L1X    B15, A1            ; copy stack pointer

        LDW     .D1     *++A1[3], A13      ; pop A13 off the stack
||      LDW     .D2     *++B15, B14        ; pop B14 off the stack

        LDW     .D1     *++A1[2], A12      ; pop A12 off the stack
||      LDW     .D2     *++B15[2], B13     ; pop B13 off the stack
```

```
        LDW     .D1     *++A1[2], A11    ; pop A11 off the stack
||      LDW     .D2     *++B15[2], B12   ; pop B12 off the stack
||      MV      .L2X    A0, B3           ; move return address

        LDW     .D1     *++A1[2], A10    ; pop A10 off the stack
||      LDW     .D2     *++B15[2], B11   ; pop B11 off the stack

        LDW     .D2     *++B15[2], B10   ; pop B10 off the stack
        LDW     .D2     *++B15, B8       ; pop AMR off the stack

        LDW     .D2*    *++B15, B0
||      B       .S2     B3
        NOP     3
        MVC     .S2     B8, AMR          ; Restore AMR
        MVC     .S2     B0, CSR          ; Restore CSR

*======================================================================
* end of fir_circ assembly code
*======================================================================
*      Copyright (C) 1997–1999 Texas Instruments Incorporated.          *
*                            All Rights Reserved                        *
*======================================================================
```