Appendix H
The ADC Module with DMA

This appendix discusses the ADC operation on PIC24 CPUs that have DMA. The ADC material in Chapter 11 is focused on PIC24 CPUs without DMA. This appendix assumes that the reader has covered the material in Chapter 11 on ADC operation and in Chapter 13 on the DMA module. This appendix frequently refers to figures in Chapters 11 and 13.

ADC DIFFERENCES WITH AND WITHOUT DMA

As discussed in Chapter 13, some PIC24 CPUs support a DMA module. For example, the PIC24HJ32GP202 CPU used in Chapter 11 that discusses ADC operation does not support DMA, while the PIC24HJ64GP502 used in Chapter 13 does have the DMA module. The ADC module in DMA-capable PIC24 CPUs operates differently in some modes than what is discussed in Chapter 11. In referring to Figure 11.7, the ADC module in a PIC24 CPU with DMA does not have ADC buffers 1 through F (registers `ADCxBUF1` through `ADCxBUFF`). What this means is that ADC operation that only involves a single conversion where the result is placed in ADCxBUF0 is the same regardless of whether DMA is supported or not. Thus, the example code in Figure 11.14 and the `convertADC1()` function of Figure 11.13 operates the same way regardless of whether DMA is supported or not.

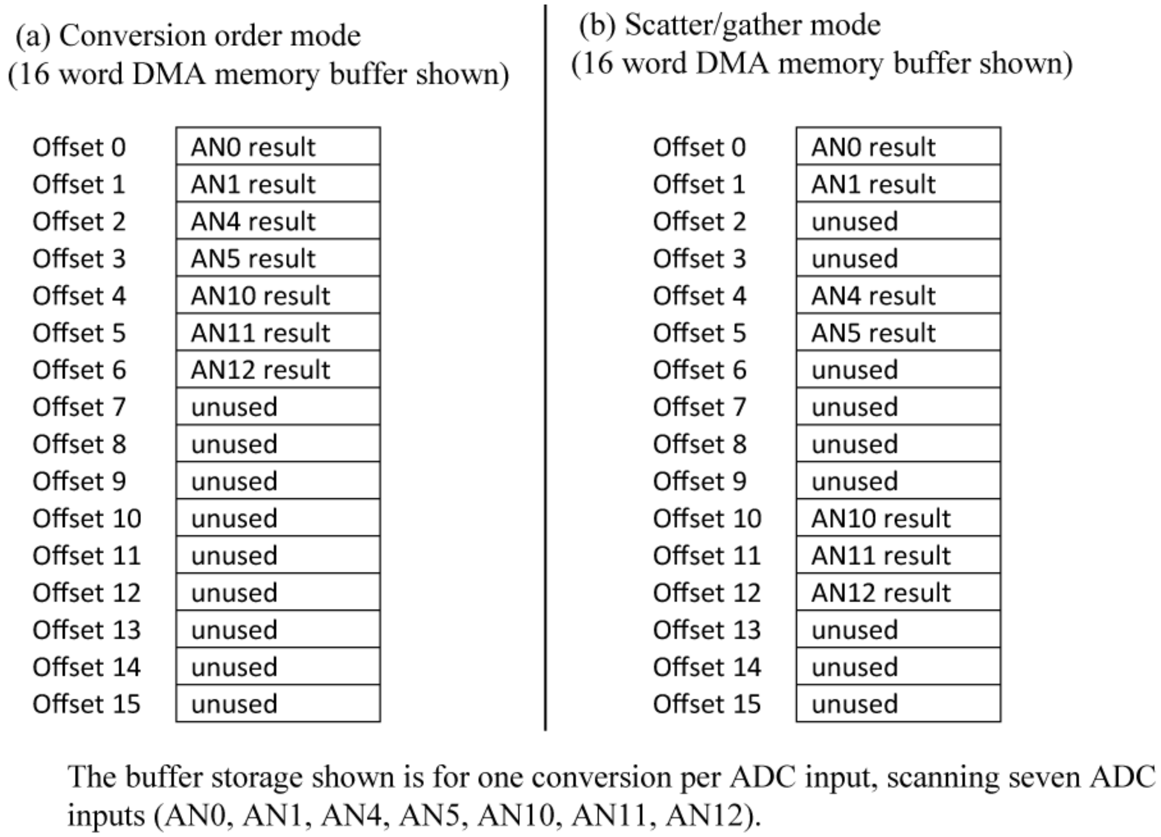
However, modes that automatically perform multiple conversions, such as the scan mode, operate differently on PIC24 CPUs with DMA. In these cases, the DMA module must be used to store the multiple ADC conversion results instead of them being placed in the `ADCxBUF1` through `ADCxBUFF` registers. The DMA module gives more flexibility in terms of the number of conversions performed and how they are stored in memory. The ADC code examples in Figures 11.17/19 (automatic scanning), Figure 11.21 (automatic scanning), Figure 11.22 (automatic scanning with ping-pong buffers), and Figures 11.25/11.26 are not appropriate for PIC24 CPUs with DMA.

AUTOMATED CHANNEL SCANNING WITH DMA

Our first example uses the same hardware setup as described in Figure 11.16. Our goal is to perform automated channel scanning on seven ADC inputs (AN0, AN1, AN4, AN5, AN10, AN11, AN12). In the code example of Figure 11.17, we noted that these results were placed in ADC registers `ADCxBUF0` through `ADCxBUF6` and that the ADC ISR interrupt occurred after the seven conversions had been completed. The ADC ISR then copied the registers to a memory buffer named `au16_buffer[]`. The `main()` code of Figure 11.19 printed these to the console as shown in Figure 11.20.

The presence of the DMA module requires the following changes:

- The ADC ISR is no longer used, as the ADC interrupt now occurs after each conversion, at which point the DMA module transfers the result to a user-specified DMA buffer. The DMA interrupt is used instead to indicate when the seven conversions have finished and the DMA ISR is used to copy these to another memory buffer.
- A DMA channel must be configured to be linked to the ADC module and a buffer in DMA memory allocated for the ADC results. The DMA module is configured for word mode since each ADC result is larger than a byte.
- There are two choices for storing the ADC results in DMA memory, *conversion order mode* or *scatter/gather mode*. In conversion order mode, the results are stored in DMA memory in the order that the conversions are performed as shown in Figure H.1a. Thus, for our seven ADC inputs we have the following: AN0 is stored at DMA buffer offset 0, AN1 at offset 1, AN4 at offset 2, AN5 at offset 3, AN10 at offset 4, AN11 at offset 5, and AN12 at offset 6. This matches the ordering of Figure 11.20 in which the seven ADC conversions are stored in ADC registers `ADCxBUF0` through `ADCxBUF6`. In scatter/gather mode, results are stored in the DMA buffer at the offset that matches the channel number. Thus, AN0 is stored at offset 0, AN1 at offset 1, AN4 at offset 4, AN5 at offset 5, AN10 at offset 10, AN11 at offset 11, and AN12 at offset 12.



(a) Conversion order mode
(16 word DMA memory buffer shown)

Offset 0	AN0 result
Offset 1	AN1 result
Offset 2	AN4 result
Offset 3	AN5 result
Offset 4	AN10 result
Offset 5	AN11 result
Offset 6	AN12 result
Offset 7	unused
Offset 8	unused
Offset 9	unused
Offset 10	unused
Offset 11	unused
Offset 12	unused
Offset 13	unused
Offset 14	unused
Offset 15	unused

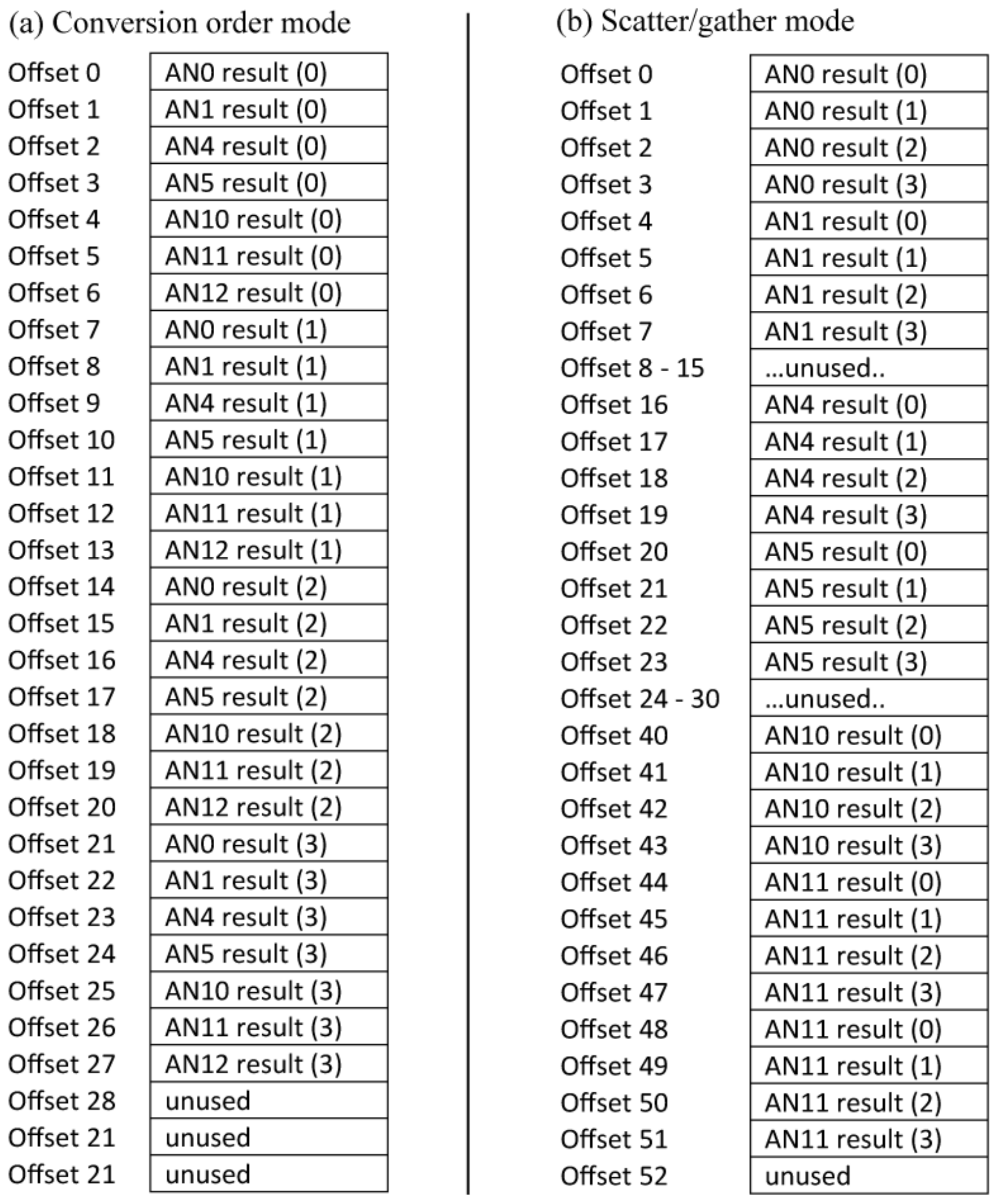
(b) Scatter/gather mode
(16 word DMA memory buffer shown)

Offset 0	AN0 result
Offset 1	AN1 result
Offset 2	unused
Offset 3	unused
Offset 4	AN4 result
Offset 5	AN5 result
Offset 6	unused
Offset 7	unused
Offset 8	unused
Offset 9	unused
Offset 10	AN10 result
Offset 11	AN11 result
Offset 12	AN12 result
Offset 13	unused
Offset 14	unused
Offset 15	unused

The buffer storage shown is for one conversion per ADC input, scanning seven ADC inputs (AN0, AN1, AN4, AN5, AN10, AN11, AN12).

Figure H.1 Conversion order mode versus gather/scatter mode for storing ADC results to DMA memory for one conversion per ADC input.

The ADC module has the capability of performing multiple conversions per ADC input during scanning. Figure H.2 shows the buffer storage using the same channels as Figure H.1, but with four conversions per ADC input. The conversion order mode is more efficient in terms of DMA memory usage since DMA memory is used for those channels not scanned during scatter/gather mode, but these locations have to be allocated. However, scatter/gather mode makes it easy to reference the conversions by input number and conversion number.



(a) Conversion order mode

Offset 0	AN0 result (0)
Offset 1	AN1 result (0)
Offset 2	AN4 result (0)
Offset 3	AN5 result (0)
Offset 4	AN10 result (0)
Offset 5	AN11 result (0)
Offset 6	AN12 result (0)
Offset 7	AN0 result (1)
Offset 8	AN1 result (1)
Offset 9	AN4 result (1)
Offset 10	AN5 result (1)
Offset 11	AN10 result (1)
Offset 12	AN11 result (1)
Offset 13	AN12 result (1)
Offset 14	AN0 result (2)
Offset 15	AN1 result (2)
Offset 16	AN4 result (2)
Offset 17	AN5 result (2)
Offset 18	AN10 result (2)
Offset 19	AN11 result (2)
Offset 20	AN12 result (2)
Offset 21	AN0 result (3)
Offset 22	AN1 result (3)
Offset 23	AN4 result (3)
Offset 24	AN5 result (3)
Offset 25	AN10 result (3)
Offset 26	AN11 result (3)
Offset 27	AN12 result (3)
Offset 28	unused
Offset 21	unused
Offset 21	unused

(b) Scatter/gather mode

Offset 0	AN0 result (0)
Offset 1	AN0 result (1)
Offset 2	AN0 result (2)
Offset 3	AN0 result (3)
Offset 4	AN1 result (0)
Offset 5	AN1 result (1)
Offset 6	AN1 result (2)
Offset 7	AN1 result (3)
Offset 8 - 15	…unused..
Offset 16	AN4 result (0)
Offset 17	AN4 result (1)
Offset 18	AN4 result (2)
Offset 19	AN4 result (3)
Offset 20	AN5 result (0)
Offset 21	AN5 result (1)
Offset 22	AN5 result (2)
Offset 23	AN5 result (3)
Offset 24 - 30	…unused..
Offset 40	AN10 result (0)
Offset 41	AN10 result (1)
Offset 42	AN10 result (2)
Offset 43	AN10 result (3)
Offset 44	AN11 result (0)
Offset 45	AN11 result (1)
Offset 46	AN11 result (2)
Offset 47	AN11 result (3)
Offset 48	AN11 result (0)
Offset 49	AN11 result (1)
Offset 50	AN11 result (2)
Offset 51	AN11 result (3)
Offset 52	unused

The buffer storage shown is for four conversions per ADC input, scanning seven ADC inputs (AN0, AN1, AN4, AN5, AN10, AN11, AN12).

Figure H.2 Conversion order mode versus scatter/gather mode for storing ADC results to DMA memory for four conversions per ADC input.

Figure H.3 shows the first part of the `configDMA_ADC()` function used to configure the ADC and DMA modules for automated scanning operations. This is a modified version of the ADC configuration code found in the Figure 11.17 and also includes DMA memory buffer allocation as originally discussed in Figure 13.4

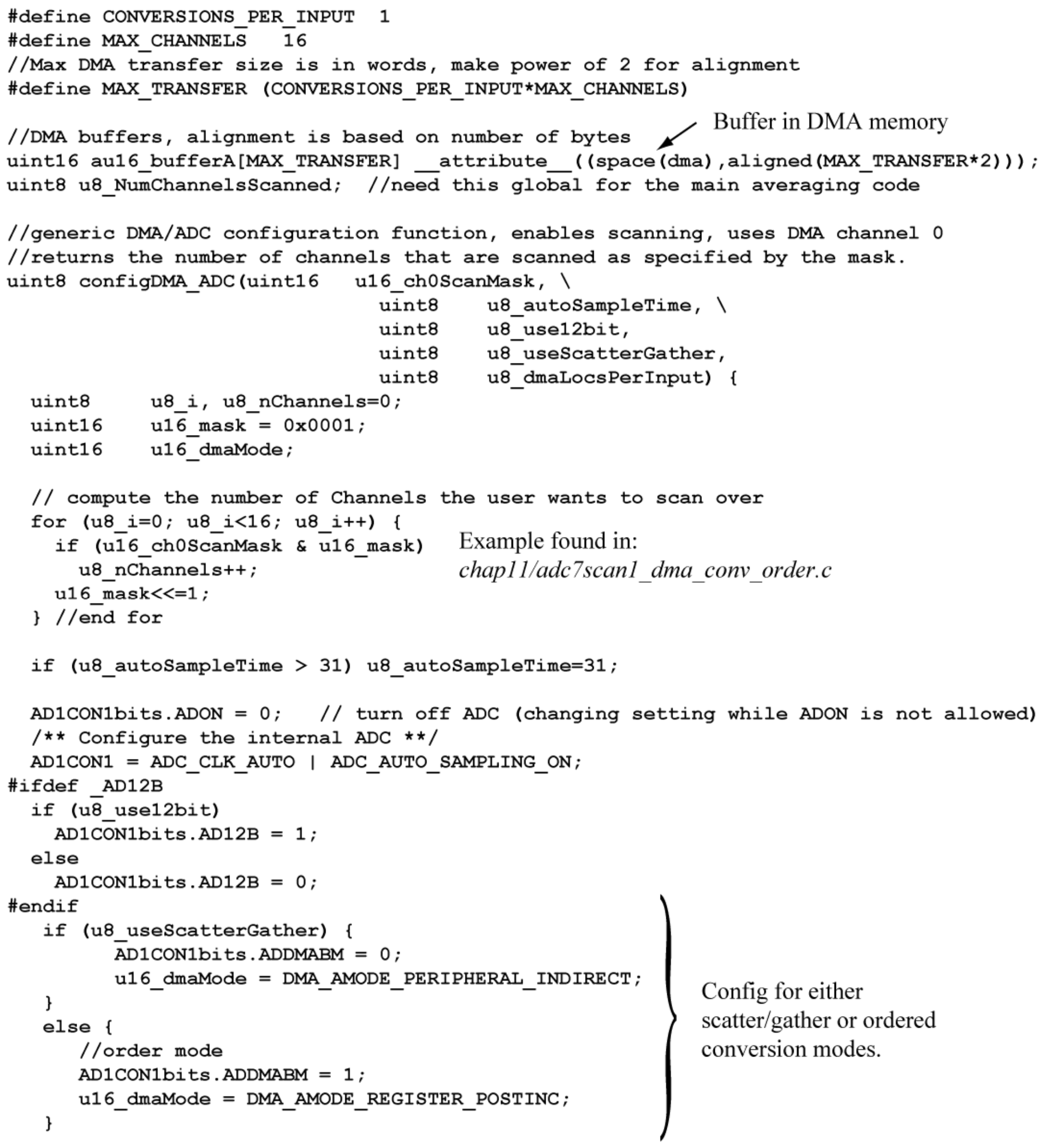
The `configDMA_ADC()` function adds two additional parameters over the parameters found in the `configADC1_AutoScanIrqCH0()` function of Figure 11.17.

- `u8_useScatterGather`: non-zero for scatter/gather mode and zero for conversion order mode.
- `u8_dmaLocsPerInput`: this specifies the number of DMA buffer locations to be used per ADC input and is only used in scatter/gather mode.

The ADDMABM bit in AD1CON1 is used to choose between scatter/gather mode (ADDMABM = 0) and conversion order mode (ADDMABM = 1). When using conversion order mode the DMA channel is configured for register post-increment addressing, while peripheral indirect is used for scatter/gather mode.

The remainder of the `configDMA_ADC()` function is given in Figure H.4. Note that the DMA request line (`DMA0REQ`) is tied to the ADC interrupt, with the DMA address register (`DMA0PAD`) set to the address of the ADC buffer register (`ADC1BUF0`). The DMA module is configured to generate an interrupt after all of the ADC conversions are performed.

The ADC is also configured to use a $T_{AD} = 10 \times T_{CY}$ instead of using the dedicated ADC internal oscillator. This is done so that we can accurately measure the T_{AD} period. At $F_{CY} = 40$ MHz, $T_{CY} = 25$ ns, so $T_{AD} = 10 \times 25$ ns $= 250$ ns.



```
#define CONVERSIONS_PER_INPUT  1
#define MAX_CHANNELS   16
//Max DMA transfer size is in words, make power of 2 for alignment
#define MAX_TRANSFER (CONVERSIONS_PER_INPUT*MAX_CHANNELS)

//DMA buffers, alignment is based on number of bytes
uint16 au16_bufferA[MAX_TRANSFER] __attribute__((space(dma),aligned(MAX_TRANSFER*2)));
uint8 u8_NumChannelsScanned;  //need this global for the main averaging code

//generic DMA/ADC configuration function, enables scanning, uses DMA channel 0
//returns the number of channels that are scanned as specified by the mask.
uint8 configDMA_ADC(uint16   u16_ch0ScanMask, \
                               uint8    u8_autoSampleTime, \
                               uint8    u8_use12bit,
                               uint8    u8_useScatterGather,
                               uint8    u8_dmaLocsPerInput) {
  uint8     u8_i, u8_nChannels=0;
  uint16    u16_mask = 0x0001;
  uint16    u16_dmaMode;

  // compute the number of Channels the user wants to scan over
  for (u8_i=0; u8_i<16; u8_i++) {
    if (u16_ch0ScanMask & u16_mask)
      u8_nChannels++;
    u16_mask<<=1;
  } //end for

  if (u8_autoSampleTime > 31) u8_autoSampleTime=31;

  AD1CON1bits.ADON = 0;   // turn off ADC (changing setting while ADON is not allowed)
  /** Configure the internal ADC **/
  AD1CON1 = ADC_CLK_AUTO | ADC_AUTO_SAMPLING_ON;
#ifdef _AD12B
  if (u8_use12bit)
    AD1CON1bits.AD12B = 1;
  else
    AD1CON1bits.AD12B = 0;
#endif
   if (u8_useScatterGather) {
         AD1CON1bits.ADDMABM = 0;
         u16_dmaMode = DMA_AMODE_PERIPHERAL_INDIRECT;
   }
   else {
      //order mode
      AD1CON1bits.ADDMABM = 1;
      u16_dmaMode = DMA_AMODE_REGISTER_POSTINC;
   }
```

Buffer in DMA memory
Example found in: *chap11/adc7scan1_dma_conv_order.c*
Config for either scatter/gather or ordered conversion modes.

Figure H.3 ADC/DMA configuration code for channel scanning, part 1

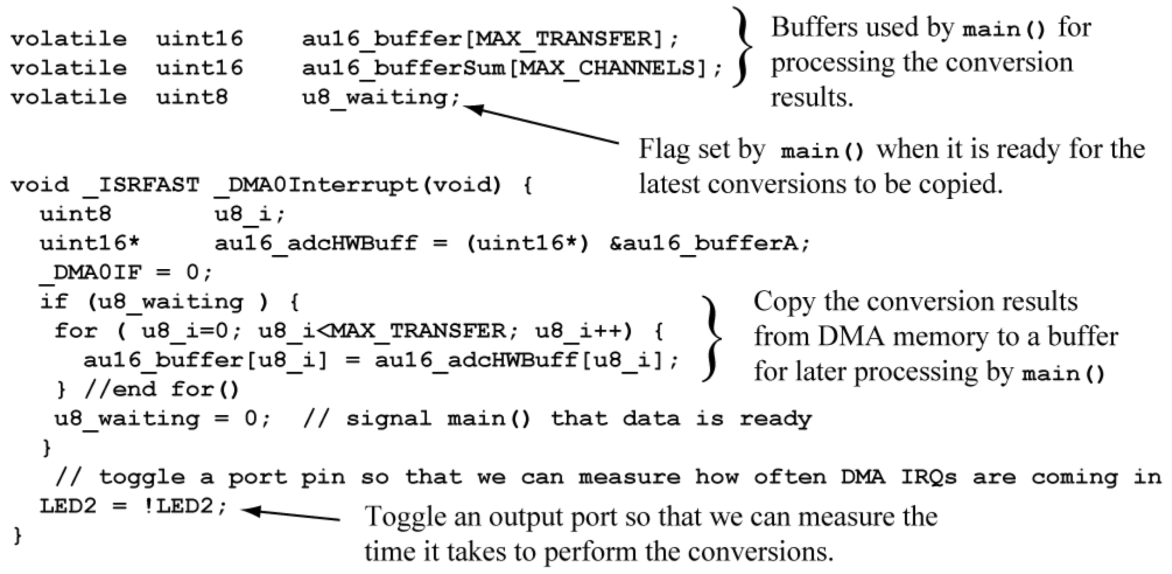
```
//at FCY = 40 MHz, Tcy = 25 ns, and use ADC clock = 10* Tcy = 10 * 25 ns = 250 ns
//use clock based on Tcy so that we can accurately measure ADC clock period
AD1CON3 = ADC_CONV_CLK_SYSTEM | (u8_autoSampleTime<<8) |ADC_CONV_CLK_10Tcy;
//Note: PIC24H family reference manual (16.13.2) says that for
// 'ordered' mode, the SMPI bits should be cleared. However,
// when scanning, this seems to be incorrect as the
//settings that work are the same ones used for 'scatter/gather' mode.
AD1CON2 = ADC_VREF_AVDD_AVSS | ADC_CONVERT_CH0 | ADC_SCAN_ON | ((u8_nChannels-1)<<2);

#ifdef __PIC24H__
  AD1CHS0 = ADC_CH0_NEG_SAMPLEA_VREFN;
#else
  AD1CHS = ADC_CH0_NEG_SAMPLEA_VREFN;
#endif
  AD1CSSL = u16_ch0ScanMask;

  //AD1CON4 is only used in scatter-gather mode
  switch (u8_dmaLocsPerInput) {
   case 1   :  AD1CON4 = ADC_1_WORD_PER_INPUT;break;
   case 2   :  AD1CON4 = ADC_2_WORD_PER_INPUT;break;
   case 4   :  AD1CON4 = ADC_4_WORD_PER_INPUT;break;
   case 8   :  AD1CON4 = ADC_8_WORD_PER_INPUT;break;
   case 16  :  AD1CON4 = ADC_16_WORD_PER_INPUT;break;
   case 32  :  AD1CON4 = ADC_32_WORD_PER_INPUT;break;
   case 64  :  AD1CON4 = ADC_64_WORD_PER_INPUT;break;
   case 128 :  AD1CON4 = ADC_128_WORD_PER_INPUT;break;
    default: AD1CON4 = ADC_1_WORD_PER_INPUT;break;
  }

 //configure the DMA channel 0 interrupt
  DMA0PAD = (unsigned int) &ADC1BUF0;
  DMA0REQ = DMA_IRQ_ADC1;
  DMA0STA = __builtin_dmaoffset(au16_bufferA);
  DMA0CNT = (u8_nChannels * u8_dmaLocsPerInput)-1;
  DMA0CON =   //configure and enable the module Module
    (DMA_MODULE_ON |
     DMA_SIZE_WORD |
     DMA_DIR_READ_PERIPHERAL |
     DMA_INTERRUPT_FULL |
     DMA_NULLW_OFF |
     u16_dmaMode |
     DMA_MODE_CONTINUOUS);

   _DMA0IF = 0;
   _DMA0IP = 6;
   _DMA0IE = 1;

   AD1CON1bits.ADON = 1;   // turn on the ADC
   return(u8_nChannels);
}
```

Example found in:
chap11/adc7scan1_dma_conv_order.c

DMA configured to interrupt after all conversions are finished.

Figure H.4 ADC/DMA configuration code for channel scanning, part 2

Figure H.5 shows the DMA ISR code used in our application. The DMA ISR is triggered whenever the ADC conversions are finished, which are the number of inputs scanned multiplied by the number of conversions performed per input. The `main()` code sets the `u8_waiting` flag to non-zero when it is ready for the latest ADC conversion results to be copied from DMA memory to a local array named `au16_buffer[]`. The `LED2` port is toggled each time the DMA ISR is executed so that we can measure how long it takes to perform the ADC conversions.



```
volatile  uint16    au16_buffer[MAX_TRANSFER];
volatile  uint16    au16_bufferSum[MAX_CHANNELS];
volatile  uint8     u8_waiting;

void _ISRFAST _DMA0Interrupt(void) {
  uint8       u8_i;
  uint16*     au16_adcHWBuff = (uint16*) &au16_bufferA;
  _DMA0IF = 0;
  if (u8_waiting ) {
   for ( u8_i=0; u8_i<MAX_TRANSFER; u8_i++) {
     au16_buffer[u8_i] = au16_adcHWBuff[u8_i];
   } //end for()
   u8_waiting = 0;  // signal main() that data is ready
  }
   // toggle a port pin so that we can measure how often DMA IRQs are coming in
  LED2 = !LED2;
}
```

Buffers used by main() for processing the conversion results.
Flag set by main() when it is ready for the latest conversions to be copied.
Copy the conversion results from DMA memory to a buffer for later processing by main()
Toggle an output port so that we can measure the time it takes to perform the conversions.

Figure H.5 DMA ISR code

Figure H.6 shows the `main()` code for processing the results that are produced by conversion order mode. The nested `for(){}` loops in the `while(1){}` loop averages the results obtained per ADC input. The outer loop performs `CONVERSIONS_PER_INPUT` iterations while the inner loop iterations are equal to the number of ADC inputs that are scanned. After averaging the results for each ADC input, the results are printed to the console.

The `u8_waiting` flag is used by `main()` to signal the DMA ISR when it is ready for new results to be copied into the `au16_buffer[]` array. This is needed so that the `au16_buffer[]` array has stable values in it while the results are being averaged.

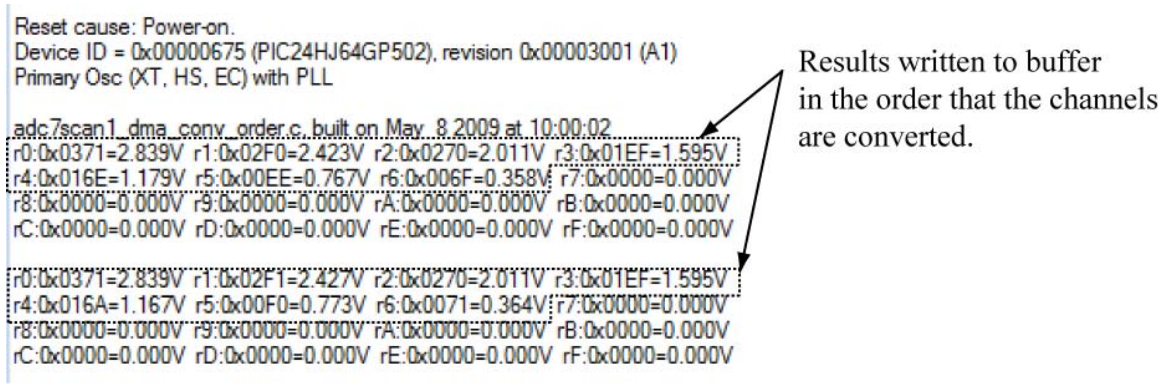
Example found in:
chap11/adc7scan1_dma_conv_order.c

```
int main (void) {
  uint8   u8_i, u8_j, u8_k;
  uint16  u16_pot;
  float   f_pot;

  configBasic(HELLO_MSG);
  CONFIG_AN0_AS_ANALOG();CONFIG_AN1_AS_ANALOG(); CONFIG_AN4_AS_ANALOG();
  CONFIG_AN5_AS_ANALOG();CONFIG_AN10_AS_ANALOG();  CONFIG_AN11_AS_ANALOG();
  CONFIG_AN12_AS_ANALOG();
  CONFIG_LED2();

  u8_NumChannelsScanned = configDMA_ADC( ADC_SCAN_AN0 | ADC_SCAN_AN1 | ADC_SCAN_AN4 |     \
                             ADC_SCAN_AN5 | ADC_SCAN_AN10 | ADC_SCAN_AN11 | ADC_SCAN_AN12,
                             31, ADC_12BIT_FLAG, 0, CONVERSIONS_PER_INPUT);

  u8_waiting = 1;
  while (1) {
    while (u8_waiting){};  // wait for valid data in ISR
    //data is updated in array by DMA ISR when u8_waiting flag is cleared
    //iterate over channels, and average results for each channel
    //data in array will not be updated again by DMA ISR until u8_waiting flag is set.
    u8_k = 0;  //buffer index
    for (u8_j=0; u8_j<CONVERSIONS_PER_INPUT; u8_j++) {
     for ( u8_i=0; u8_i<u8_NumChannelsScanned; u8_i++) {  //each channel
      //each result per channel
         if (u8_j == 0) au16_bufferSum[u8_i] = au16_buffer[u8_k];
            else au16_bufferSum[u8_i] += au16_buffer[u8_k];
         u8_k++;
      }
    }
    //zero out unused channels
    for (u8_i=u8_NumChannelsScanned; u8_i<MAX_CHANNELS ;u8_i++) {
      au16_bufferSum[u8_i] = 0;
    }
    //now average and print
    for ( u8_i=0; u8_i<MAX_CHANNELS ; u8_i++) {
      u16_pot = au16_bufferSum[u8_i]/CONVERSIONS_PER_INPUT; //take the average
      f_pot = 3.3 / ADC_NSTEPS * u16_pot;
      printf("r");
      if (u8_i < 10) outChar( '0'+u8_i );
      else outChar( 'A'-10+u8_i );
      printf(":0x%04X=%1.3fV  ",  u16_pot, (double) f_pot );
      if ((u8_i % 4) == 3) printf("\n");
    } //end for()
    printf("\n");
    u8_waiting = 1;
    doHeartbeat();
    DELAY_MS(1500);
  } //endof while()
} // endof main()
```

Selects conversion order mode.

Figure H.6 `main()` code for processing results produced by conversion order mode.

Figure H.7 shows the results of the conversion order mode. Observe that the first seven buffer locations have the conversions from our scanned channels (AN0, AN1, AN4, AN5, AN10, AN11, AN12). This is a similar result to that shown in Figure 11.20 for a PIC24 CPU without DMA.



```
Reset cause: Power-on.
Device ID = 0x00000675 (PIC24HJ64GP502), revision 0x00003001 (A1)
Primary Osc (XT, HS, EC) with PLL

adc7scan1_dma_conv_order.c, built on May  8 2009 at 10:00:02
r0:0x0371=2.839V r1:0x02F0=2.423V r2:0x0270=2.011V r3:0x01EF=1.595V
r4:0x016E=1.179V r5:0x00EE=0.767V r6:0x006F=0.358V r7:0x0000=0.000V
r8:0x0000=0.000V r9:0x0000=0.000V rA:0x0000=0.000V rB:0x0000=0.000V
rC:0x0000=0.000V rD:0x0000=0.000V rE:0x0000=0.000V rF:0x0000=0.000V

r0:0x0371=2.839V r1:0x02F1=2.427V r2:0x0270=2.011V r3:0x01EF=1.595V
r4:0x016A=1.167V r5:0x00F0=0.773V r6:0x0071=0.364V r7:0x0000=0.000V
r8:0x0000=0.000V r9:0x0000=0.000V rA:0x0000=0.000V rB:0x0000=0.000V
rC:0x0000=0.000V rD:0x0000=0.000V rE:0x0000=0.000V rF:0x0000=0.000V
```

Figure H.7 Conversion order mode results

Figure H.8 shows the `main()` code for testing scatter/gather mode. This code reverses the iteration order of nested `for(){}` loops that performs the averaging in order to match the data order of Figure H.2b. Otherwise, the code of Figure H.8 is similar to that of Figure H.6.

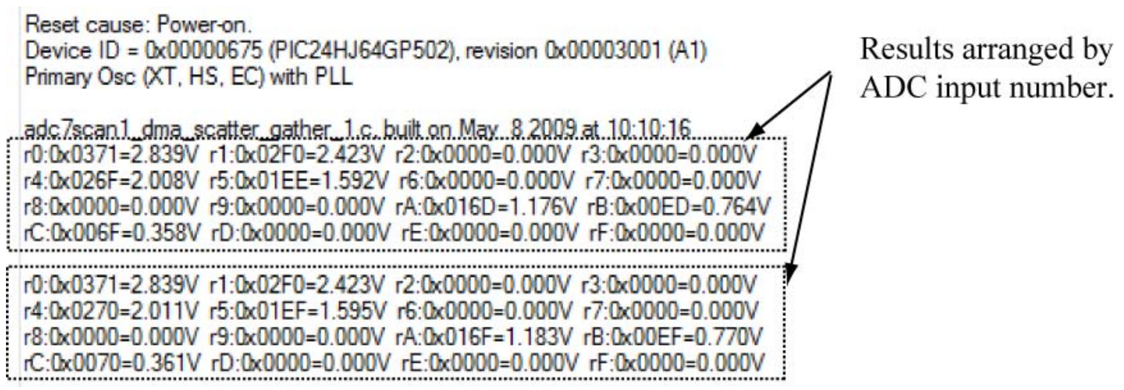
Example found in:
chap11/adc7scan1_scatter_gather_1.c

```
int main (void) {
  uint8   u8_i, u8_j, u8_k;
  uint16  u16_sum;
  uint16  u16_pot;
  float   f_pot;

  configBasic(HELLO_MSG);

  CONFIG_AN0_AS_ANALOG(); CONFIG_AN1_AS_ANALOG();   CONFIG_AN4_AS_ANALOG();
  CONFIG_AN5_AS_ANALOG(); CONFIG_AN10_AS_ANALOG(); CONFIG_AN11_AS_ANALOG();
  CONFIG_AN12_AS_ANALOG();
  CONFIG_LED2();

  configDMA_ADC( ADC_SCAN_AN0 | ADC_SCAN_AN1 | ADC_SCAN_AN4 |     \
                           ADC_SCAN_AN5 | ADC_SCAN_AN10 | ADC_SCAN_AN11 | ADC_SCAN_AN12,
                           31, ADC_12BIT_FLAG, 1, CONVERSIONS_PER_INPUT);

  u8_waiting = 1;
  while (1) {
    while (u8_waiting){};  // wait for valid data in ISR
    //data is updated in array by DMA ISR when u8_waiting flag is cleared
    //iterate over channels, and average results for each channel
      //data in array will not be updated again by DMA ISR until u8_waiting flag is set.
    u8_k = 0;  //buffer index
    for ( u8_i=0; u8_i<16; u8_i++) {  //each channel
      for (u8_j=0; u8_j<CONVERSIONS_PER_INPUT; u8_j++) {  //each result per channel
         if (u8_j == 0) u16_sum = au16_buffer[u8_k];
            else u16_sum += au16_buffer[u8_k];
         u8_k++;
      }
      u16_pot = u16_sum/CONVERSIONS_PER_INPUT; //take the average
      f_pot = 3.3 / ADC_NSTEPS * u16_pot;
      printf("r");
      if (u8_i < 10) outChar( '0'+u8_i );
      else outChar( 'A'-10+u8_i );
      printf(":0x%04X=%1.3fV  ",  u16_pot, (double) f_pot );
      if ((u8_i % 4) == 3) printf("\n");
    } //end for()
    printf("\n");
    u8_waiting = 1;
    doHeartbeat();
    DELAY_MS(1500);
  } //endof while()
} // endof main()
```

Selects scatter/gather mode.

Figure H.8 `main()` code for processing results produce by scatter-gather order mode.

Figure H.9 shows the console output for scatter/gather mode. Observe that values are arranged by ADC input number.



```
Reset cause: Power-on.
Device ID = 0x00000675 (PIC24HJ64GP502), revision 0x00003001 (A1)
Primary Osc (XT, HS, EC) with PLL

adc7scan1_dma_scatter_gather_1.c, built on May 8 2009 at 10:10:16
r0:0x0371=2.839V r1:0x02F0=2.423V r2:0x0000=0.000V r3:0x0000=0.000V
r4:0x026F=2.008V r5:0x01EE=1.592V r6:0x0000=0.000V r7:0x0000=0.000V
r8:0x0000=0.000V r9:0x0000=0.000V rA:0x016D=1.176V rB:0x00ED=0.764V
rC:0x006F=0.358V rD:0x0000=0.000V rE:0x0000=0.000V rF:0x0000=0.000V

r0:0x0371=2.839V r1:0x02F0=2.423V r2:0x0000=0.000V r3:0x0000=0.000V
r4:0x0270=2.011V r5:0x01EF=1.595V r6:0x0000=0.000V r7:0x0000=0.000V
r8:0x0000=0.000V r9:0x0000=0.000V rA:0x016F=1.183V rB:0x00EF=0.770V
rC:0x0070=0.361V rD:0x0000=0.000V rE:0x0000=0.000V rF:0x0000=0.000V
```

Figure H.9 Scatter/gather mode results

Recall that the DMA ISR toggled the LED2 output. Figure H.10(a) shows the timing results for the seven ADC inputs for the case of one conversion per input. In 10-bit mode, each conversion takes 10.75 µs for T_{AD} = 250 ns (Recall that the ADC was configured for $T_{AD} = 10 \times T_{CY}$ with F_{CY} = 40 MHz, T_{CY} = 25 ns, so $T_{AD} = 10 \times 25$ ns = 250 ns). Each conversion takes 43 T_{AD} periods: 31 T_{AD} for sampling, 10 T_{AD} for the conversion itself, and 2 T_{AD} overhead. So, one conversion = 43 × 250 ns = 10.75 µs.

The time for seven conversions is then 10.75 µs × 7 = 75.25 µs, which matched the measured half-period of the LED2 as shown in the logic analyzer screenshot.

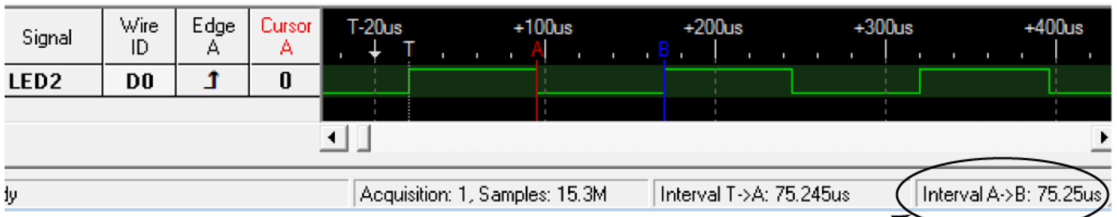
Figure H.10(b) shows the timing results for the seven ADC inputs for the case of four conversions per input. This causes the DMA ISR interrupt time to increase by a factor of four, or 4 × 75.25 µs = 301 µs.

One Conversion time = Tsamp + Tconv + Toverhead
Tsamp = 31 × TAD ; Tconv = 10 bits × TAD ; Toverhead = 2 × TAD
One Conversion tiime = (31 + 10 + 2) × TAD = 43 × TAD;

Test conditions: TCY = 25 ns (FCY = 40 MHz); TAD = 10 × TCY = 10 × 25 ns = 250 ns

One Conversion time = 43 × TAD = 43 × 250 ns = 10.75 μs

(a) For 7 ADC inputs to be scanned, with one conversion per input, the DMA ISR will be called every 7 × Conversion Time = 7 × 10.75 μs = 75.25 μs.
The logic analyzer screen shot below shows the LED2 output begin toggled in the DMA ISR every 75.25 μs.

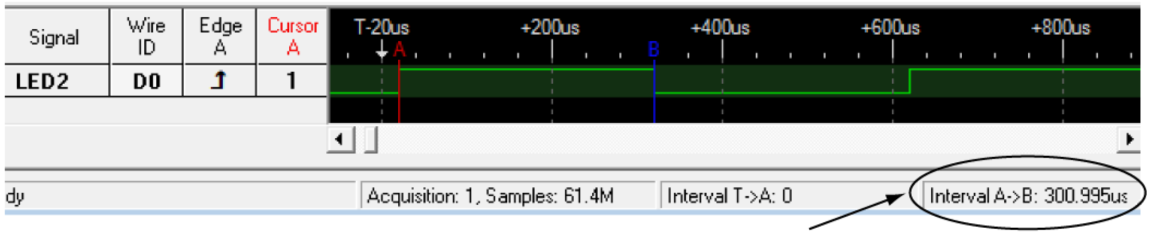


Signal	Wire ID	Edge A	Cursor A
LED2	D0	↥	0

Acquisition: 1, Samples: 15.3M
Interval T->A: 75.245us
Interval A->B: 75.25us

The logic analyzer screen shot above shows the LED2 output begin toggled in the DMA ISR every 75.25 μs.

(b) If CONVERSIONS_PER_INPUT is changed from 1 to 4, then the total number of conversions increases by × 4, so the new DMA ISR interrupt time is 4 × 75.25 μs = 301 μs.



Signal	Wire ID	Edge A	Cursor A
LED2	D0	↥	1

Acquisition: 1, Samples: 61.4M
Interval T->A: 0
Interval A->B: 300.995us

The logic analyzer screen shot above shows the LED2 output begin toggled in the DMA ISR every 301 μs after CONVERSIONS_PER_INPUT was changed to 4.

H.10 Timing results for LED2 output toggled by the DMA ISR

PING-PONG BUFFERING

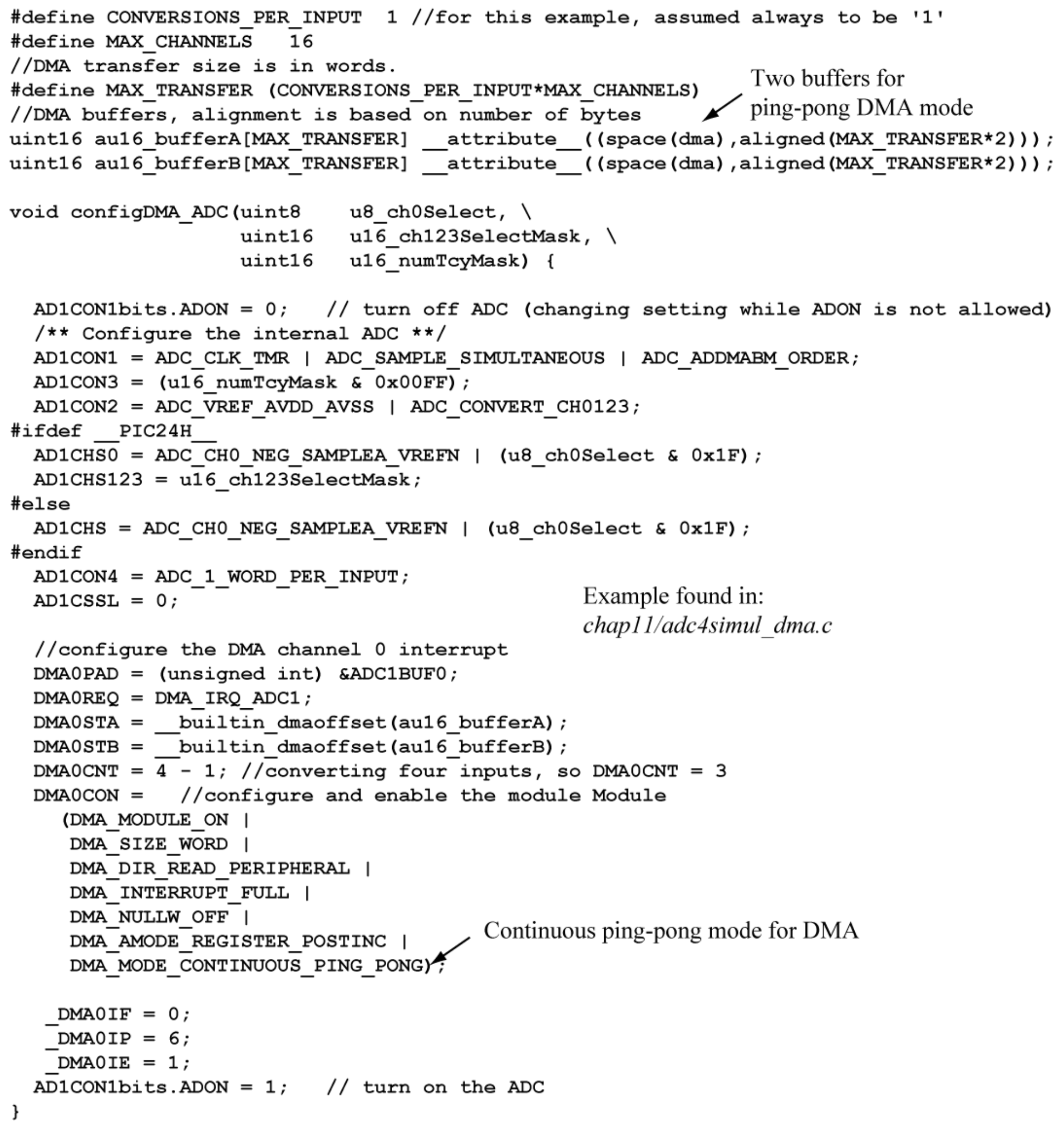
Figure 11.21 discussed how to use the ping-pong buffering mode of ADC module in a PIC24 CPU without DMA. This was accomplished by setting the buffer fill mode select bit (BUFM) of ADxCON2, causing the ADC to write to the first half of the 16-entry hardware buffer on the first interrupt, and the second half on the next interrupt. The ADC in a PIC24 CPU with DMA does not have this capability since the hardware buffer is only one register. However, ping-pong buffering is supported by a DMA mode as discussed in Figure 13.4. Thus, to use ping-pong buffering, you would simply allocate two DMA buffers, and use the DMA ping-pong mode for storage in a similar manner as done in Figure 13.4.

SIMULTANEOUS SAMPLING EXAMPLE

The code in Figures 11.25 and 11.26 performed simultaneous sampling on four channels (AN12 – Channel 0, AN1 – Channel 1, AN2 – Channel 2, AN3 – Channel 3), averaged them over 64 samples, and printed the results. The ADC module was also configured for ping-pong buffering.

The next example accomplishes the same task, except for a PIC24 with DMA. The test setup is the same as Figure 11.16 (8 resistor string), except input AN2/RB0 replaces input AN4/RB2.

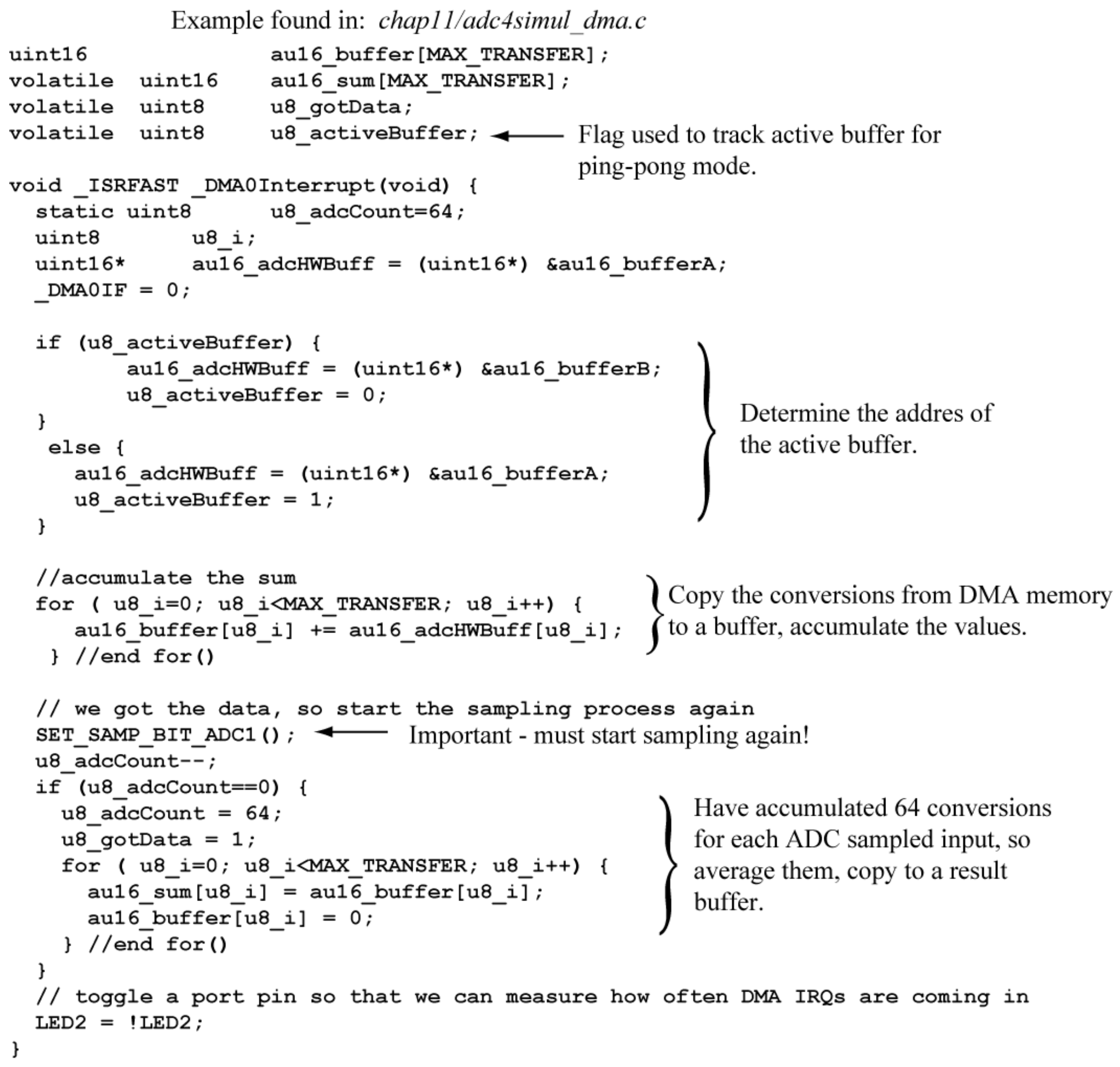
Figure H.11 shows the `configDMA_ADC()` code that accomplishes the same functionality as the `configADC1_Simul4ChanIrq()` function of Figure 11.25. The ADC is configured for simultaneous sampling of inputs AN12, AN1, AN2, AN3 and uses a Timer3 expiration event to begin conversion. Ping-pong buffering is accomplished by allocating two buffers in DMA memory, and then configuring the DMA module for continuous ping-pong mode as was originally done for the code in Figure 13.4. For simplicity, the `configDMA_ADC()` function always configures for ordered conversion mode and for one sample per ADC input. The ADC is configured for a manual sampling start.


Two buffers for ping-pong DMA mode
Example found in: chap11/adc4simul_dma.c
Continuous ping-pong mode for DMA

```
#define CONVERSIONS_PER_INPUT  1 //for this example, assumed always to be '1'
#define MAX_CHANNELS   16
//DMA transfer size is in words.
#define MAX_TRANSFER (CONVERSIONS_PER_INPUT*MAX_CHANNELS)
//DMA buffers, alignment is based on number of bytes
uint16 au16_bufferA[MAX_TRANSFER] __attribute__((space(dma),aligned(MAX_TRANSFER*2)));
uint16 au16_bufferB[MAX_TRANSFER] __attribute__((space(dma),aligned(MAX_TRANSFER*2)));

void configDMA_ADC(uint8    u8_ch0Select, \
                   uint16   u16_ch123SelectMask, \
                   uint16   u16_numTcyMask) {

  AD1CON1bits.ADON = 0;   // turn off ADC (changing setting while ADON is not allowed)
  /** Configure the internal ADC **/
  AD1CON1 = ADC_CLK_TMR | ADC_SAMPLE_SIMULTANEOUS | ADC_ADDMABM_ORDER;
  AD1CON3 = (u16_numTcyMask & 0x00FF);
  AD1CON2 = ADC_VREF_AVDD_AVSS | ADC_CONVERT_CH0123;
#ifdef __PIC24H__
  AD1CHS0 = ADC_CH0_NEG_SAMPLEA_VREFN | (u8_ch0Select & 0x1F);
  AD1CHS123 = u16_ch123SelectMask;
#else
  AD1CHS = ADC_CH0_NEG_SAMPLEA_VREFN | (u8_ch0Select & 0x1F);
#endif
  AD1CON4 = ADC_1_WORD_PER_INPUT;
  AD1CSSL = 0;

  //configure the DMA channel 0 interrupt
  DMA0PAD = (unsigned int) &ADC1BUF0;
  DMA0REQ = DMA_IRQ_ADC1;
  DMA0STA = __builtin_dmaoffset(au16_bufferA);
  DMA0STB = __builtin_dmaoffset(au16_bufferB);
  DMA0CNT = 4 - 1; //converting four inputs, so DMA0CNT = 3
  DMA0CON =   //configure and enable the module Module
     (DMA_MODULE_ON |
      DMA_SIZE_WORD |
      DMA_DIR_READ_PERIPHERAL |
      DMA_INTERRUPT_FULL |
      DMA_NULLW_OFF |
      DMA_AMODE_REGISTER_POSTINC |
      DMA_MODE_CONTINUOUS_PING_PONG);

   _DMA0IF = 0;
   _DMA0IP = 6;
   _DMA0IE = 1;
  AD1CON1bits.ADON = 1;   // turn on the ADC
}
```

H.11 ADC/DMA configuration code for simultaneous sampling example

Figure H.12 shows the DMA ISR for the simultaneous sampling example. It is similar in nature to the ADC ISR used in Figure 11.25 for the original simultaneous sampling example of Chapter 11. The code determines the active ping-pong buffer and then accumulates these to a temporary buffer (`au16_buffer[]`). Once 64 conversions have been accumulated, the values are averaged to a buffer named `au16_sum[]` and the `u8_goData` flag is set to signal `main()` that the averaged values are ready. The `main()` code for this example is the same as found in Figure 11.26, except the `configADC1_Simul4ChanIrq()` function call has been replaced by the `configDMA_ADC()` function shown in Figure H.11.



Example found in: *chap11/adc4simul_dma.c*

```
uint16              au16_buffer[MAX_TRANSFER];
volatile  uint16    au16_sum[MAX_TRANSFER];
volatile  uint8     u8_gotData;
volatile  uint8     u8_activeBuffer;

void _ISRFAST _DMA0Interrupt(void) {
  static uint8      u8_adcCount=64;
  uint8       u8_i;
  uint16*     au16_adcHWBuff = (uint16*) &au16_bufferA;
  _DMA0IF = 0;

  if (u8_activeBuffer) {
         au16_adcHWBuff = (uint16*) &au16_bufferB;
         u8_activeBuffer = 0;
  }
   else {
     au16_adcHWBuff = (uint16*) &au16_bufferA;
     u8_activeBuffer = 1;
  }

  //accumulate the sum
  for ( u8_i=0; u8_i<MAX_TRANSFER; u8_i++) {
     au16_buffer[u8_i] += au16_adcHWBuff[u8_i];
   } //end for()

  // we got the data, so start the sampling process again
  SET_SAMP_BIT_ADC1();
  u8_adcCount--;
  if (u8_adcCount==0) {
    u8_adcCount = 64;
    u8_gotData = 1;
    for ( u8_i=0; u8_i<MAX_TRANSFER; u8_i++) {
      au16_sum[u8_i] = au16_buffer[u8_i];
      au16_buffer[u8_i] = 0;
    } //end for()
  }
  // toggle a port pin so that we can measure how often DMA IRQs are coming in
  LED2 = !LED2;
}
```

Flag used to track active buffer for ping-pong mode.

Determine the addres of the active buffer.

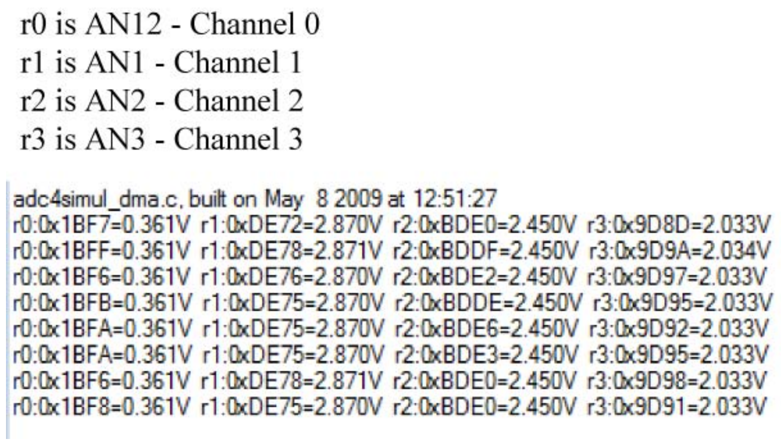
Copy the conversions from DMA memory to a buffer, accumulate the values.

Important - must start sampling again!

Have accumulated 64 conversions for each ADC sampled input, so average them, copy to a result buffer.

H.12 DMA ISR code for simultaneous sampling example

Figure H.13 shows the console results for the simultaneous sampling example. Note that r0 corresponds to AN12 – Channel 0 (first conversion), r1 to AN1 – Channel 1 (second conversion), r2 to AN2 – Channel 2 (third conversion), and r3 to AN3 – Channel 3 (fourth conversion). The values for AN12, AN1, AN2 correlate with results found in Figures H.7 and H.9.



r0 is AN12 - Channel 0
r1 is AN1 - Channel 1
r2 is AN2 - Channel 2
r3 is AN3 - Channel 3

```
adc4simul_dma.c, built on May  8 2009 at 12:51:27
r0:0x1BF7=0.361V r1:0xDE72=2.870V r2:0xBDE0=2.450V r3:0x9D8D=2.033V
r0:0x1BFF=0.361V r1:0xDE78=2.871V r2:0xBDDF=2.450V r3:0x9D9A=2.034V
r0:0x1BF6=0.361V r1:0xDE76=2.870V r2:0xBDE2=2.450V r3:0x9D97=2.033V
r0:0x1BFB=0.361V r1:0xDE75=2.870V r2:0xBDDE=2.450V r3:0x9D95=2.033V
r0:0x1BFA=0.361V r1:0xDE75=2.870V r2:0xBDE6=2.450V r3:0x9D92=2.033V
r0:0x1BFA=0.361V r1:0xDE75=2.870V r2:0xBDE3=2.450V r3:0x9D95=2.033V
r0:0x1BF6=0.361V r1:0xDE78=2.871V r2:0xBDE0=2.450V r3:0x9D98=2.033V
r0:0x1BF8=0.361V r1:0xDE75=2.870V r2:0xBDE0=2.450V r3:0x9D91=2.033V
```

H.13 Results for simultaneous sampling example

SUMMARY

If only single ADC conversions are being used (no scanning or simultaneous sampling), then the code examples in Chapter 11 work the same on PIC24 CPUs with or without DMA because the result is always written to `ADCxBUF0`. However, if multiple conversions are done, then PIC24 CPUs without DMA have 15 additional registers named `ADCxBUF1` through `ADCxBUFF` for buffering these results, and the ADC interrupt is used to signal when a group of conversions is finished. For PIC24 CPUs with DMA, these registers are not present, and DMA memory is used to buffer the results, with the DMA interrupt occurring when a group of conversions has finished.