Section 22. Direct Memory Access (DMA)

HIGHLIGHTS

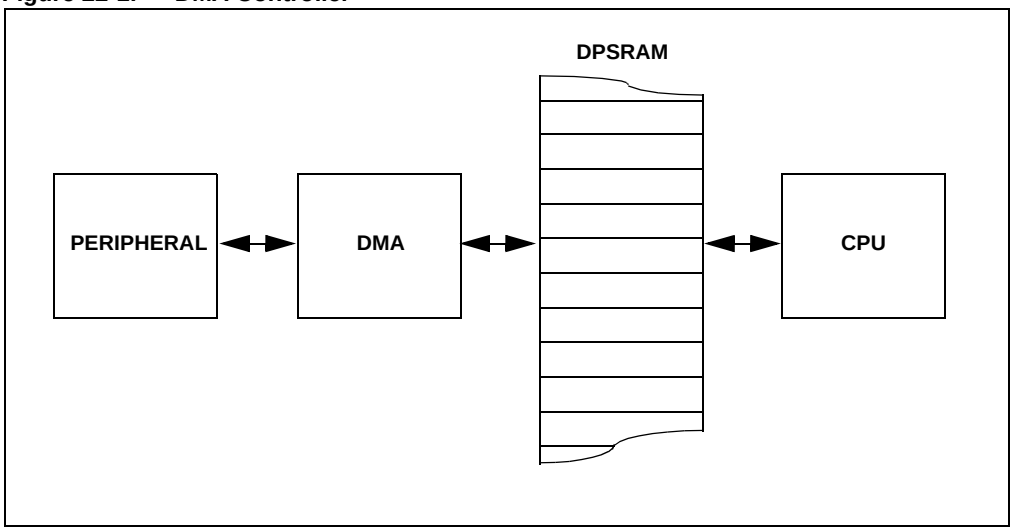
This section of the manual contains the following topics:

22.1 INTRODUCTION

The Direct Memory Access (DMA) controller is an important subsystem in Microchip's high-performance 16-bit Digital Signal Controller (DSC) families. This subsystem facilitates the transfer of data between the CPU and its peripheral without CPU assistance. The dsPIC33F DMA controller is optimized for high-performance, real-time, embedded applications where determinism and system latency are priorities.

The DMA controller transfers data between peripheral data registers and data space SRAM. The dsPIC33F DMA subsystem uses dual-ported SRAM memory (DPSRAM) and register structures that allow the DMA to operate across its own, independent address and data buses with no impact on CPU operation. This architecture eliminates the need for cycle stealing, which halts the CPU when a higher priority DMA transfer is requested. Both the CPU and DMA controller can write and read to/from addresses within data space without interference, such as CPU stalls, resulting in maximized, real-time performance. Alternatively, DMA operation and data transfer to/from the memory and peripherals are not impacted by CPU processing. For example, when a Run-Time Self-Programming (RTSP) operation is performed, the CPU does not execute any instructions until RTSP is finished. This condition, however, does not impact data transfer to/from memory and the peripherals.

Figure 22-1: DMA Controller



The DMA controller supports eight independent channels. Each channel can be configured for transfers to or from selected peripherals. Peripherals supported by the DMA controller include:

- ECAN™ technology
- Data Converter Interface (DCI)
- 10-bit/12-bit Analog-to-Digital Converter (ADC)
- Serial Peripheral Interface (SPI)
- UART
- Input Capture
- Output Compare

In addition, DMA transfers can be triggered by Timers as well as external interrupts. Each DMA channel is unidirectional. Two DMA channels must be allocated to read **and** write to a peripheral. Should more than one channel receive a request to transfer data, a simple fixed priority scheme, based on channel number, dictates which channel completes the transfer and which channel, or channels, are left pending. Each DMA channel moves a block of up to 1024 data elements, after which it generates an interrupt to the CPU to indicate that the block is available for processing.

The DMA controller provides these functional capabilities:

- Eight DMA channels
- Register Indirect With Post-increment Addressing mode
- Register Indirect Without Post-increment Addressing mode
- Peripheral Indirect Addressing mode (peripheral generates destination address)
- CPU interrupt after half or full-block transfer complete
- Byte or word transfers
- Fixed priority channel arbitration
- Manual (software) or Automatic (peripheral DMA requests) transfer initiation
- One-Shot or Auto-Repeat block transfer modes
- Ping-Pong mode (automatic switch between two DPSRAM start addresses after each block transfer complete)
- DMA request for each channel can be selected from any supported interrupt source
- Debug support features

22.2 DMA REGISTERS

Each DMA channel has a set of six status and control registers.

- **DMAxCON: DMA Channel x Control Register**

  This register configures the corresponding DMA channel by enabling/disabling the channel, specifying data transfer size, direction and block interrupt method, and selecting DMA Channel Addressing mode, Operating mode and Null Data Write mode.

- **DMAxREQ: DMA Channel x IRQ Select Register**

  This register associates the DMA channel with a specific DMA capable peripheral by assigning the peripheral IRQ to the DMA channel.

- **DMAxSTA: DMA Channel x DPSRAM Start Address Offset Register A**

  This register specifies the primary start address offset from the DMA DPSRAM base address of the data block to be transferred by DMA channel x to or from the DPSRAM. Reads of this register return the value of the latest DPSRAM transfer address offset. Writes to this register while the channel x is enabled (i.e., active) may result in unpredictable behavior and should be avoided.

- **DMAxSTB: DMA Channel x DPSRAM Start Address Offset Register B**

  This register specifies the secondary start address offset from the DMA DPSRAM base address of the data block to be transferred by DMA channel x to or from the DPSRAM. Reads of this register return the value of the latest DPSRAM transfer address offset. Writes to this register while the channel x is enabled (i.e., active) may result in unpredictable behavior and should be avoided.

- **DMAxPAD: DMA Channel x Peripheral Address Register**

  This read/write register contains the static address of the peripheral data register. Writes to this register while the corresponding DMA channel is enabled (i.e., active) may result in unpredictable behavior and should be avoided.

- **DMAxCNT: DMA Channel x Transfer Count Register**

  This register contains the transfer count. DMAxCNT + 1 represents the number of DMA requests the channel must service before the data block transfer is considered complete. That is, a DMAxCNT value of '0' will transfer one element. The value of the DMAxCNT register is independent of the transfer data size (SIZE bit in the DMAxCON register). Writes to this register while the corresponding DMA channel is enabled (i.e., active) may result in unpredictable behavior and should be avoided.

In addition to the individual DMA channel registers, the DMA Controller has three DMA status registers.

- **DSADR: Most Recent DMA DPSRAM Address Register**

  This 16-bit, read-only, status register is common to all DMA channels. It captures the address of the most recent DPSRAM access (read or write). It is cleared at Reset and, therefore, contains the value '0x0000' if read prior to any DMA activity. This register is accessible at any time but is primarily intended as a debug aid.

- **DMACS0: DMA Controller Status Register 0**

  This 16-bit, read-only status register contains the DPSRAM and Peripheral Write Collision flags, XWCOLx and PWCOLx, respectively. See **22.10 "Data Write Collisions"** for more detailed information.

- **DMACS1: DMA Controller Status Register 1**

  This 16-bit, read-only status register indicates which DMA channel was most recently active and provides the Ping-Pong mode status of each DMA channel by indicating which DPSRAM Start Address Offset register is selected (DMAxSTA or DMAxSTB).

Register 22-1: DMAxCON: DMA Channel x Control Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	U-0	U-0
CHEN	SIZE	DIR	HALF	NULLW	—	—	—
bit 15							bit 8

U-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0
—	—	AMODE<1:0>		—	—	MODE<1:0>	
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **CHEN:** Channel Enable bit
1 = Channel enabled
0 = Channel disabled

bit 14 **SIZE:** Data Transfer Size bit
1 = Byte
0 = Word

bit 13 **DIR**: Transfer Direction bit (source/destination bus select)
1 = Read from DPSRAM address, write to peripheral address
0 = Read from Peripheral address, write to DPSRAM address

bit 12 **HALF:** Block Transfer Interrupt Select bit
1 = Initiate interrupt when half of the data has been moved
0 = Initiate interrupt when all of the data has been moved

bit 11 **NULLW:** Null Data Peripheral Write Mode Select bit
1 = Null data write to peripheral in addition to DPSRAM write (DIR bit must also be clear)
0 = Normal operation

bit 10-6 **Unimplemented:** Read as '0'

bit 5-4 **AMODE<1:0>:** DMA Channel Addressing Mode Select bits
11 = Reserved
10 = Peripheral Indirect Addressing mode
01 = Register Indirect without Post-Increment mode
00 = Register Indirect with Post-Increment mode

bit 3-2 **Unimplemented:** Read as '0'

bit 1-0 **MODE<1:0>:** DMA Channel Operating Mode Select bits
11 = One-Shot, Ping-Pong modes enabled (one block transfer from/to each DMA RAM buffer)
10 = Continuous, Ping-Pong modes enabled
01 = One-Shot, Ping-Pong modes disabled
00 = Continuous, Ping-Pong modes disabled

Register 22-2: DMAxREQ: DMA Channel x IRQ Select Register

R/S-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
FORCE(1)	—	—	—	—	—	—	—
bit 15							bit 8

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	IRQSEL<6:0>						
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **FORCE:** Force DMA Transfer bit(1)
1 = Force a single DMA transfer (manual mode)
0 = Automatic DMA transfer initiation by DMA Request

bit 14-7 **Unimplemented:** Read as '0'

bit 6-0 **IRQSEL<6:0>:** DMA Peripheral IRQ Number Select bits
0000000 = INT0 – External Interrupt 0
0000001 = IC1 – Input Capture 1
0000010 = OC1 – Output Compare 1
0000101 = IC2 – Input Capture 2
0000110 = OC2 – Output Compare 2
0000111 = TMR2 – Timer2
0001000 = TMR3 – Timer3
0001010 = SPI1 – Transfer Done
0001011 = UART1RX – UART1 Receiver
0001100 = UART1TX – UART1 Transmitter
0001101 = ADC1 – ADC1 Convert done
0010101 = ADC2 – ADC2 Convert Done
0011110 = UART2RX – UART2 Receiver
0011111 = UART2TX – UART2 Transmitter
0100001 = SPI2 Transfer Done
0100010 = ECAN1 – RX Data Ready
0110111 = ECAN2 – RX Data Ready
0111100 = DCI – CODEC Transfer Done
1000110 = ECAN1 – TX Data Request
1000111 = ECAN2 – TX Data Request

Note 1: The FORCE bit cannot be cleared by the user. The FORCE bit is cleared by hardware when the forced DMA transfer is complete.

Register 22-3: DMAxSTA: DMA Channel x DPSRAM Start Address Offset Register A

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
STA<15:8>							
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
STA<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **STA<15:0>:** Primary DPSRAM Start Address Offset bits (source or destination)

Register 22-4: DMAxSTB: DMA Channel x DPSRAM Start Address Offset Register B

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
STB<15:8>							
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
STB<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **STB<15:0>:** Secondary DPSRAM Start Address Offset bits (source or destination)

Register 22-5: DMAxPAD: DMA Channel x Peripheral Address Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PAD<15:8>							
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PAD<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **PAD<15:0>:** Peripheral Address Register bits

Register 22-6: DMAxCNT: DMA Channel x Transfer Count Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	—	—	—	CNT<9:8>	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
CNT<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-10 **Reserved**

bit 9-0 **CNT<9:0>:** DMA Transfer Count Register bits

Register 22-7: DSADR: Most Recent DMA DPSRAM Address Register

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
DSADR<15:8>							
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
DSADR<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **DSADR<15:0>:** Most Recent DMA DPSRAM Address Accessed by DMA bits

Register 22-8: DMACS0: DMA Controller Status Register 0

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
PWCOL7	PWCOL6	PWCOL5	PWCOL4	PWCOL3	PWCOL2	PWCOL1	PWCOL0
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
XWCOL7	XWCOL6	XWCOL5	XWCOL4	XWCOL3	XWCOL2	XWCOL1	XWCOL0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **PWCOL7:** Channel 7 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 14 **PWCOL6:** Channel 6 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 13 **PWCOL5:** Channel 5 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 12 **PWCOL4:** Channel 4 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 11 **PWCOL3:** Channel 3 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 10 **PWCOL2:** Channel 2 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 9 **PWCOL1:** Channel 1 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 8 **PWCOL0:** Channel 0 Peripheral Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 7 **XWCOL7:** Channel 7 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 6 **XWCOL6:** Channel 6 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 5 **XWCOL5:** Channel 5 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 4 **XWCOL4:** Channel 4 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 3 **XWCOL3:** Channel 3 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

Register 22-8: DMACS0: DMA Controller Status Register 0 (Continued)

bit 2 **XWCOL2:** Channel 2 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 1 **XWCOL1:** Channel 1 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

bit 0 **XWCOL0:** Channel 0 DPSRAM Write Collision Flag bit
1 = Write collision detected
0 = No write collision detected

Register 22-9: DMACS1: DMA Controller Status Register 1

U-0	U-0	U-0	U-0	R-1	R-1	R-1	R-1
—	—	—	—	LSTCH<3:0>			
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
PPST7	PPST6	PPST5	PPST4	PPST3	PPST2	PPST1	PPST0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15-12 **Unimplemented:** Read as '0'

bit 11-8 **LSTCH<3:0>:** Last DMAC Channel Active bits
1111 = No DMA transfer has occurred since system Reset
1110-1000 = Reserved
0111 = Last data transfer was by Channel 7
0110 = Last data transfer was by Channel 6
0101 = Last data transfer was by Channel 5
0100 = Last data transfer was by Channel 4
0011 = Last data transfer was by Channel 3
0010 = Last data transfer was by Channel 2
0001 = Last data transfer was by Channel 1
0000 = Last data transfer was by Channel 0
Set to '1111' at Reset. This field is accessible at any time but is primarily intended as a debugging aid.

bit 7 **PPST7:** Channel 7 'Ping-Pong' Mode Status Flag
1 = DMA7STB register selected
0 = DMA7STA register selected

bit 6 **PPST6:** Channel 6 'Ping-Pong' Mode Status Flag
1 = DMA6STB register selected
0 = DMA6STA register selected

bit 5 **PPST5:** Channel 5 'Ping-Pong' Mode Status Flag
1 = DMA5STB register selected
0 = DMA5STA register selected

bit 4 **PPST4:** Channel 4 'Ping-Pong' Mode Status Flag
1 = DMA4STB register selected
0 = DMA4STA register selected

bit 3 **PPST3:** Channel 3 'Ping-Pong' Mode Status Flag
1 = DMA3STB register selected
0 = DMA3STA register selected

bit 2 **PPST2:** Channel 2 'Ping-Pong' Mode Status Flag
1 = DMA2STB register selected
0 = DMA2STA register selected

bit 1 **PPST1:** Channel 1 'Ping-Pong' Mode Status Flag
1 = DMA1STB register selected
0 = DMA1STA register selected

bit 0 **PPST0:** Channel 0 'Ping-Pong' Mode Status Flag
1 = DMA0STB register selected
0 = DMA0STA register selected

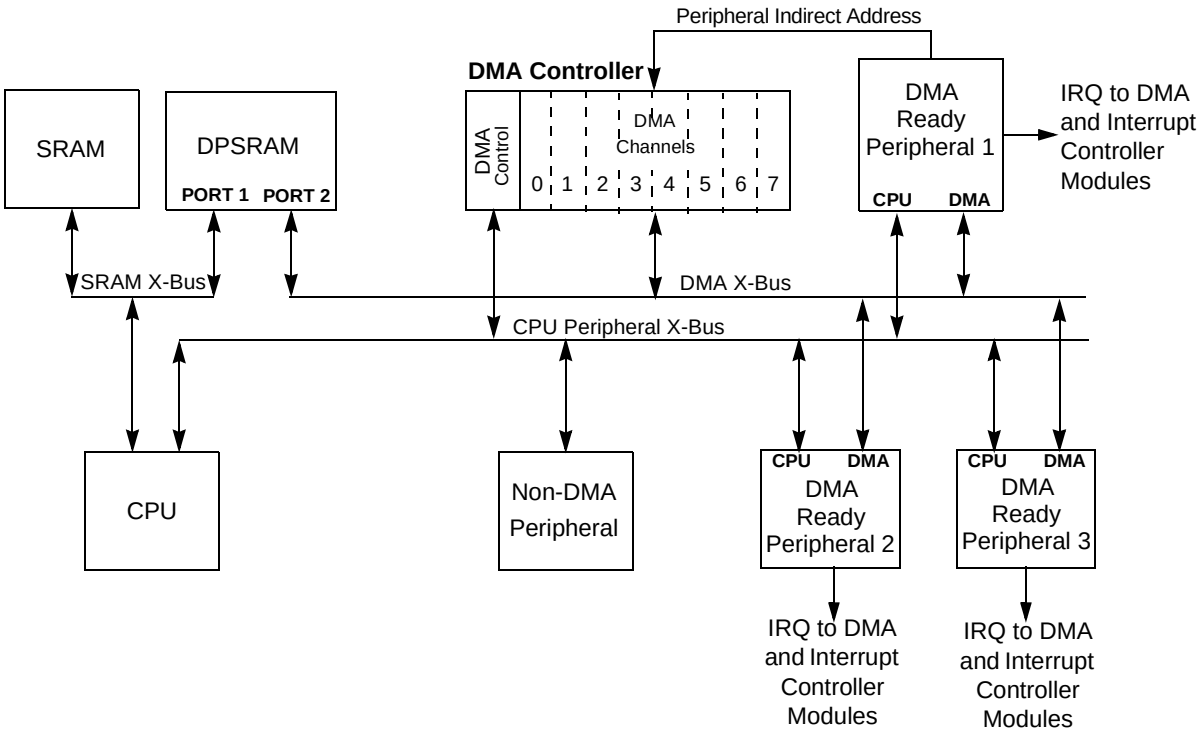
Note: This register is read-only.

22.3 DMA BLOCK DIAGRAM

Figure 22-2 is a block diagram that shows how the DMA integrates into the dsPIC33F internal architecture. The CPU communicates with conventional SRAM across the X-bus. It also communicates with Port 1 of the Dual Port SRAM (DPSRAM) block across the same X-bus. The CPU communicates with the peripherals across a separate Peripheral X-bus, which also resides within X data space.

The DMA channels communicate with Port 2 of the DPSRAM and the DMA port of each of the DMA-ready peripherals across a dedicated DMA bus.

Figure 22-2: DMA Controller Block Diagram



Note: CPU and DMA address buses are not shown for clarity.

Unlike other architectures, the dsPIC33F CPU is capable of a read and a write access within each CPU bus cycle. Similarly, the DMA can complete the transfer of a byte or word every bus cycle across its dedicated bus. This also ensures that all DMA transfers are not interrupted. That is, once the transfer has started, it will complete within the same cycle, irrespective of other channel activity.

The user application can designate any DMA-ready peripheral interrupt to be a DMA request, the term given to an IRQ when it is directed to the DMA. It is assumed, of course, that when a DMA channel is configured to respond to a particular interrupt as a DMA request, the corresponding CPU interrupt is disabled, otherwise a CPU interrupt will also be requested.

Each DMA channel can also be triggered manually through software. Setting the FORCE bit in the DMAxCON register initiates a manual DMA request that is subject to the same arbitration as all interrupt-based DMA requests (see **22.8 “DMA Channel Arbitration and Overruns”**).

22.4 DMA DATA TRANSFER

Figure 22-3 illustrates a data transfer between a peripheral and Dual Port SRAM.

A. In this example, DMA Channel 5 is configured to operate with DMA-Ready Peripheral 1.

B. When data is ready to be transferred from the peripheral, a DMA Request is issued by the peripheral. The DMA request is arbitrated with any other coincident requests. If this channel has the highest priority, the transfer is completed during the next cycle. Otherwise, the DMA request remains pending until it becomes the highest priority.

C. The DMA Channel executes a data read from the designated peripheral address, which is user-application defined within the active channel.

D. The DMA Channel writes the data to the designated DPSRAM address.

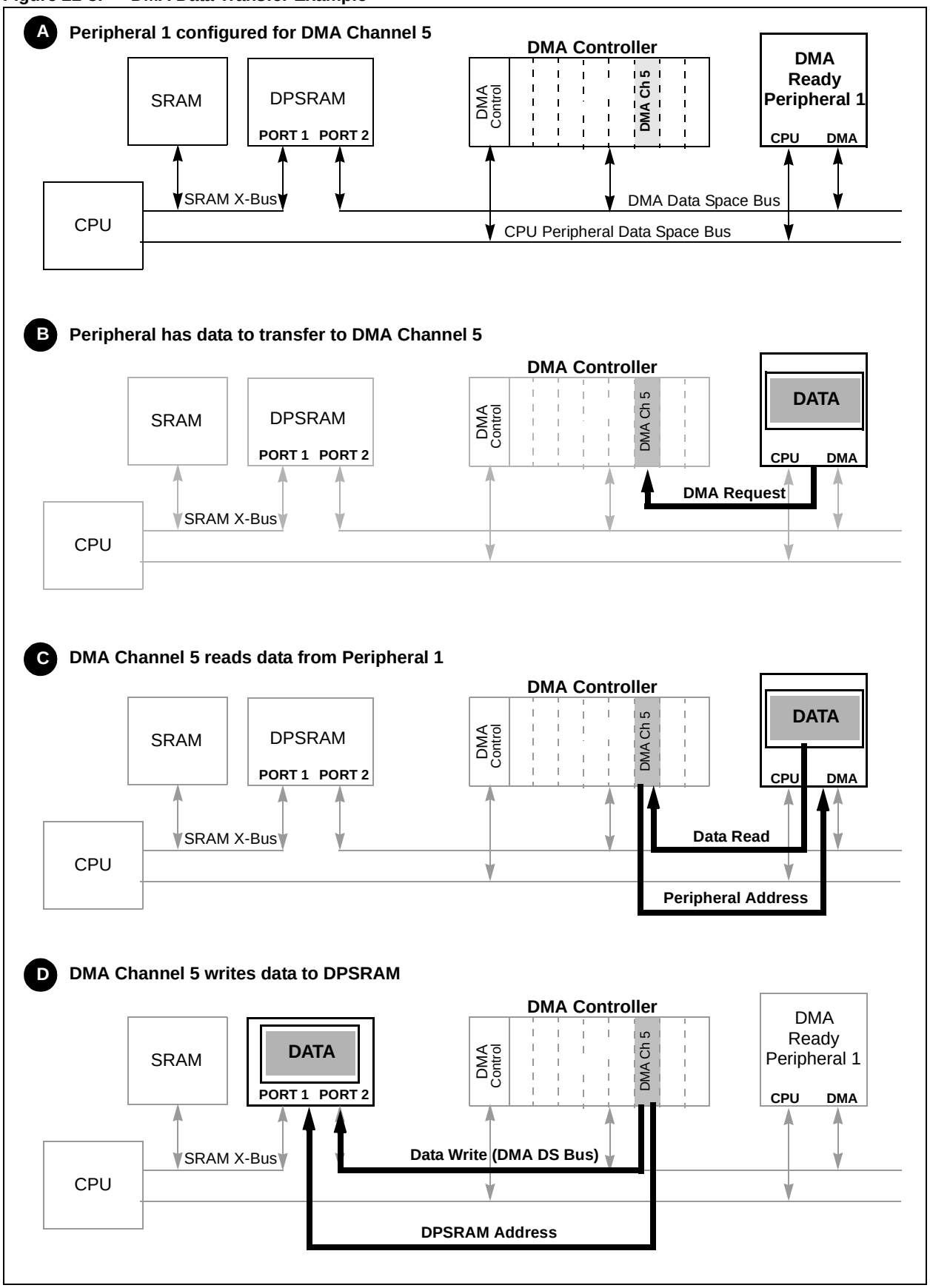
This example represents Register Indirect Mode, where the DPSRAM address is designated within the DMA Channel via the DMA Status registers (DMAxSTA or DMAxSTB). In Peripheral Indirect Mode, the DPSRAM address is derived from the peripheral, not the active channel. More information on this topic is presented in **22.6.6 “Peripheral Indirect Addressing Mode”**.

The entire DMA read and write transfer operation is accomplished uninterrupted in a single instruction cycle. During this entire process, DMA request remains latched in the DMA channel until the data transfer is complete.

The DMA channel concurrently monitors the Transfer Counter register (DMA5CNT). When the transfer count reaches a user-application specified limit, data transfer is considered complete and a CPU interrupt is asserted to alert the CPU to process the newly received data.

During the data transfer cycle, the DMA controller also continues to arbitrate pending or subsequent DMA requests to maximize throughput.

Figure 22-3: DMA Data Transfer Example



22.5 DMA SET UP

For DMA data transfer to function properly, the DMA channels and peripherals must be appropriately configured:

- DMA channels must be associated with peripherals (see **22.5.1 "DMA Channel to Peripheral Association Setup"**)
- Peripherals must be properly configured (see **22.5.2 "Peripheral Configuration Setup"**)
- DPSRAM data start addresses must be initialized (see **22.5.3 "Memory Address Initialization"**)
- Initializing DMA transfer count must be initialized (see **22.5.4 "DMA Transfer Count Set Up"**)
- Appropriate addressing and operating modes must be selected (see **22.6 "DMA Operating Modes"**)

22.5.1 DMA Channel to Peripheral Association Setup

The DMA Channel needs to know which peripheral target address to read from or write to, and when to do so. This information is configured in the DMA Channel x Peripheral Address Register (DMAxPAD) and DMA Channel x IRQ Select Register (DMAxREQ), respectively.

Table 22-1 shows which values should be written to these registers to associate a particular peripheral with a given DMA channel.

Table 22-1: DMA Channel to Peripheral Associations

Peripheral to DMA Association	DMAxREQ Register IRQSEL<6:0> Bits	DMAxPAD Register Values to Read From Peripheral	DMAxPAD Register Values to Write to Peripheral
INT0 – External Interrupt 0	0000000	—	—
IC1 – Input Capture 1	0000001	0x0140 (IC1BUF)	—
IC2 – Input Capture 2	0000101	0x0144 (IC2BUF)	—
OC1 – Output Compare 1 Data	0000010	—	0x0182 (OC1R)
OC1 – Output Compare 1 Secondary Data	0000010	—	0x0180 (OC1RS)
OC2 – Output Compare 2 Data	0000110	—	0x0188 (OC2R)
OC2 – Output Compare 2 Secondary Data	0000110	—	0x0186 (OC2RS)
TMR2 – Timer2	0000111	—	—
TMR3 – Timer3	0001000	—	—
SPI1 – Transfer Done	0001010	0x0248 (SPI1BUF)	0x0248 (SPI1BUF)
SPI2 – Transfer Done	0100001	0x0268 (SPI2BUF)	0x0268 (SPI2BUF)
UART1RX – UART1 Receiver	0001011	0x0226 (U1RXREG)	—
UART1TX – UART1 Transmitter	0001100	—	0x0224 (U1TXREG)
UART2RX – UART2 Receiver	0011110	0x0236 (U2RXREG)	—
UART2TX – UART2 Transmitter	0011111	—	0x0234 (U2TXREG)
ECAN1 – RX Data Ready	0100010	0x0440 (C1RXD)	—
ECAN1 – TX Data Request	1000110	—	0x0442 (C1TXD)
ECAN2 – RX Data Ready	0110111	0x0540 (C2RXD)	—
ECAN2 – TX Data Request	1000111	—	0x0542 (C2TXD)
DCI – CODEC Transfer Done	0111100	0x0290 (RXBUF0)	0x0298 (TXBUF0)
ADC1 – ADC1 Convert Done	0001101	0x0300 (ADC1BUF0)	—
ADC2 – ADC2 Convert Done	0010101	0x0340 (ADC2BUF0)	—

If two DMA channels select the same peripheral as the source of their DMA request, both channels receive the DMA request simultaneously. However, the highest priority channel executes its transfer first, leaving the other channel pending. This situation is common where a single DMA request is used to move data both from and to a peripheral (For example, SPI). Two DMA channels are used. One is allocated for peripheral reads, and the other is allocated for peripheral data writes. Both use the same DMA request.

If the DMAxPAD register is initialized to a value not listed in the Table 22-1, DMA channel writes to this peripheral address will be ignored. DMA channel reads from this address will result in a read of '0'.

22.5.2 Peripheral Configuration Setup

The second step in the DMA set-up process is to properly configure DMA-ready peripherals for DMA operation. Table 22-2 outlines the configuration requirements for DMA-ready peripherals.

Table 22-2: Configuration Considerations for DMA-Ready Peripherals

DMA-Ready Peripheral	Configuration Considerations
ECAN	ECAN buffers are allocated in the DMA RAM. The overall size of the CAN buffer area and FIFO in the DMA RAM is specified by the user and must be defined via the DMA Buffer Size bits DMABS<2:0> in the ECAN FIFO Control (C1FCTRL) register. Sample code is shown in Example 22-9.
Data Converter Interface (DCI)	The DCI must be configured to generate an interrupt for every buffered data word by setting Buffer Length Control bits (BLEN<1:0>) to '00' in the DCI Control 2 (DCICON2) register. The same DCI interrupt must be used as the request for two DMA channels to support RX and TX data transfers. If the DCI module is operating as Master and only receiving data, the second DMA channel must be used to send dummy transmit data. Sample code is shown in Example 22-11.
10-bit/12-bit Analog-to-Digital Converter (ADC)	When the ADC is used with the DMA in Peripheral Indirect mode, the Increment Rate for the DMA Addresses bits (SMPI<3:0>) in the ADCx Control 2 (ADCxCON2) register, and the number of DMA Buffer Locations per Analog Input bits (DMABL<2:0>) in the ADCx Control 4 (ADCxCON4) register must be set properly. Also, the DMA Buffer Build mode bit (ADDMABM) in the ADCx Control 1 (ADxCON1) register must be properly set for ADC address generation. See **22.6.6.1 "ADC Support for DMA Address Generation"** for detailed information. Sample code is shown in Example 22-5 and Example 22-7.
Serial Peripheral Interface (SPI)	If the SPI module is operating as master and only receiving data, the second DMA channel must be allocated and used to send dummy transmit data. Alternatively, a single DMA channel can be used in Null Data Write mode. See **22.6.11 "Null Data Write Mode"** for detailed information. Sample code is shown in Example 22-12.
UART	The UART must be configured to generate interrupts for every character received or transmitted. For the UART receiver to generate an RX interrupt for each character received, Receive Interrupt Mode Selection bits (URXISEL<1:0>) must be set to '00' or '01' in the Status and Control register (UxSTA). For the UART transmitter to generate a TX interrupt for each character transmitted, Transmission Interrupt Mode Selection bits UTXISEL0 and UTXISEL1 must be set to '0' in the Status and Control (UxSTA) register. Sample code is shown in Example 22-10.
Input Capture	The Input Capture module must be configured to generate an interrupt for each capture event by setting Number of Captures per Interrupt bits (ICI<1:0>) to '00' in Input Capture Control (ICxCON) register. Sample code is shown in Example 22-4.
Output Compare	The Output Compare module requires no special configuration to work with DMA. Typically, however, the Timer is used to provide the DMA request, and it needs to be properly configured. Sample code is shown in Example 22-3.

Table 22-2: Configuration Considerations for DMA-Ready Peripherals (Continued)

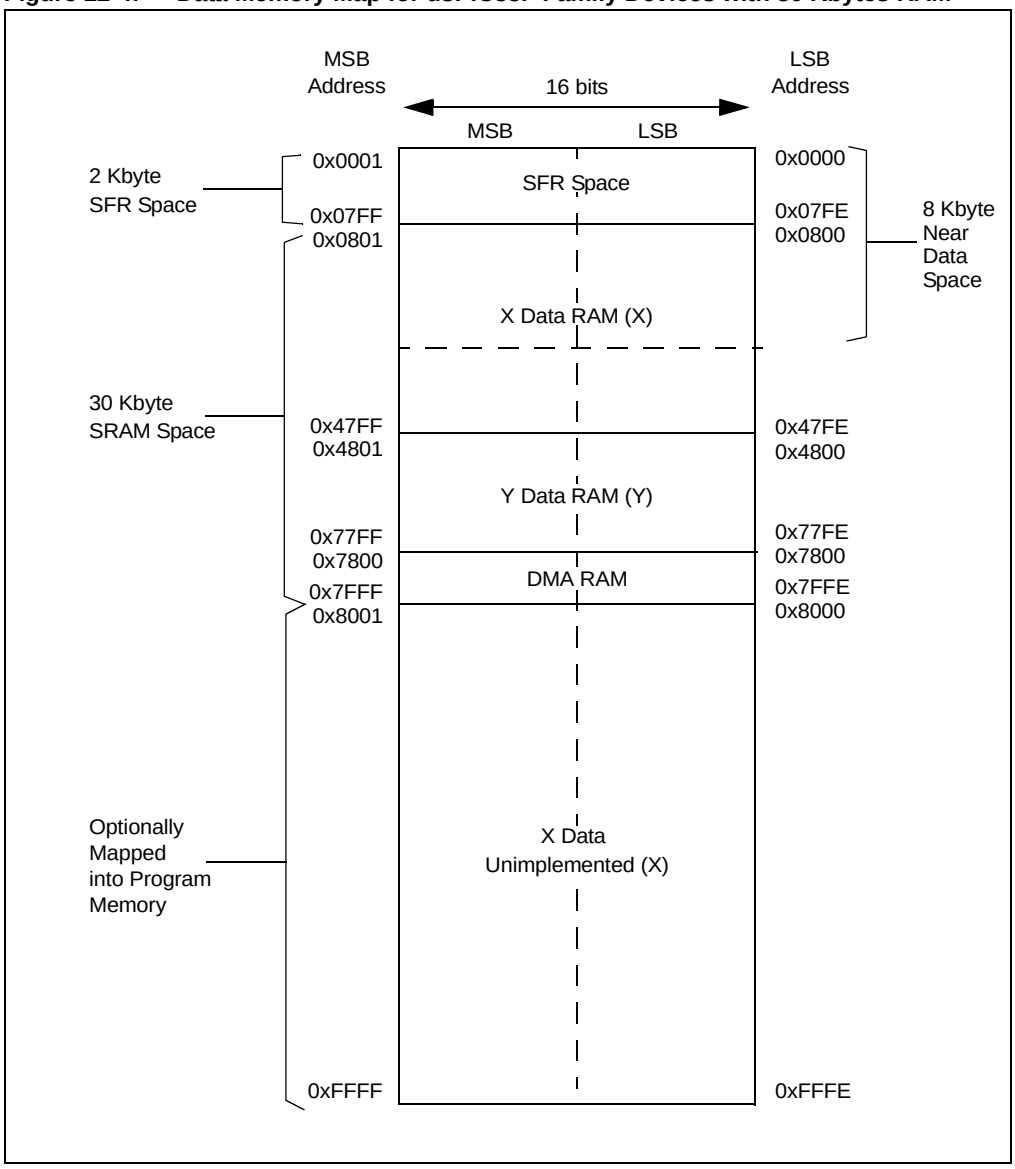
DMA-Ready Peripheral	Configuration Considerations
External Interrupt and Timers	Only External Interrupt 0 and Timers 2 and 3 can be selected for DMA request. Although these peripherals do not support DMA transfer themselves, they can be used to trigger DMA transfers for other DMA supported peripherals. For example, Timer2 can trigger DMA transactions for the Output Compare peripheral in PWM mode. Sample code is shown in Example 22-3.

An error condition within a DMA-enabled peripheral generally sets a status flag and generates an interrupt (if interrupts are enabled by the user application). When a peripheral is serviced by the CPU, the data interrupt handler is required to check for error flags and, if necessary, take appropriate action. However, when a peripheral is serviced by the DMA channel, the DMA can only respond to data transfer requests and it is not aware of any subsequent error conditions. All error conditions in DMA compatible peripherals, must have an associated interrupt enabled and be serviced by the user-defined Interrupt Service Routine (ISR), if such an interrupt is present in the peripheral.

22.5.3 Memory Address Initialization

The third DMA setup requirement is to allocate memory buffers within a specific memory area for DMA access. The location and size of this memory area depends on the dsPIC33F device (refer to the device data sheet for specific information). Figure 22-4 shows a DMA memory area of 2 Kbytes for dsPIC33F devices with 30 Kbytes of RAM.

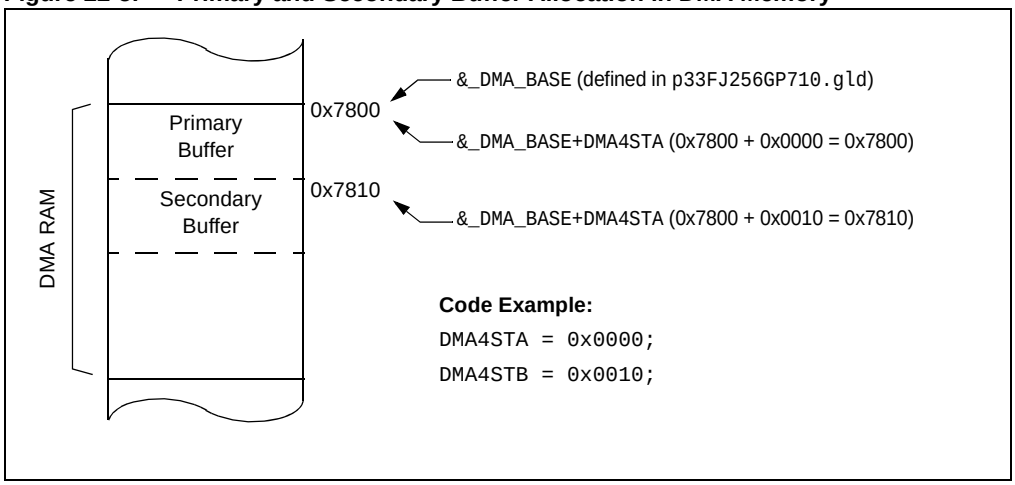
Figure 22-4: Data Memory Map for dsPIC33F Family Devices with 30 Kbytes RAM



To operate properly, the DMA needs to know the DPSRAM address to read from or write to as an offset from the beginning of the DMA memory. This information is configured in the DMA Channel x DPSRAM Start Address Offset A (DMAxSTA) register and DMA Channel x DPSRAM Start Address Offset B (DMAxSTB) register.

Figure 22-5 is an example that shows how the primary and secondary DMA Channel 4 buffers are set up on the dsPIC33FJ256GP710 device at address 0x7800 and 0x7810, respectively.

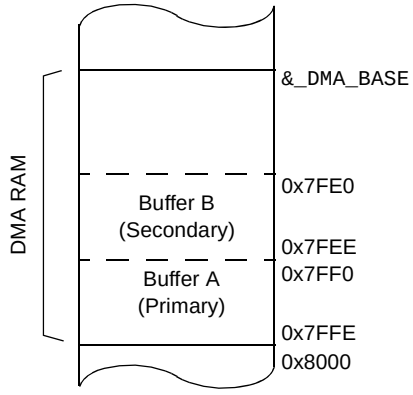
Figure 22-5: Primary and Secondary Buffer Allocation in DMA Memory



In this example, you must be familiar with the memory layout for the device in order to hard code this information into the application. Also, you must use pointer arithmetic to access these buffers after the DMA transfer is complete. As a result, this implementation is difficult to port from one processor to another.

The MPLAB® C30 compiler simplifies DMA buffer initialization and access by providing built-in C language primitives for that purpose. For example, the code in Figure 22-6 allocates two buffers in the DMA memory and initializes the DMA channel to point to them.

Figure 22-6: Primary and Secondary DMA Buffer Allocation with MPLAB® IDE



Code Example:

```
unsigned int BufferA[8] __attribute__((space(dma)));
unsigned int BufferB[8] __attribute__((space(dma)));

DMA0STA = __builtin_dmaoffset(BufferA);
DMA0STB = __builtin_dmaoffset(BufferB);
```

Note:	MPLAB LINK30 linker allocates the primary and secondary buffers in reverse order starting at the bottom of the DMA memory space.

If the DMAxSTA (and/or DMAxSTB) register is initialized to a value that will result in the DMA channel reading or writing RAM addresses outside of DMA RAM space, DMA channel writes to this memory address are ignored. DMA channel reads from this memory address result in a read of '0'.

22.5.4 DMA Transfer Count Set Up

In the fourth step of the DMA set-up process, each DMA channel must be programmed to service N + 1 number of requests before the data block transfer is considered complete. The value 'N' is specified by programming DMA Channel x Transfer Count (DMAxCNT) register. That is, a DMAxCNT value of '0' will transfer one element.

The value of the DMAxCNT register is independent of the transfer data size (byte or word), which is specified in the SIZE bit in the DMAxCON register.

If the DMAxCNT register is initialized to a value that will result in the DMA channel reading or writing RAM addresses outside of DMA RAM space, DMA channel writes to this memory address are ignored. DMA channel reads from this memory address result in a read of '0'.

22.5.5 Operating Mode Set Up

The fifth and final DMA set up step is to specify the mode of operation for each DMA channel by configuring the DMA Channel x Control (DMAxCON) register. See **22.6 "DMA Operating Modes"** for specific setup information.

22.6 DMA OPERATING MODES

The DMA channel supports these modes of operation:

- Word or Byte data transfer
- Transfer direction (peripheral to DPSRAM, or DPSRAM to peripheral)
- Full or half transfer interrupts to CPU
- Post-Increment or static DPSRAM addressing
- Peripheral Indirect Addressing
- One-Shot or continuous block transfers
- Auto-switch between two start addresses offsets (DMAxSTA or DMAxSTB) after each transfer complete (Ping-Pong mode)
- Null Data Write mode

Additionally, DMA supports a manual mode, which forces a single DMA transfer.

22.6.1 Word or Byte Data Transfer

Each DMA channel can be configured to transfer data by word or byte. Word data can only be moved to and from aligned (even) addresses. Byte data, on the other hand, can be moved to or from any (legal) address.

If the SIZE bit (DMAxCON<14>) is clear, word-sized data is transferred. If Register Indirect With Post-Increment Addressing mode is enabled, the address is post-incremented by 2 after every word transfer (see **22.6.5 “Register Indirect Without Post-increment Addressing Mode”**).

If the SIZE bit is set, byte-sized data is transferred. If Register Indirect With Post-Increment Addressing mode is enabled, the address is incremented by 1 after every byte transfer.

22.6.2 Transfer Direction

Each DMA channel can be configured to transfer data from a peripheral to the DPSRAM or from the DPSRAM to a peripheral.

If the Transfer Direction (DIR) bit in DMAxCON is clear, data is read from the peripheral (using the Peripheral Address as provided by DMAxPAD) and the destination write is directed to the DPSRAM DMA memory address offset (using DMAxSTA or DMAxSTB).

If the DIR bit is set, data is read from the DPSRAM DMA memory address offset (using DMAxSTA or DMAxSTB) and the destination write is directed to the peripheral (using the peripheral address, as provided by DMAxPAD).

Once configured, each channel is a unidirectional data conduit. That is, should a peripheral require read and write data using the DMA module, two channels must be assigned – one for read and one for write.

22.6.3 Full or Half-Block Transfer Interrupts

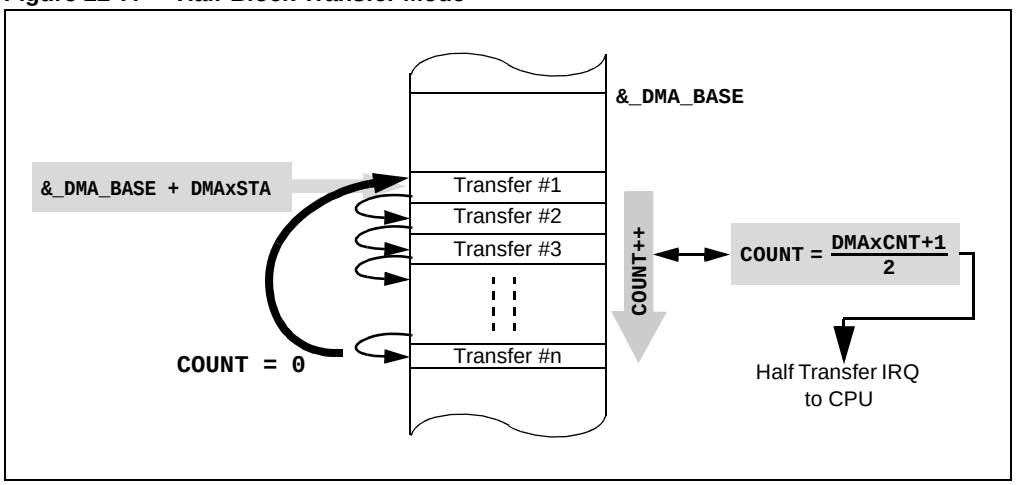
Each DMA channel provides an interrupt to the interrupt controller when block data transfer is complete or half complete. This mode is designated by clearing or setting the HALF bit in the DMA Channel x Control (DMAxCON) register:

HALF = 0 (initiate interrupt when all of the data has been moved)

HALF = 1 (initiate interrupt when half of the data has been moved)

When DMA Continuous mode is used, the CPU must be able to process the incoming or outgoing data at least as fast as the DMA is moving it. The half transfer interrupt helps mitigate this problem by generating an interrupt when only half of the data has been transferred. For example, if an ADC is being continuously read by the DMA controller, the half transfer interrupt allows the CPU to process the buffer before it becomes completely full. Provided it never gets ahead of the DMA writes, this scheme can be used to relax the CPU response time requirements. Figure 22-7 illustrates this process.

Figure 22-7: Half-Block Transfer Mode



In all modes, when the HALF bit is set, the DMA issues an interrupt only when the first half of Buffer A and/or B is transferred. No interrupt is issued when Buffer A and/or B is completely transferred. In other words, interrupts are only issued when DMA completes (DMAxCNT + 1)/2 transfers. If (DMAxCNT + 1) is equal to an odd number, interrupts are issued after (DMAxCNT + 2)/2 transfers.

For example, if DMA3 is configured for One-Shot, Ping-Pong buffers (MODE<1:0> = 11), and DMA3CNT = 7, two DMA3 interrupts are issued – one after transferring four elements from Buffer A, and one after transferring four elements from Buffer B. (For more information see **22.6.7 "One-Shot Mode"** and **22.6.9 "Ping-Pong Mode"**.)

Even though the DMA channel issues an interrupt on either half- **or** full-block transfers, the user application can "trick" the DMA channel into issuing an interrupt on half- **and** full-block transfers by toggling the value of the HALF bit during each DMA interrupt. For example, if the DMA channel is set up with the HALF bit set to '1', an interrupt is issued after each half-block transfer. If the user application resets the HALF bit to '0' while the interrupt is being serviced, the DMA issues another interrupt when the full-block transfer is complete.

To enable these interrupts, the corresponding DMA Interrupt Enable bit (DMAxIE) must be set in the Interrupt Enable Control (IECx) register in the interrupt controller module, as shown in Table 22-3.

Table 22-3: Interrupt Controller Settings for Enabling/Disabling DMA Interrupts

DMA Channel	Interrupt Controller Register Name and Bit Number	Corresponding Register Bit Name	C Structure Access Code
0	IEC0<4>	DMA0IE	IEC0bits.DMA0IE
1	IEC0<14>	DMA1IE	IEC1bits.DMA1IE
2	IEC1<8>	DMA2IE	IEC1bits.DMA2IE
3	IEC2<4>	DMA3IE	IEC2bits.DMA3IE
4	IEC2<14>	DMA4IE	IEC2bits.DMA4IE
5	IEC3<13>	DMA5IE	IEC3bits.DMA5IE
6	IEC4<4>	DMA6IE	IEC4bits.DMA6IE
7	IEC4<5>	DAM7IE	IEC4bits.DMA7IE

Example 22-1shows how DMA channel 0 interrupt is enabled:

Example 22-1: Code to Enable DMA Channel 0 Interrupt

```
IEC0bits.DMA0IE = 1;
```

Each DMA channel transfer interrupt sets a corresponding status flag in the interrupt controller, which triggers the Interrupt Service Routine (ISR). The user application must then clear that status flag to prevent the transfer-complete ISR from re-executing.

Table 22-4 shows the Interrupt Flag Status (IFSx) register and corresponding bit name (DMAxIF) in the interrupt controller module. It also shows the C Structure Access Code that clears the flag.

Table 22-4: Interrupt Controller Settings for Clearing DMA Interrupt Status Flags

DMA Channel	Interrupt Controller Register Name and Bit Number	Corresponding Register Bit Name	C Structure Access Code
0	IFS0<4>	DMA0IF	IFS0bits.DMA0IE
1	IFS0<14>	DMA1IF	IFS0bits.DMA1IE
2	IFS1<8>	DMA2IF	IFS1bits.DMA2IE
3	IFS2<4>	DMA3IF	IFS2bits.DMA3IE
4	IFS2<14>	DMA4IF	IFS2bits.DMA4IE
5	IFS3<13>	DMA5IF	IFS3bits.DMA5IE
6	IFS4<4>	DMA6IF	IFS4bits.DMA6IE
7	IFS4<5>	DMA7IF	IFS4bits.DMA7IE

As an example, assume DMA channel 0 interrupt is enabled, DMA channel 0 transfer has finished and the associated interrupt has been issued to the Interrupt controller. The following code must be present in the DMA channel 0 ISR to clear the status flag and prevent a pending interrupt.

Example 22-2: Code to Clear DMA Channel 0 Interrupt

```
void __attribute__((__interrupt__)) _DMA0Interrupt(void)
{
    . . .

IFS0bits.DMA0IF = 0;
}
```

22.6.4 Register Indirect With Post-Increment Addressing Mode

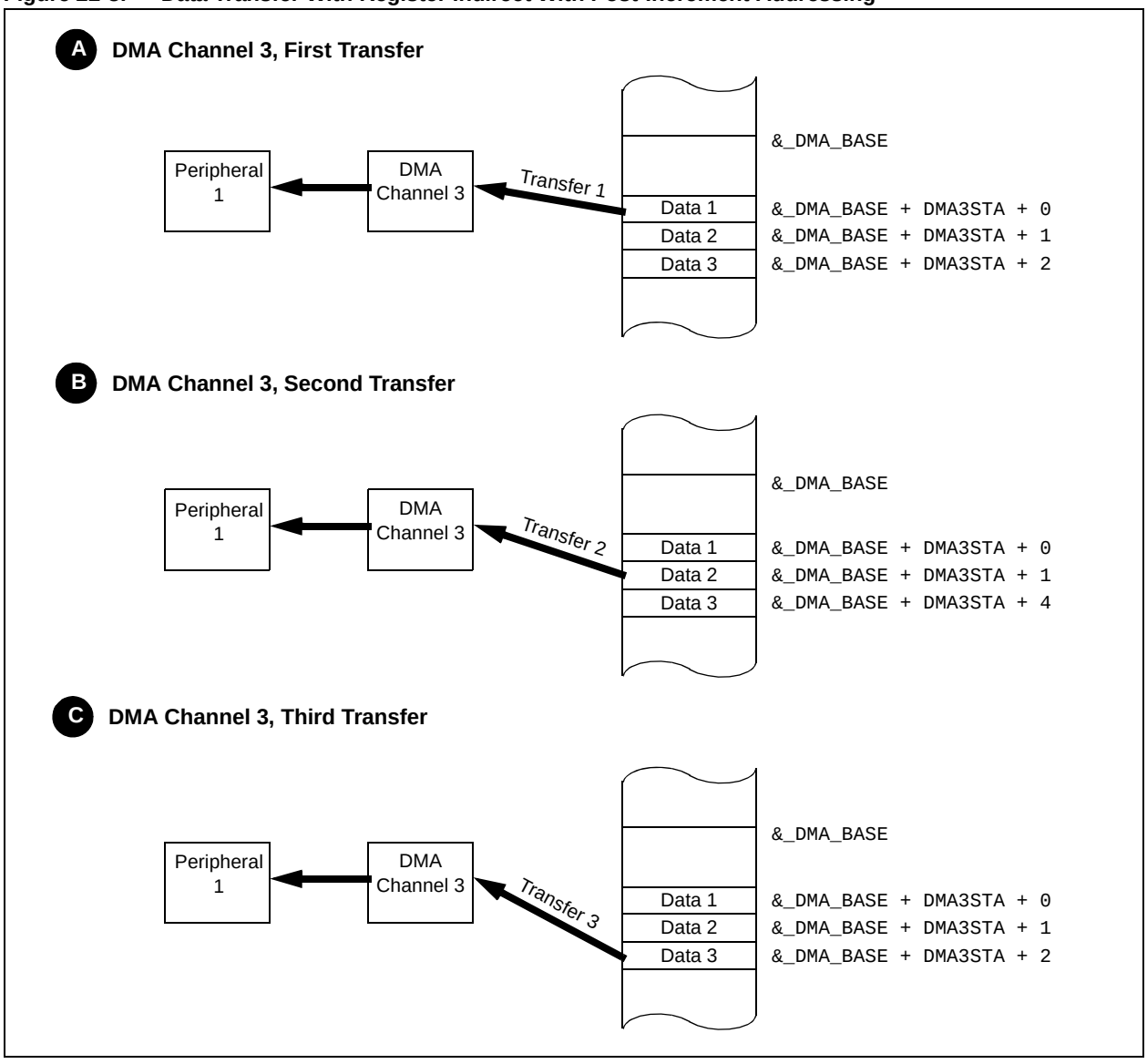
Register Indirect With Post-Increment Addressing is used to move blocks of data by incrementing the DPSRAM address after each transfer.

The DMA channel defaults to this mode after the DMA controller is reset. This mode is selected by programming the Addressing Mode Select bits (AMODE<1:0>) to '00' in the DMA Channel Control (DMAxCON) register. In this mode, the DPSRAM Start Address Offset (DMAxSTA or DMAxSTB) register provides the starting address of DPSRAM buffer.

The user application determines the latest DPSRAM transfer address offset by reading the DPSRAM Start Address Offset register. However, the contents of this register are not modified by the DMA controller.

Figure 22-8 illustrates data transfer in this mode.

Figure 22-8: Data Transfer With Register Indirect With Post-Increment Addressing



Example 22-3: Code for Output Compare and DMA with Register Indirect Post-Increment Mode

Set up Output Compare 1 module for PWM mode:

```
OC1CON = 0;                         // Reset OC module
OC1R = 0x60;                        // Initialize PWM Duty Cycle
OC1RS = 0x60;                       // Initialize PWM Duty Cycle Buffer

OC1CONbits.OCM = 6;                 // Configure OC for the PWM mode
```

Set up DMA Channel 3 for in Post Increment mode with Timer2 Request source:

```
unsigned int BufferA[32] __attribute__((space(dma)));
/* Insert code here to initialize BufferA with desired Duty Cycle values */

DMA3CONbits.AMODE = 0;              // Configure DMA for Register indirect mode
                                    // with post-increment
DMA3CONbits.MODE = 0;               // Configure DMA for Continuous mode
DMA3CONbits.DIR   = 1;              // RAM-to-Peripheral data transfers
DMA3PAD = (volatile unsigned int)&OC1RS; // Point DMA to OC1RS
DMA3CNT = 31;                       // 32 DMA request
DMA3REQ = 7;                        // Select Timer2 as DMA Request source

DMA3STA = __builtin_dmaoffset(BufferA);

IFS2bits.DMA3IF = 0;                // Clear the DMA interrupt flag bit
IEC2bits.DMA3IE = 1;                // Set the DMA interrupt enable bit

DMA3CONbits.CHEN = 1;               // Enable DMA
```

Set up Timer2 for Output Compare PWM mode:

```
PR2 = 0xBF;                         // Initialize PWM period
T2CONbits.TON = 1;                  // Start Timer2
```

Set up DMA Channel 3 Interrupt Handler:

```
void __attribute__((__interrupt__)) _DMA3Interrupt(void)
{
    /* Update BufferA with new Duty Cycle values if desired here*/

    IFS2bits.DMA3IF = 0;            //Clear the DMA3 Interrupt Flag
}
```

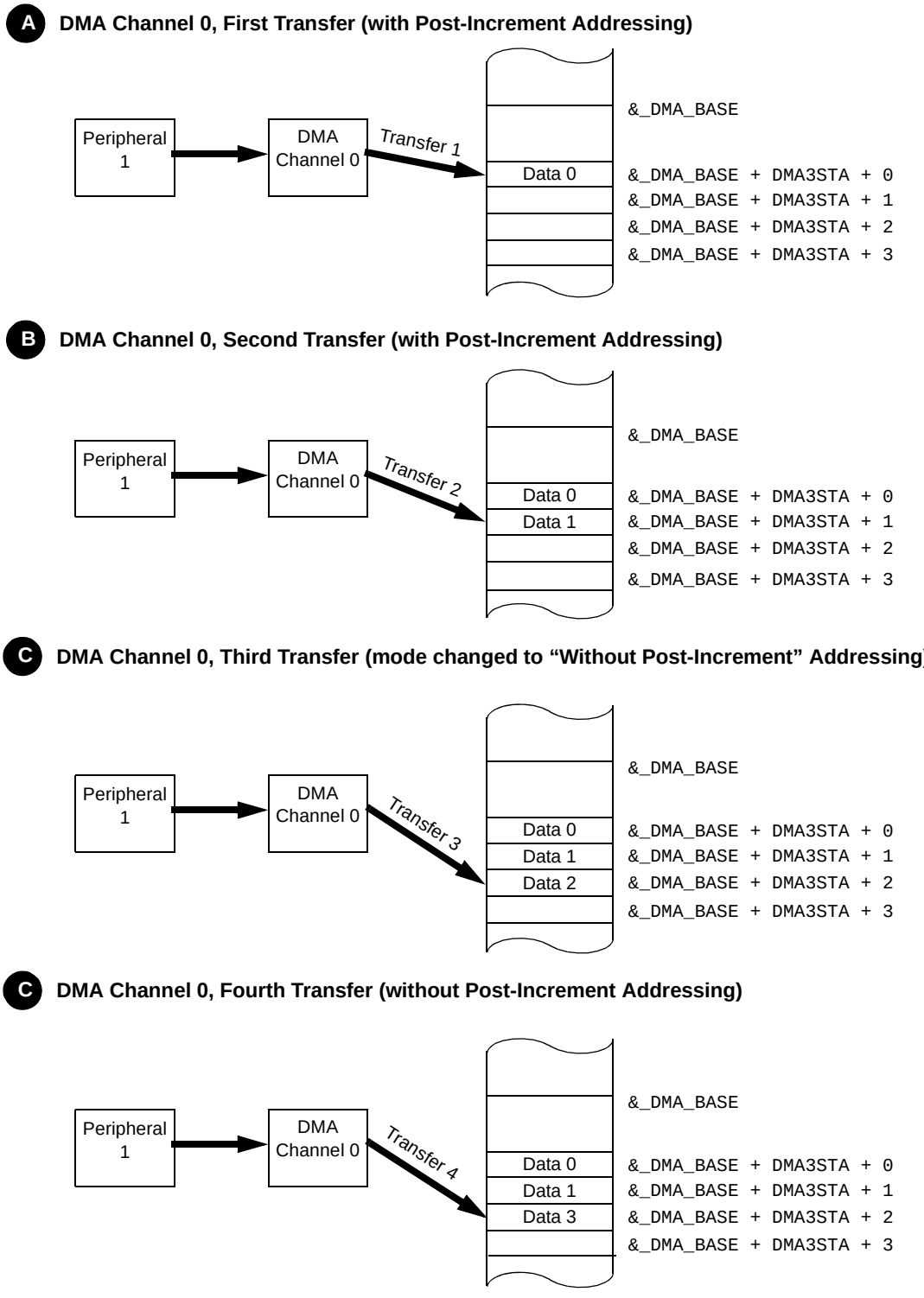
22.6.5 Register Indirect Without Post-increment Addressing Mode

Register Indirect Without Post-Increment Addressing is used to move blocks of data without incrementing the starting address of the data buffer after each transfer. In this mode, the DPSRAM Start Address Offset (DMAxSTA or DMAxSTB) register provides offset to the starting address of the DPSRAM buffer. When the DMA data transfer takes place, the DPSRAM Address does not increment to the next location. So, the next DMA data transfer is initiated to the same DPSRAM address.

This mode is selected by programming the Addressing Mode Select bits (AMODE<1:0>) to '01' in the DMA Channel Control (DMAxCON) register.

If the addressing mode is changed to Register Indirect Without Post-Increment Addressing while the DMA channel is active (i.e., after some DMA transfers have occurred), the DMA DPSRAM address will point to the current DPSRAM buffer location (i.e., not the contents of the DMAxSTA or DMAxSTB, which by then could differ from the current DPSRAM buffer location). Figure 22-9 illustrates data transfer from the peripheral to the DMA DPSRAM, contrasting the use with and without post-increment addressing.

Figure 22-9: Contrast of Data Transfer With and Without Post-Increment Addressing



Example 22-4: Code for Input Capture and DMA with Register Indirect Without Post-Increment Addressing

Set up Input Capture 1 module for DMA operation:

```
IC1CON = 0;                    // Reset IC module
IC1CONbits.ICTMR = 1;          // Select Timer2 contents for capture
IC1CONbits.ICM   = 2;          // Capture every falling edge
IC1CONbits.ICI   = 0;          // Generate DMA request on every capture event
```

Set up Timer2 to be used by Input Capture module:

```
PR2 = 0xBF;                    // Initialize count value
T2CONbits.TON = 1;             // Start timer
```

Set up DMA Channel 0 for no Post Increment mode:

```
unsigned int CaptureValue __attribute__((space(dma)));

DMA0CONbits.AMODE = 1;         // Configure DMA for Register indirect
                               // without post-increment
DMA0CONbits.MODE = 0;          // Configure DMA for Continuous mode
DMA0PAD = (volatile unsigned int)&IC1BUF; // Point DMA to IC1BUF
DMA0CNT = 0;                   // Interrupt after each transfer
DMA0REQ = 1;                   // Select Input Capture module as DMA Request source

DMA3STA = __builtin_dmaoffset(&CaptureValue);

IFS0bits.DMA0IF = 0;           // Clear the DMA interrupt flag bit
IEC0bits.DMA0IE = 1;           // Set the DMA interrupt enable bit

DMA0CONbits.CHEN = 1;          // Enable DMA
```

Set up DMA Channel 0 Interrupt Handler:

```
void __attribute__((__interrupt__)) _DMA3Interrupt(void)
{
    /* Process CaptureValue variable here*/

    IFS0bits.DMA0IF = 0;       //Clear the DMA3 Interrupt Flag
}
```

22.6.6 Peripheral Indirect Addressing Mode

Peripheral Indirect Addressing mode is a special addressing mode where the peripheral, not the DMA channel, provides the variable part of the DPSRAM address. That is, the peripheral generates the Least Significant bits (LSb) of the DPSRAM address while the DMA channel provides the fixed buffer base address. However, the DMA channel continues to coordinate the actual data transfer, keep track of the transfer count and generate the corresponding CPU interrupts.

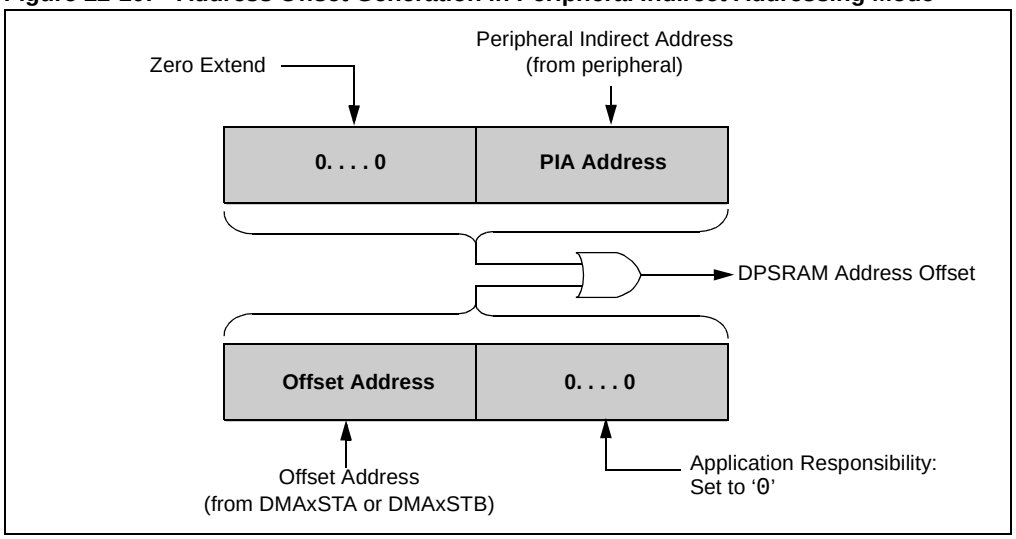
Peripheral Indirect Addressing mode can operate bidirectionally, depending upon the peripheral need, so the DMA channel still needs to be configured appropriately to support target peripheral read or write.

Peripheral Indirect Addressing mode is selected by programming the Addressing Mode Select bits (AMODE<1:0>) to ‘1x’ in the DMA Channel Control (DMAxCON) register.

The DMA capability in Peripheral Indirect Addressing mode can be specifically tailored to meet the needs of each peripheral that supports it. The peripheral defines the address sequence for accessing the data within the DPSRAM, allowing it, for example, to sort incoming ADC data into multiple buffers, relieving the CPU of the task.

If Peripheral Indirect Addressing mode is supported by a peripheral, a DMA request interrupt from that peripheral is accompanied by an address that is presented to the DMA channel. If the DMA channel that responds to the request is also enabled for Peripheral Indirect Addressing, it will logically OR the buffer base address with the zero extended incoming Peripheral Indirect Address to create the actual DPSRAM offset address, as shown in Figure 22-10.

Figure 22-10: Address Offset Generation in Peripheral Indirect Addressing Mode



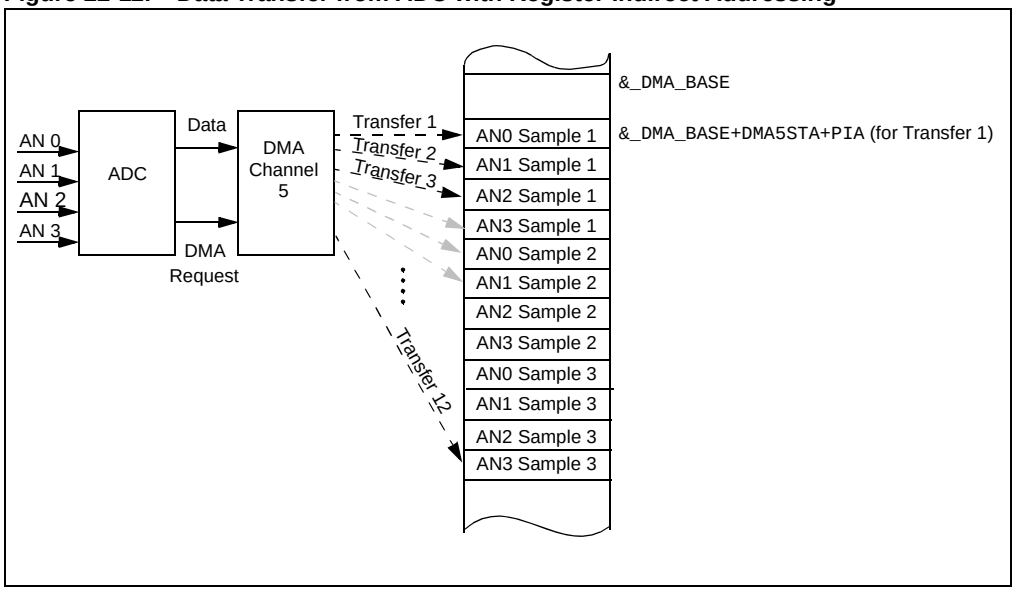
The peripheral determines how many Least Significant address bits it will control. The application program must select a base address for the buffer in DPSRAM and ensure that the corresponding number of Least Significant bits of that address offset are zero. As with other modes, when the DPSRAM Start Address Offset register is read, it returns a value of the latest DPSRAM transfer address offset, which includes the address offset calculation described above. If the DMA channel is not configured for Peripheral Indirect Addressing, the incoming address is ignored and the data transfer occurs as normal.

Peripheral Indirect Addressing mode is compatible with all other operating modes and is currently supported by the ADC and ECAN modules.

22.6.6.1 ADC SUPPORT FOR DMA ADDRESS GENERATION

In Peripheral Indirect Addressing mode, the peripheral defines the addressing sequence, which is more tailored to peripheral functionality. For example, if the ADC is configured to continuously convert inputs 0 through 3 in sequence (For example, 0, 1, 2, 3, 0, 1, and so on) and it is associated with a DMA channel that is configured for Register Indirect Addressing with Post-Increment, DMA transfer moves this data into a sequential buffer as shown in Figure 22-11. Example 22-5 illustrates the code for this configuration.

Figure 22-11: Data Transfer from ADC with Register Indirect Addressing



Example 22-5: Code for Data Transfer from ADC with Register Indirect Addressing

Set up ADC1 for Channel 0-3 sampling:

```
AD1CON1bits.FORM = 3;         // Data Output Format: Signed Fraction (Q15 format)
AD1CON1bits.SSRC = 2;         // Sample Clock Source: GP Timer starts conversion
AD1CON1bits.ASAM = 1;         // Sampling begins immediately after conversion
AD1CON1bits.AD12B = 0;        // 10-bit ADC operation
AD1CON1bits.SIMSAM = 0;       // Samples individual channels sequentially

AD1CON2bits.BUFM = 0;
AD1CON2bits.CSCNA = 1;        // Scan CH0+ Input Selections during Sample A bit
AD1CON2bits.CHPS = 0;         // Converts CH0

AD1CON3bits.ADRC = 0;         // ADC Clock is derived from Systems Clock
AD1CON3bits.ADCS = 63;        // ADC Conversion Clock

//AD1CHS0: A/D Input Select Register
AD1CHS0bits.CH0SA = 0;        // MUXA +ve input selection (AIN0) for CH0
AD1CHS0bits.CH0NA = 0;        // MUXA -ve input selection (VREF-) for CH0

//AD1CHS123: A/D Input Select Register
AD1CHS123bits.CH123SA = 0;    // MUXA +ve input selection (AIN0) for CH1
AD1CHS123bits.CH123NA = 0;    // MUXA -ve input selection (VREF-) for CH1

//AD1CSSH/AD1CSSL: A/D Input Scan Selection Register
AD1CSSH = 0x0000;
AD1CSSL = 0x000F;             // Scan AIN0, AIN1, AIN2, AIN3 inputs
```

Set up Timer3 to trigger ADC1 conversions:

```
TMR3 = 0x0000;
PR3 = 4999;                   // Trigger ADC1 every 125 μs @ 40 MIPS
IFS0bits.T3IF = 0;            // Clear Timer3 interrupt
IEC0bits.T3IE = 0;            // Disable Timer3 interrupt

T3CONbits.TON = 1;            //Start Timer3
```

Set up DMA Channel 5 for Register Indirect with Post-Increment Addressing:

```
unsigned int BufferA[32] __attribute__((space(dma)));
unsigned int BufferB[32] __attribute__((space(dma)));

DMA5CONbits.AMODE = 0;        // Configure DMA for Register indirect mode
                              // with post-increment
DMA5CONbits.MODE = 2;         // Configure DMA for Continuous Ping-Pong mode
DMA5PAD = (volatile unsigned int)&ADC1BUF0; // Point DMA to ADC1BUF0
DMA5CNT = 31;                 // 32 DMA request
DMA5REQ = 13;                 // Select ADC1 as DMA Request source

DMA5STA = __builtin_dmaoffset(BufferA);
DMA5STB = __builtin_dmaoffset(BufferB);

IFS3bits.DMA5IF = 0;          //Clear the DMA interrupt flag bit
IEC3bits.DMA5IE = 1;          //Set the DMA interrupt enable bit

DMA5CONbits.CHEN=1;           // Enable DMA
```

Example 22-5: Code for Data Transfer from ADC with Register Indirect Addressing (Continued)

Set up DMA Channel 5 Interrupt Handler:

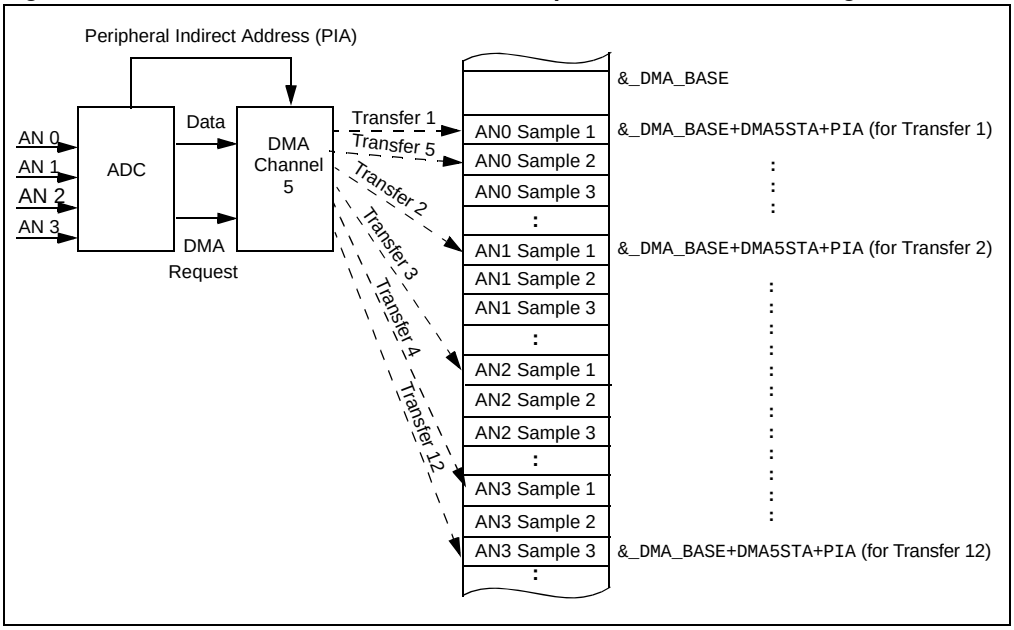
```
unsigned int DmaBuffer = 0;

void __attribute__((__interrupt__)) _DMA5Interrupt(void)
{
    // Switch between Primary and Secondary Ping-Pong buffers
if(DmaBuffer == 0)
    {
        ProcessADCSamples(BufferA);
    }
    else
    {
        ProcessADCSamples(BufferB);
    }

    DmaBuffer ^= 1;

    IFS3bits.DMA5IF = 0;      //Clear the DMA5 Interrupt Flag
}
```

Set up ADC1 for DMA operation:

```
AD1CON1bits.ADDMABM = 0;      // Don't Care: ADC address generation is
                              // Ignored by DMA
AD1CON2bits.SMPI    = 3;      // Don't Care
AD1CON4bits.DMABL   = 3;      // Don't Care

IFS0bits.AD1IF      = 0;      // Clear the A/D interrupt flag bit
IEC0bits.AD1IE      = 0;      // Do Not Enable A/D interrupt
AD1CON1bits.ADON    = 1;      // Turn on the A/D converter
```

A typical algorithm would operate on a per ADC data channel basis, requiring it to either sort transferred data or index it by jumping unwanted data. Either of these methods requires more code and consumes more execution time. ADC Peripheral Indirect Addressing mode defines a special addressing technique where data for each ADC channel is placed into its own buffer. For the above example, if the DMA channel is configured for Peripheral Indirect Addressing mode, DMA transfer moves ADC data into separate buffers, as shown in Figure 22-12.

Figure 22-12: Data Transfer from ADC with Peripheral Indirect Addressing



To enable this kind of ADC addressing, the DMA Buffer Build Mode (ADDMABM) bit in the ADCx Control 1 (ADxCON1) register must be cleared. If this bit is set, the ADC generates addresses in the order of conversion (same as DMA Register Indirect Addressing with Post-Increment mode).

As mentioned earlier, you must pay special attention to the number of Least Significant bits that are reserved for the peripheral when the DPSRAM Start Address Offset registers (DMAxSTA and DMAxSTB) are initialized by the user application. For the ADC, the number of bits will depend on the size and number of the ADC buffers.

The number of ADC buffers is initialized with Increment Rate for DMA Addresses bits SMPI<3:0> in the ADCx Control 2 (ADxCON2) register. The size of each ADC buffer is initialized with Number of DMA Buffer Locations per Analog Input bits DMABL<2:0> in the ADCx Control 4 (ADCxCON4) register. For example, if SMPI<3:0> is initialized to 3 and DMABL<2:0> is initialized to 3, there will be four ADC buffers (SMPI<3:0> + 1), each with eight words ($2^{DMABL<2:0>}$), for the total of 32 words (64 bytes). This means that the address offset that is written into the DMAxSTA and DMAxSTB must have 6 ($2^{6\ bits}$ = 64 bytes) Least Significant bits set to zero.

If the MPLAB® C30 compiler is used to initialized the DMAxSTA and DMAxSTAB registers, proper data alignment must be specified via data attributes. For the above conditions, the code shown in Example 22-6 will properly initialize DMAxSTA and DMAxSTB registers.

Example 22-6: DMA buffer alignment with MPLAB® C30

```
int  BufferA[4][8] __attribute__((space(dma),aligned(64)));
int  BufferB[4][8] __attribute__((space(dma),aligned(64)));

DMA0STA = __builtin_dmaoffset(BufferA);
DMA0STB = __builtin_dmaoffset(BufferB);
```

Example 22-7 illustrates the code for this configuration.

Example 22-7: Code for ADC and DMA with Peripheral Indirect Addressing

Set up ADC1 for Channel 0-3 sampling:

```
AD1CON1bits.FORM = 3;           // Data Output Format: Signed Fraction (Q15 format)
AD1CON1bits.SSRC = 2;           // Sample Clock Source: GP Timer starts conversion
AD1CON1bits.ASAM = 1;           // Sampling begins immediately after conversion
AD1CON1bits.AD12B = 0;          // 10-bit ADC operation
AD1CON1bits.SIMSAM = 0;         // Samples multiple channels sequentially

AD1CON2bits.BUFM = 0;
AD1CON2bits.CSCNA = 1;          // Scan CH0+ Input Selections during Sample A bit
AD1CON2bits.CHPS = 0;           // Converts CH0

AD1CON3bits.ADRC = 0;           // ADC Clock is derived from Systems Clock
AD1CON3bits.ADCS = 63;          // ADC Conversion Clock

//AD1CHS0: A/D Input Select Register
AD1CHS0bits.CH0SA = 0;          // MUXA +ve input selection (AIN0) for CH0
AD1CHS0bits.CH0NA = 0;          // MUXA -ve input selection (VREF-) for CH0

//AD1CHS123: A/D Input Select Register
AD1CHS123bits.CH123SA = 0;      // MUXA +ve input selection (AIN0) for CH1
AD1CHS123bits.CH123NA = 0;      // MUXA -ve input selection (VREF-) for CH1

//AD1CSSH/AD1CSSL: A/D Input Scan Selection Register
AD1CSSH = 0x0000;
AD1CSSL = 0x000F;               // Scan AIN0, AIN1, AIN2, AIN3 inputs
```

Set up Timer3 to trigger ADC1 conversions:

```
TMR3 = 0x0000;
PR3 = 4999; // Trigger ADC1 every 125usec
IFS0bits.T3IF = 0;              // Clear Timer3 interrupt
IEC0bits.T3IE = 0;              // Disable Timer3 interrupt

T3CONbits.TON = 1;              //Start Timer3
```

Set up DMA Channel 5 for Peripheral Indirect Addressing:

```
struct
{
    unsigned int Adc1Ch0[8];
    unsigned int Adc1Ch1[8];
    unsigned int Adc1Ch2[8];
    unsigned int Adc1Ch3[8];
} BufferA __attribute__((space(dma)));

struct
{
    unsigned int Adc1Ch0[8];
    unsigned int Adc1Ch1[8];
    unsigned int Adc1Ch2[8];
    unsigned int Adc1Ch3[8];
} BufferB __attribute__((space(dma)));

int BufferA[4][8] __attribute__((space(dma), aligned(64)));
BufferB[4][8] __attribute__((space(dma), aligned(64)));

DMA5CONbits.AMODE = 2;          // Configure DMA for Peripheral indirect mode
DMA5CONbits.MODE = 2;           // Configure DMA for Continuous Ping-Pong mode
DMA5PAD = (volatile unsigned int)&ADC1BUF0; // Point DMA to ADC1BUF0
DMA5CNT = 31;                   // 32 DMA request (4 buffers, each with 8 words)
DMA5REQ = 13;                   // Select ADC1 as DMA Request source
DMA5STA = __builtin_dmaoffset(BufferA);
DMA5STB = __builtin_dmaoffset(BufferB);

IFS3bits.DMA5IF = 0;            //Clear the DMA interrupt flag bit
IEC3bits.DMA5IE = 1;            //Set the DMA interrupt enable bit

DMA5CONbits.CHEN=1;             // Enable DMA
```

Example 22-7: Code for ADC and DMA with Peripheral Indirect Addressing (Continued)

Set up DMA Channel 5 Interrupt Handler:

```
unsigned int DmaBuffer = 0;

void __attribute__((__interrupt__)) _DMA5Interrupt(void)
{
    // Switch between Primary and Secondary Ping-Pong buffers
if(DmaBuffer == 0)
    {
        ProcessADCSamples(BufferA.Adc1Ch0);
        ProcessADCSamples(BufferA.Adc1Ch1);
        ProcessADCSamples(BufferA.Adc1Ch2);
        ProcessADCSamples(BufferA.Adc1Ch3);
    }
    else
    {
        ProcessADCSamples(BufferB.Adc1Ch0);
        ProcessADCSamples(BufferB.Adc1Ch1);
        ProcessADCSamples(BufferB.Adc1Ch2);
        ProcessADCSamples(BufferB.Adc1Ch3);
    }

    DmaBuffer ^= 1;

    IFS3bits.DMA5IF = 0;        //Clear the DMA5 Interrupt Flag
}
```

Set up ADC1 for DMA operation:

```
AD1CON1bits.ADDMABM = 0;        // DMA buffers are built in scatter/gather mode
AD1CON2bits.SMPI    = 3;        // 4 ADC buffers
AD1CON4bits.DMABL   = 3;        // Each buffer contains 8 words

IFS0bits.AD1IF      = 0;        // Clear the A/D interrupt flag bit
IEC0bits.AD1IE      = 0;        // Do Not Enable A/D interrupt
AD1CON1bits.ADON    = 1;        // Turn on the A/D converter
```

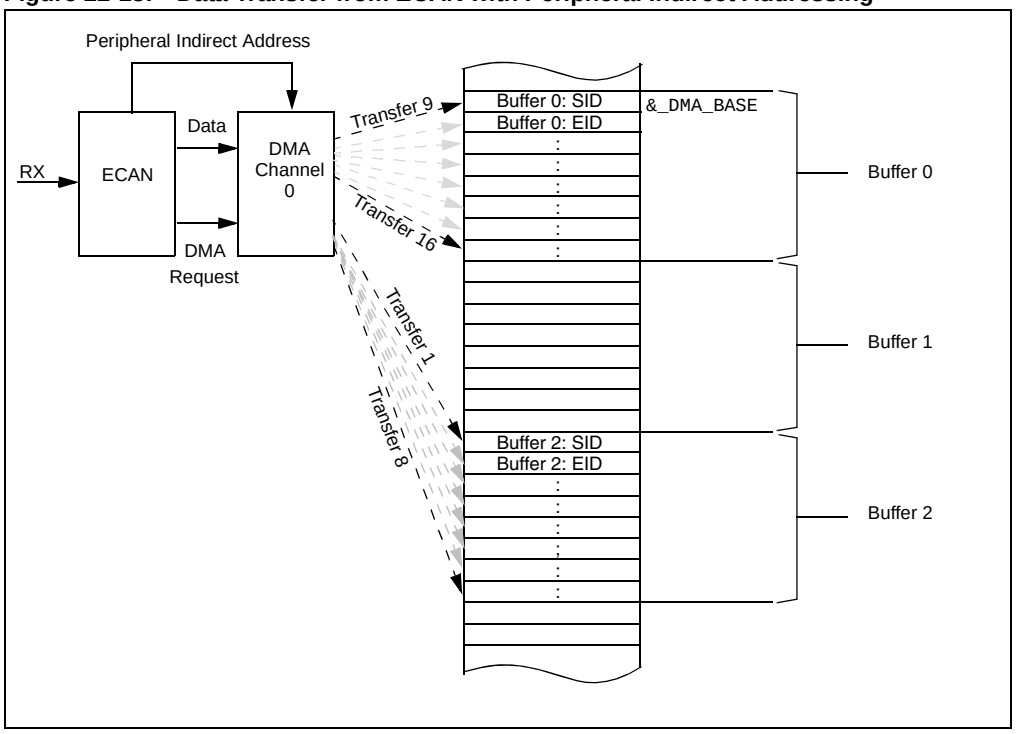
22.6.6.2 ECAN SUPPORT FOR DMA ADDRESS GENERATION

Peripheral Indirect Addressing can also be used with the ECAN module to let ECAN define more specific addressing functionality. When the dsPIC33F device filters and receives messages via the CAN bus, the messages can be categorized into two groups:

- Received messages that must be processed
- Received messages that must be forwarded to other CAN nodes without processing

In the first case, received messages must be reconstructed into buffers of eight words each before they can be processed by the user application. With multiple ECAN buffers located in the DMA RAM, it would be easier to let the ECAN peripheral generate RAM addresses for incoming (or outgoing) data, as shown in Figure 22-13. In this example, Buffer 2 is received first, followed by Buffer 0. The ECAN module generates destination addresses to properly place data in the DMA RAM (Peripheral Indirect Addressing).

Figure 22-13: Data Transfer from ECAN with Peripheral Indirect Addressing



As mentioned earlier, you must pay special attention to the number of Least Significant bits that are reserved for the peripheral when the DPSRAM Start Address Offset registers (DMAxSTA and DMAxSTB) are initialized by the user application and the DMA is operating in Peripheral Indirect Addressing mode. For the ECAN, the number of bits depends on the number of ECAN buffers defined by the DMA Buffer Size bits (DMABS<2:0>) in the ECAN FIFO Control register (CiFCTRL).

For example, if the ECAN module reserves 12 buffers by setting DMABS<2:0> bits to '3', there will be 12 buffers with eight words each, for a total of 96 words (192 bytes). This means that the address offset that is written into the DMAxSTA and DMAxSTB registers must have 8 ($2^{8\ bits}$ = 256 bytes) Least Significant bits set to '0'. If the MPLAB C30 compiler is used to initialize the DMAxSTA register, proper data alignment must be specified via data attributes. For the above example, the code in Example 22-8 properly initializes the DMAxSTA register.

Example 22-8: DMA Buffer Alignment with MPLAB® C30

```
int  BufferA[12][8] __attribute__((space(dma),aligned(256)));

DMA0STA = __builtin_dmaoffset(&BufferA[0][0]);
```

Example 22-9 illustrates the code for this configuration.

However, processing of incoming messages may not always be a requirement. For instance, in some automotive applications, received messages can simply be forwarded to another node rather than being processed by the CPU. In this case, received buffers do not have to be sorted in memory and can be forwarded as they become available.

This mode of data transfer can be achieved with the DMA in Register Indirect Addressing with Post-Increment. Figure 22-14 illustrates this scenario.

Example 22-9: Code for ECAN and DMA with Peripheral Indirect Addressing

Set up ECAN1 with two filters:

```
/* Initialize ECAN clock first. See ECAN section for example code */


C1CTRL1bits.WIN = 1;          // Enable filter window
C1FEN1bits.FLTEN0 = 1;        // Filter 0 is enabled
C1FEN1bits.FLTEN1 = 1;        // Filter 1 is enabled
C1BUFPNT1bits.F0BP = 0;       // Filter 0 points to Buffer0
C1BUFPNT1bits.F1BP = 2;       // Filter 1 points to Buffer2

C1RXF0SID = 0xFFEA;           // Filter 0 configuration
C1RXF0EID = 0xFFFF;

C1RXF1SID = 0xFFEB;           // Filter 1 configuration
C1RXF1EID = 0xFFFF;

C1FMSKSEL1bits.F0MSK = 0;     // Mask 0 used for both filters
C1FMSKSEL1bits.F1MSK = 0;     // Mask 0 used for both filters
C1RXM0SID = 0xFFEB;
C1RXM0EID = 0xFFFF;

C1FCTRLbits.DMABS = 3;        // 12 buffers in DMA RAM
C1FCTRLbits.FSA   = 3;        // FIFO starts from TX/RX Buffer 3

C1CTRL1bits.WIN     = 0;
C1TR01CONbits.TXEN0 = 0;      // Buffer 0 is a receive buffer
C1TR23CONbits.TXEN2 = 0;      // Buffer 2 is a receive buffer

C1TR01CONbits.TX0PRI = 0b11;  //High Priority
C1TR01CONbits.TX1PRI = 0b10;  //Intermediate High Priority

C1CTRL1bits.REQOP = 0;// Enable Normal Operation Mode
```

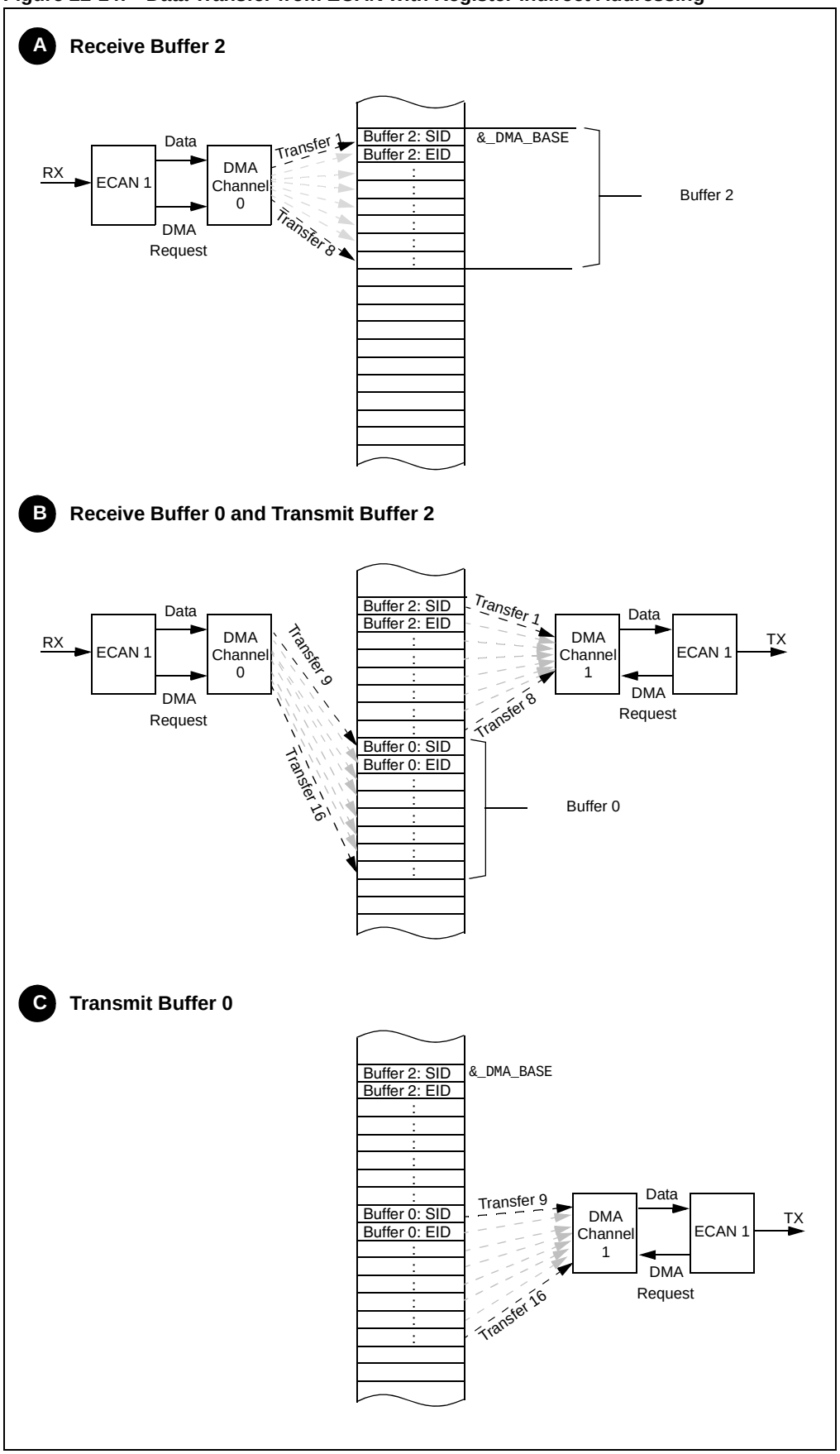
Set up DMA Channel 0 for Peripheral Indirect Addressing:

```
unsigned int Ecan1Rx[12][8] __attribute__((space(dma)));// 12 buffers, 8
words each

DMA0CONbits.AMODE = 2;        // Continuous mode, single buffer
DMA0CONbits.MODE = 0;         // Peripheral Indirect Addressing

DMA0PAD = (volatile unsigned int) &C1RXD;    // Point to ECAN1 RX register
DMA0STA = __builtin_dmaoffset(Ecan1Rx);      // Point DMA to ECAN1 buffers

DMA0CNT = 7;                  // 8 DMA request (1 buffer, each with 8 words)
DMA0REQ = 0x22;               // Select ECAN1 RX as DMA Request source

IEC0bits.DMA0IE = 1;          // Enable DMA Channel 0 interrupt
DMA0CONbits.CHEN = 1;         // Enable DMA Channel 0
```

Set up DMA Interrupt Handler:

```
void __attribute__((__interrupt__)) _DMA0Interrupt(void)
{
   ProcessData(Ecan1Rx[C1VECbits.ICODE]);    // Process received buffer;

   IFS0bits.DMA0IF = 0;       // Clear the DMA0 Interrupt Flag;
}
```

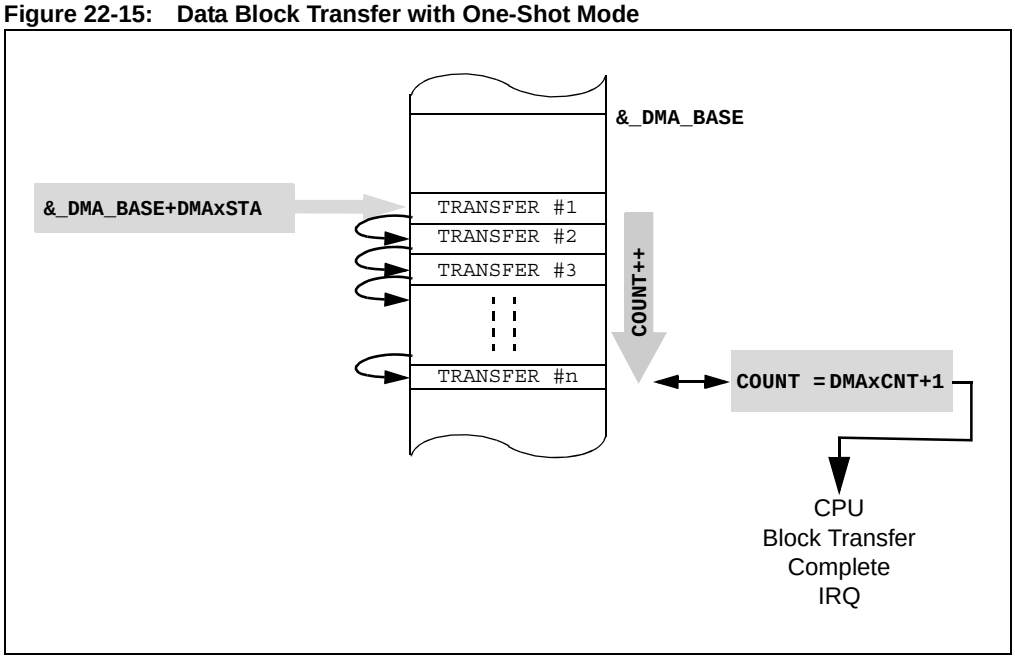
Figure 22-14: Data Transfer from ECAN with Register Indirect Addressing



22.6.7 One-Shot Mode

One-Shot mode is used by the application program when repetitive data transfer is not required. One-Shot mode is selected by programming the Operating Mode Select bits (MODE<1:0>) to 'x1' in the DMA Channel Control (DMAxCON) register. In this mode, when the entire data block is moved (block length as defined by DMAxCNT), the data block end is detected and the channel is automatically disabled (i.e., the CHEN bit in the DMA Channel Control (DMAxCON) register is cleared by the hardware). Figure 22-15 illustrates One-Shot mode.

Figure 22-15: Data Block Transfer with One-Shot Mode



If the HALF bit is set in the DMA Channel Control (DMAxCON) register, the DMAxIF bit is set (and the DMA interrupt is generated, if enabled by the application program) when half of the data block transfer is complete and the channel remains enabled. When the full-block transfer is complete, no interrupt flag is set and the channel is automatically disabled. See **22.6.3 "Full or Half-Block Transfer Interrupts"** for information on how to set up the DMA channel to interrupt on both half and full-block transfer.

If the channel is re-enabled by setting CHEN in DMAxCON to '1', the block transfer takes place from the start address, as provided by DPSRAM Start Address Offset (DMAxSTA and DMAxSTB) registers. Example 22-10 illustrates the code for One-Shot operation.

Example 22-10: Code for UART and DMA with One-Shot Mode

Set up UART for RX and TX:

```
#define FCY      40000000
#define BAUDRATE 9600
#define BRGVAL   ((FCY/BAUDRATE)/16)-1

U2MODEbits.STSEL = 0;     // 1-stop bit
U2MODEbits.PDSEL = 0;     // No Parity, 8-data bits
U2MODEbits.ABAUD = 0;     // Autobaud Disabled

U2BRG = BRGVAL;// BAUD Rate Setting for 9600

U2STAbits.UTXISEL0 = 0;   // Interrupt after one TX character is transmitted
U2STAbits.UTXISEL1 = 0;
U2STAbits.URXISEL = 0;    // Interrupt after one RX character is received

U2MODEbits.UARTEN   = 1;  // Enable UART
U2STAbits.UTXEN     = 1;  // Enable UART TX
```

Example 22-10: Code for UART and DMA with One-Shot Mode (Continued)

Set up DMA Channel 0 to Transmit in One-Shot, Single-Buffer mode:

```
unsigned int BufferA[8] __attribute__((space(dma)));
unsigned int BufferB[8] __attribute__((space(dma)));


DMA0CON = 0x2001;             // One-Shot, Post-Increment, RAM-to-Peripheral
DMA0CNT = 7;                  // 8 DMA requests
DMA0REQ = 0x001F;             // Select UART2 Transmitter

DMA0PAD = (volatile unsigned int) &U2TXREG;
DMA0STA = __builtin_dmaoffset(BufferA);

IFS0bits.DMA0IF = 0;          // Clear DMA Interrupt Flag
IEC0bits.DMA0IE = 1;          // Enable DMA interrupt
```

Set up DMA Channel 1 to Receive in Continuous Ping-Pong mode:

```
DMA1CON = 0x0002;             // Continuous, Ping-Pong, Post-Inc., Periph-RAM
DMA1CNT = 7;                  // 8 DMA requests
DMA1REQ = 0x001E;             // Select UART2 Receiver

DMA1PAD = (volatile unsigned int) &U2RXREG;
DMA1STA = __builtin_dmaoffset(BufferA);
DMA1STB = __builtin_dmaoffset(BufferB);


IFS0bits.DMA1IF = 0;          // Clear DMA interrupt
IEC0bits.DMA1IE = 1;          // Enable DMA interrupt
DMA1CONbits.CHEN = 1;         // Enable DMA Channel
```

Set up DMA Interrupt Handler:

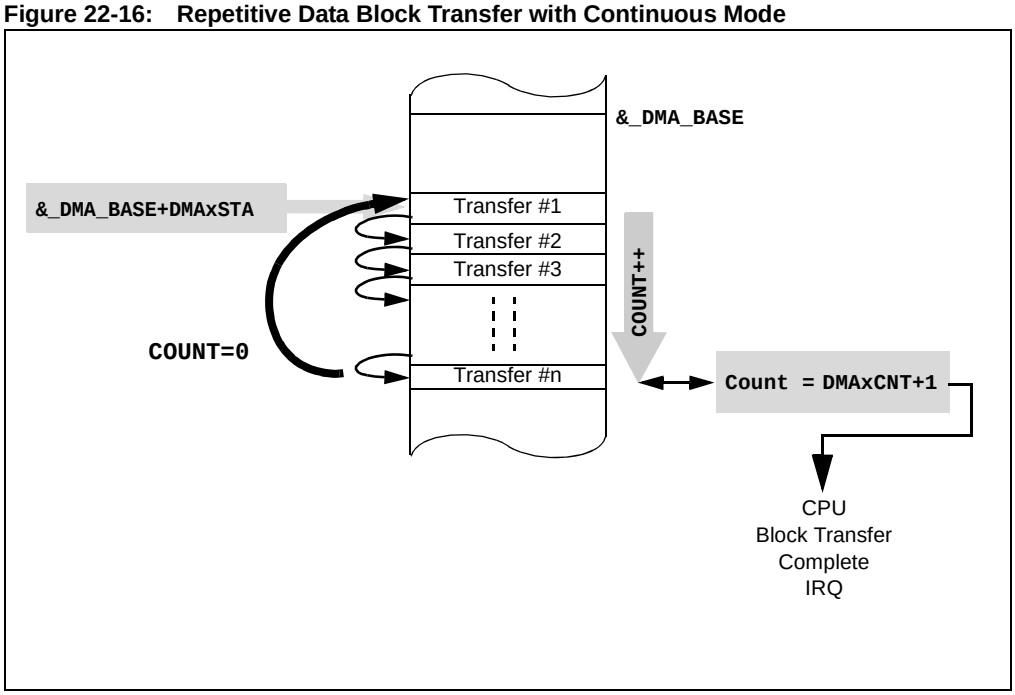
```
void __attribute__((__interrupt__)) _DMA0Interrupt(void)
{
   IFS0bits.DMA0IF = 0;       // Clear the DMA0 Interrupt Flag;
}


void __attribute__((__interrupt__)) _DMA1Interrupt(void)
{
   static unsigned int BufferCount = 0;      // Keep record of which buffer
                                             // contains RX Data

   if(BufferCount == 0)
   {
    DMA0STA = __builtin_dmaoffset(BufferA);  // Point DMA 0 to data
                                             // to be transmitted
   }
   else
 {
    DMA0STA = __builtin_dmaoffset(BufferB);  // Point DMA 0 to data
                                             // to be transmitted
   }

   DMA0CONbits.CHEN = 1;      // Enable DMA0 Channel
   DMA0REQbits.FORCE = 1;     // Manual mode: Kick-start the 1st transfer

   BufferCount ^= 1;
   IFS0bits.DMA1IF = 0;       // Clear the DMA1 Interrupt Flag
}
```

22.6.8 Continuous Mode

Continuous mode is used by the application program when repetitive data transfer is required throughout the life of the program.

This mode is selected by programming the Operating Mode Select bits (MODE<1:0>) to 'x0' in the DMA Channel Control (DMAxCON) register. In this mode, when the entire data block is moved (block length as defined by DMAxCNT), the data block end is detected and the channel remains enabled. During the last data transfer, DMA DPSRAM address resets back to (primary) DPSRAM Start Address Offset A (DMAxSTA) register. Figure 22-16 illustrates Continuous mode.

Figure 22-16: Repetitive Data Block Transfer with Continuous Mode



If the HALF bit is set in the DMA Channel Control (DMAxCON) register, the DMAxIF is set (and DMA interrupt is generated, if enabled) when half of the data block transfer is complete. The channel remains enabled. When the full-block transfer is complete, no interrupt flag is set and the channel remains enabled. See **22.6.3 "Full or Half-Block Transfer Interrupts"**, for information on how to set up the DMA channel to interrupt on both half and full-block transfer.

22.6.9 Ping-Pong Mode

Ping-Pong mode allows the CPU to process one buffer while a second buffer operates with the DMA channel. The net result is that the CPU has the entire DMA block transfer time in which to process the buffer currently not being used by the DMA channel. Of course, this transfer mode doubles the amount of DPSRAM needed for a given size of buffer.

In all DMA operating modes, when the DMA channel is enabled, the (primary) DMA Channel x DPSRAM Start Address Offset A (DMAxSTA) register is selected by default to generate the initial DPSRAM effective address. As each block transfer completes and the DMA channel is reinitialized, the buffer start address is sourced from the same DMAxSTA register.

In Ping-Pong mode, the buffer start address is derived from two registers:

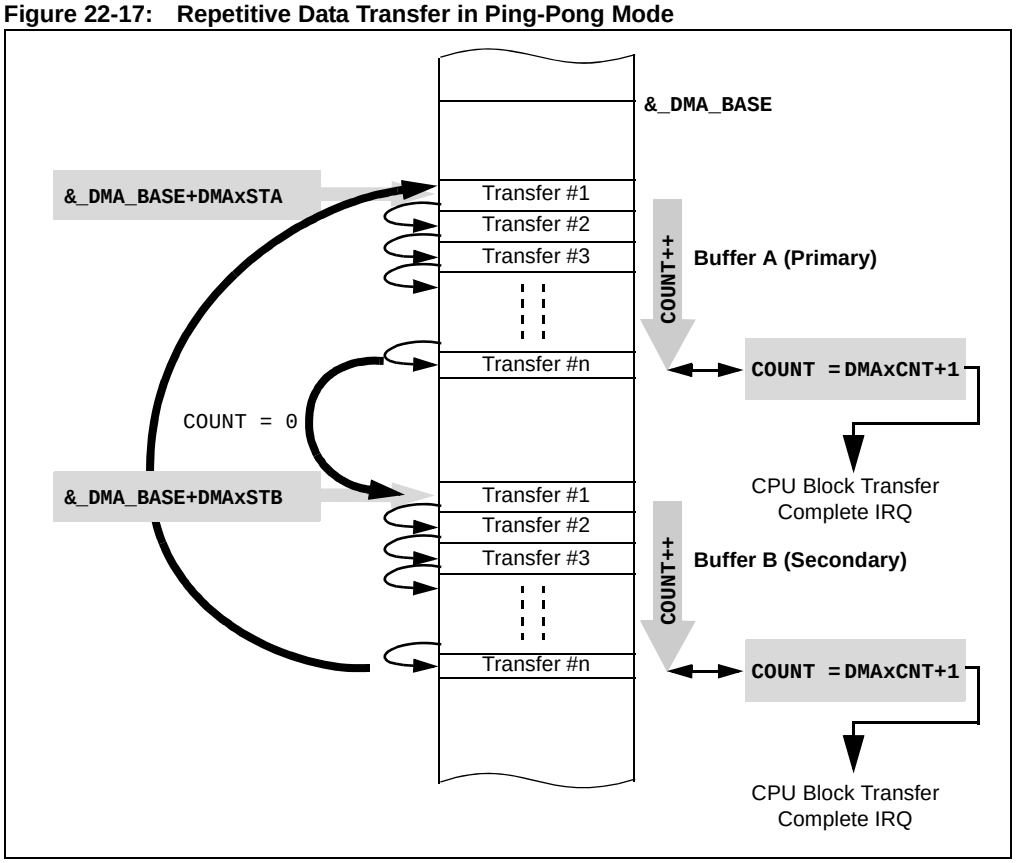
- Primary: DMA Channel x DPSRAM Start Address Offset A (DMAxSTA) register
- Secondary: DMA Channel x DPSRAM Start Address Offset B (DMAxSTB) register

The DMA uses a secondary buffer for alternate block transfers. As each block transfer completes and the DMA channel is reinitialized, the buffer start address is derived from the alternate register.

Ping-Pong mode is selected by programming Operating Mode Select bits (MODE<1:0>) to '1x' in the DMA Channel Control (DMAxCON) register.

If Continuous mode is selected while the DMA is operating in Ping-Pong mode, the DMA responds by reinitializing to point to the secondary buffer after transferring the primary buffer, and then transfers the secondary buffer. Subsequent block transfers alternate between primary and secondary buffers. Interrupts are generated (if enabled by the application program) after each buffer is transferred. Figure 22-17 illustrates Ping-Pong mode with Continuous operation. Example 22-11 illustrates the code used for Ping-Pong operation using the DCI module as an example.

Figure 22-17: Repetitive Data Transfer in Ping-Pong Mode



Example 22-11: Code for DCI and DMA with Continuous Ping-Pong Operation

Set up DCI for RX and TX:

```
#define FCY    40000000
#define FS     48000
#define FCSCK  64*FS
#define BCGVAL (FCY/(2*FS))-1

DCICON1bits.CSCKD = 0; // Serial Bit Clock (CSCK pin) is output
DCICON1bits.CSCKE = 0; // Data sampled on falling edge of CSCK
DCICON1bits.COFSD = 0; // Frame Sync Signal is output
DCICON1bits.UNFM = 0;  // Transmit '0's on a transmit underflow
DCICON1bits.CSDOM = 0; // CSDO pin drives '0's during disabled TX time slots
DCICON1bits.DJST = 0;  // TX/RX starts 1 serial clock cycle after frame sync pulse
DCICON1bits.COFSM = 1; // Frame Sync Signal set up for I2S mode

DCICON2bits.BLEN = 0;  // One data word will be buffered between interrupts
DCICON2bits.COFSG = 1; // Data frame has 2 words: LEFT & RIGHT samples
DCICON2bits.WS = 15;   // Data word size is 16 bits

DCICON3 = BCG_VAL;// Set up CSCK Bit Clock Frequency

TSCONbits.TSE0 = 1;    // Transmit on Time Slot 0
TSCONbits.TSE1 = 1;    // Transmit on Time Slot 1
RSCONbits.RSE0 = 1;    // Receive on Time Slot 0
RSCONbits.RSE1 = 1;    // Receive on Time Slot 1
```

Set up DMA Channel 0 for Transmit in Continuous Ping-Pong mode:

```
unsigned int TxBufferA[16] __attribute__((space(dma)));
unsigned int TxBufferB[16] __attribute__((space(dma)));

DMA0CON = 0x2002;      // Ping-Pong, Continous, Post-Increment, RAM-to-Peripheral
DMA0CNT = 15;          // 15 DMA requests
DMA0REQ = 0x003C;      // Select DCI as DMA Request source

DMA0PAD = (volatile unsigned int) &TXBUF0;
DMA0STA = __builtin_dmaoffset(TxBufferA);
DMA0STB = __builtin_dmaoffset(TxBufferB);

IFS0bits.DMA0IF = 0;  // Clear DMA Interrupt Flag
IEC0bits.DMA0IE = 1;  // Enable DMA interrupt
DMA0CONbits.CHEN = 1;  // Enable DMA Channel
```

Set up DMA Channel 1 for Receive in Continuous Ping-Pong mode:

```
unsigned int RxBufferA[16] __attribute__((space(dma)));
unsigned int RxBufferB[16] __attribute__((space(dma)));


DMA1CON = 0x0002;      // Continuous, Ping-Pong, Post-Inc., Periph-RAM
DMA1CNT = 15;          // 16 DMA requests
DMA1REQ = 0x003C;      // Select DCI as DMA Request source

DMA1PAD = (volatile unsigned int) &RXBUF0;
DMA1STA = __builtin_dmaoffset(RxBufferA);
DMA1STB = __builtin_dmaoffset(RxBufferB);

IFS0bits.DMA1IF = 0;// Clear DMA interrupt
IEC0bits.DMA1IE = 1;// Enable DMA interrupt
DMA1CONbits.CHEN = 1;// Enable DMA Channel
```

Example 22-11: Code for DCI and DMA with Continuous Ping-Pong Operation (Continued)

Set up DMA Interrupt Handler:

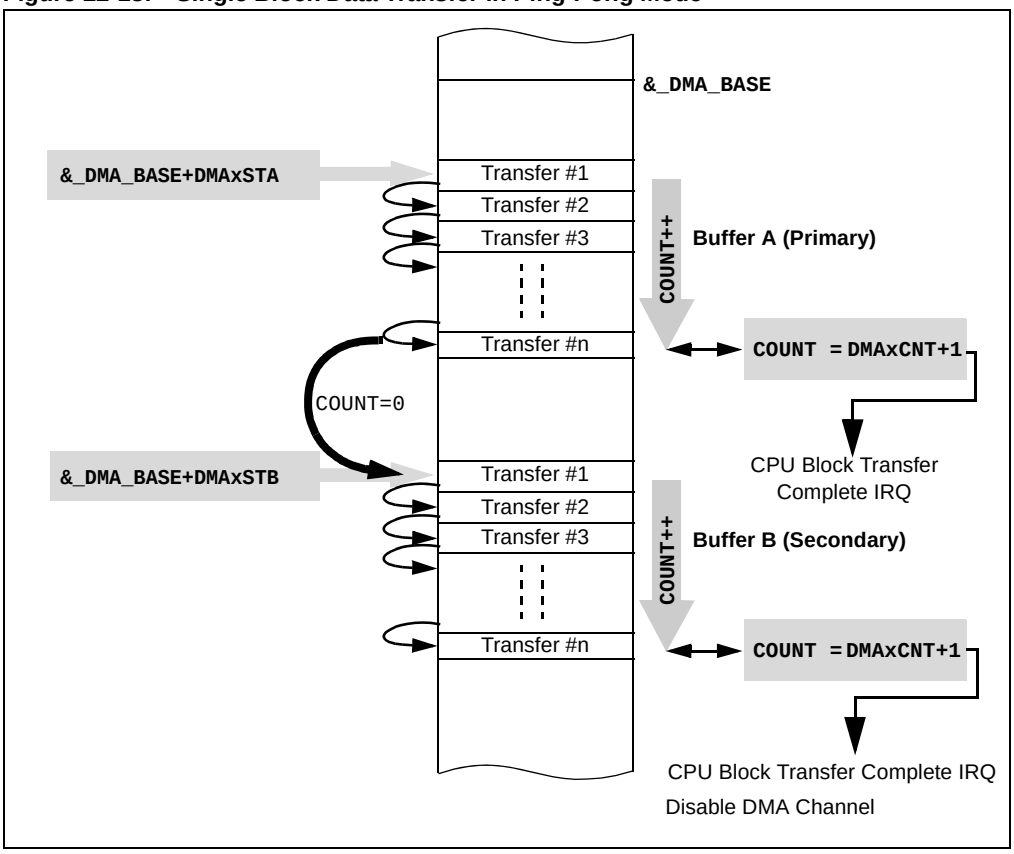
```
void __attribute__((__interrupt__)) _DMA0Interrupt(void)
{
   static unsigned int TxBufferCount = 0;// Keep record of which buffer
                                         // has RX Data

   if(BufferCount == 0)
   {
    /* Notify application that TxBufferA has been transmitted */
   }
   else
   {
    /* Notify application that TxBufferB has been transmitted */
   }

   BufferCount ^= 1;
   IFS0bits.DMA0IF = 0;                  // Clear the DMA0 Interrupt Flag;
}


void __attribute__((__interrupt__)) _DMA1Interrupt(void)
{
   static unsigned int RxBufferCount = 0; // Keep record of which buffer
                                         // has RX Data

   if(BufferCount == 0)
   {
    /* Notify application that RxBufferA has been received */
   }
   else
   {
    /* Notify application that RxBufferB has been received */   }

   BufferCount ^= 1;
   IFS0bits.DMA1IF = 0;                  // Clear the DMA1 Interrupt Flag
}
```

Enable DCI:

```
/* Force First two words to fill-in TX buffer/shift register */
DMA0REQbits.FORCE = 1;
while(DMA0REQbits.FORCE == 1);

DMA0REQbits.FORCE = 1;
while(DMA0REQbits.FORCE == 1);

DCICON1bits.DCIEN = 1;                   // Enable DCI
```

If One-Shot mode is selected while the DMA is operating in Ping-Pong mode, the DMA responds by reinitializing to point to the secondary buffer after transferring primary buffer and then transfers the secondary buffer. Subsequent block transfers will not occur, however, because the DMA channel disables itself. Figure 22-18 illustrates One-Shot data transfer in Ping-Pong mode.

Figure 22-18: Single Block Data Transfer in Ping-Pong Mode



22.6.10 Manual Transfer Mode

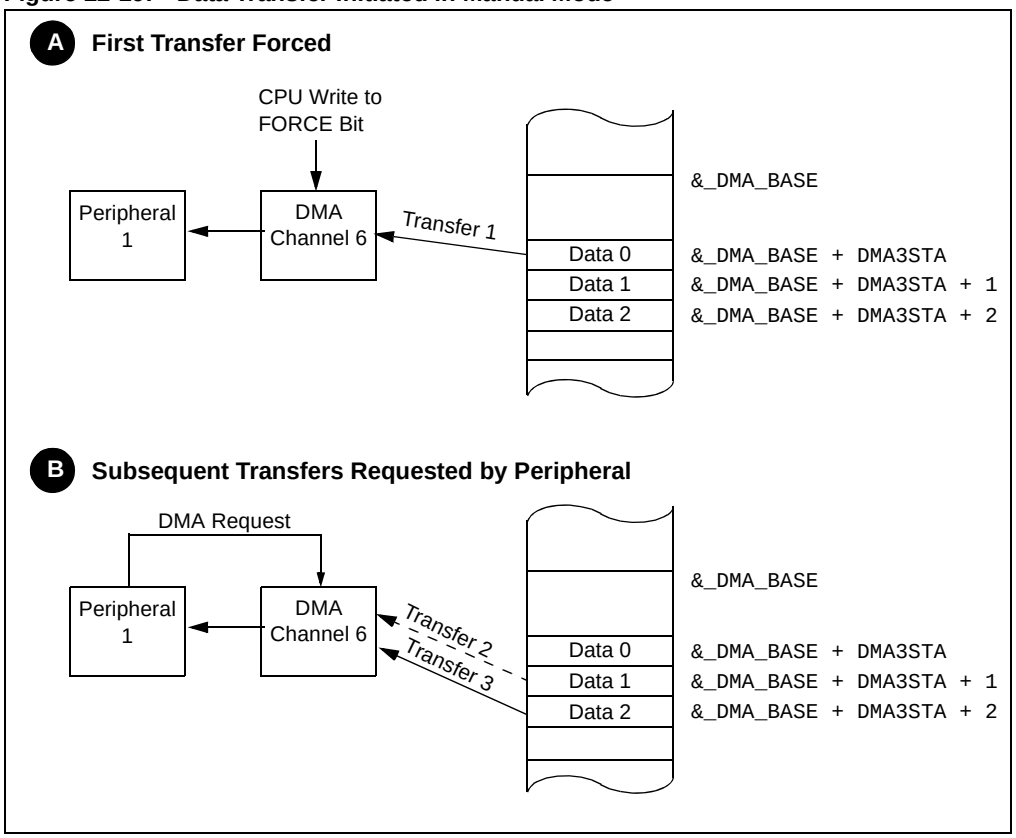
For peripherals that are sending data to the DPSRAM using the DMA controller, the DMA data transfer starts automatically after the DMA channel and peripheral are initialized. When the peripheral is ready to move data to the DPSRAM, it issues a DMA request. If data also needs to be sent to the peripheral at this time, the same DMA request can be used to activate another channel to read data from DPSRAM and write it to the peripheral.

On the other hand, if the application only needs to send data to a peripheral (from a DPSRAM buffer) an initial (manual) data load into the peripheral may be required to start the process (see **22.7 "Starting DMA Transfer"**). This process could be initiated with conventional software. However, a more convenient approach is to simply mimic the channel DMA request by setting a bit within the selected DMA channel. The DMA channel processes the forced request as it would any other request and transfers the first data element to start the sequence. When the peripheral is ready for the next piece of data, it sends a normal DMA request and the DMA sends the next data element. This process is illustrated in Figure 22-19.

A manual DMA request can be created by setting the FORCE bit in the DMA Channel x IRQ Select (DMAxREQ) register. Once set, the FORCE bit can not be cleared by the user application. It must be cleared by hardware when the forced DMA transfer is complete. Depending on when the FORCE bit is set, these special conditions apply:

- Setting the FORCE bit while DMA transfer is in progress has no effect and is ignored.
- Setting the FORCE bit while the channel x is being configured (i.e., setting the FORCE bit during the same write that configures DMA channel) can result in unpredictable behavior and should be avoided.
- An attempt to set the FORCE bit while a peripheral interrupt request is pending (for this channel) is discarded in favor of the interrupt-based request. However, an error condition is generated by setting both the DMA RAM Write Collision Flag bit (XWCOLx) and the Peripheral Write Collision Flag bit (PWCOLx) in the DMA Controller Status 0 (DMACS0) register. See **22.10 "Data Write Collisions"** for more details.

Figure 22-19: Data Transfer Initiated in Manual Mode



22.6.11 Null Data Write Mode

Null Data Write mode is the most useful in applications in which sequential reception of data is required without any data transmission like SPI.

The SPI is essentially a simple shift register, clocking a bit of data in and out for each clock period. However, an unusual situation arises when the SPI is configured in Master mode (i.e., when the SPI is to be the source of the clock) but only received data is of interest. In this case, something must be written to the SPI data register in order to start the SPI data clock and receive the external data.

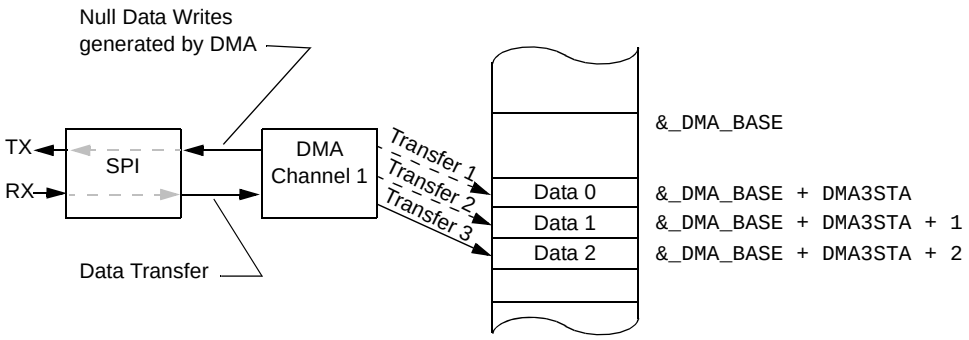
It would be possible to allocate two DMA channels, one for data reception and the other to simply feed null, or zero, data into the SPI. However, a more efficient solution is to use a DMA Null Data Write mode that automatically writes a null value to the SPI data register after each data element has been received and transferred by the DMA channel configured for peripheral data reads.

If the Null Data Peripheral Write Mode Select bit (NULLW) is set in the DMA Channel x Control (DMAxCON) register, and the DMA channel is configured to read from the peripheral, then the DMA channel also executes a null (all zeros) write to the peripheral address in the same cycle as the peripheral data read. This write occurs across the peripheral bus concurrently with the (data) write to the DPSRAM (across the DPSRAM bus). Figure 22-20 illustrates this mode.

During normal operation in this mode, the Null Data Write can only occur in response to a peripheral DMA request (i.e., after data has been received and is available for transfer). An initial CPU write to the peripheral is required to start reception of the first word, after which the DMA takes care of all subsequent peripheral (null) data writes. That is, the CPU null write starts the SPI (master) sending/receiving data which in turn eventually generates a DMA request to move the newly received data.

Alternatively, a forced DMA transfer could be used to ‘kick start’ the process. However, this will also include a redundant peripheral read (data not valid) and an associated DPSRAM pointer adjustment, which must be taken into account.

Figure 22-20: Data Transfer With Null Data Write Mode



Example 22-12: SPI and DMA With Null Data Write Mode

Set up SPI for Master mode:

```
SPI1CON1bits.MODE16 = 1;      // Communication is word-wide (16 bits)
SPI1CON1bits.MSTEN = 1;       // Master Mode Enabled
SPI1STATbits.SPIEN = 1;       // Enable SPI Module
```

Set up DMA Channel 1 for Null Data Write mode:

```
unsigned int BufferA[16] __attribute__((space(dma)));
unsigned int BufferB[16] __attribute__((space(dma)));

DMA1CON = 0x0802;             // Null Write, Continuous, Ping-Pong,
                              // Post-Increment, Periph-to-RAM
DMA1CNT = 15;                 // Transfer 16 words at a time
DMA1REQ = 0x000A;             // Select SPI1 as DMA request source

DMA1STA = __builtin_dmaoffset(BufferA);
DMA1STB = __builtin_dmaoffset(BufferB);
DMA1PAD = (volatile unsigned int) &SPI1BUF;

IFS0bits.DMA1IF = 0;
IEC0bits.DMA1IE = 1;          // Enable DMA interrupt
DMA1CONbits.CHEN = 1;         // Enable DMA Channel

DMA1REQbits.FORCE = 1;        // Force First word after Enabling SPI
```

Set up DMA Interrupt Handler:

```
void __attribute__((__interrupt__)) _DMA1Interrupt(void)
{
   static unsigned int BufferCount = 0;  // Keep record of which buffer
                                         // contains RX Data

   if(BufferCount == 0)
   {
    ProcessRxData(BufferA);              // Process received SPI data in
                                         // DMA RAM Primary buffer
   }
   else
   {
    ProcessRxData(BufferB);              // Process received SPI data in
                                         // DMA RAM Secondary buffer
   }

   BufferCount ^= 1;
   IFS0bits.DMA1IF = 0;                  // Clear the DMA1 Interrupt Flag
}
```

22.7 STARTING DMA TRANSFER

Before DMA transfers can begin, the DMA channel must be enabled by setting the CHEN bit to '1' in the DMAxCON register. When the DMA channel is active, it can be reinitialized by disabling this channel (CHEN = 0), followed by re-enabling it (CHEN = 1). This process resets the DMA transfer count to zero and sets the active DMA buffer to the primary buffer.

When the DMA channel and peripheral are properly initialized, the DMA transfer starts as soon as the peripheral is ready to move data and issues a DMA request. However, some peripherals may not issue a DMA request (and therefore will not start the DMA transfer) until certain conditions exist. In these cases, a combination of different DMA modes and procedures may need to be applied to initiate the DMA transfer:

22.7.1 Starting DMA with the Serial Peripheral Interface (SPI)

Starting the DMA transfer to/from the SPI peripheral depends upon SPI data direction and Slave or Master mode:

- **TX only in Master mode –** In this configuration, no DMA request is issued until the first block of SPI data is sent. To initiate DMA transfers, the user application must first send data using the DMA Manual Transfer mode, or it must first write data into the SPI buffer (SPIxBUF) independently of the DMA.
- **RX only in Master mode –** In this configuration, no DMA request is issued until the first block of SPI data is received. However, in Master mode, no data is received until SPI transmits first. To initiate DMA transfers, the user application must use DMA Null Data Write mode, and start DMA Manual Transfer mode.
- **RX and TX in Master mode –** In this configuration, no DMA request is issued until the first block of SPI data is received. However, in Master mode, no data is received until the SPI transmits it. To initiate DMA transfers, the user application must first send data using the DMA Manual Transfer mode, or it must first write data into the SPI buffer (SPIxBUF) independently of the DMA.
- **TX only in Slave mode –** In this configuration, no DMA request is issued until the first block of SPI data is received. To initiate DMA transfers, the user application must first send data using the DMA Manual Transfer mode, or it must first write data into the SPI buffer (SPIxBUF) independently of the DMA.
- **RX only in Slave mode –** This configuration generates a DMA request as soon as the first SPI data has arrived, so no special steps need to be taken by the user to initiate DMA transfer.
- **RX and TX in Slave mode –** In this configuration, no DMA request is issued until the first SPI data block is received. To initiate DMA transfers, the user application must first send data utilizing the DMA Manual Transfer mode, or it must first write data into the SPI buffer (SPIxBUF) independently of the DMA.

22.7.2 Starting DMA with the Data Converter Interface (DCI)

Unlike other serial peripherals, the DCI starts transmitting as soon as it is enabled (assuming it is the Master). It constantly feeds synchronous frames of data to the external codec to which it is connected. Before enabling the DCI you must:

- Configure the DCI as described in **22.5.2 "Peripheral Configuration Setup"**
- If connected to a stereo codec, use DMA Manual Transfer mode to initiate the first two data transfers.
  - Set the FORCE bit in the DMAxREQ register to transfer the DCI left channel sample.
  - Set the FORCE bit for the second time to transfer the DCI right channel sample.

After these steps are completed, enable the DCI peripheral (see Example 22-11).

22.7.3 Starting DMA with the UART

The UART receiver issues a DMA request as soon as data is received. No special steps need to be taken by the user application to initiate DMA transfer. The UART transmitter issues a DMA request as soon as the UART and transmitter are enabled. This means that the DMA channel and buffers must be initialized and enabled before the UART and transmitter.

Ensure that the UART is configured as described in **22.5.2 “Peripheral Configuration Setup”** (Table 22-2).

Alternatively, the UART and UART transmitter can be enabled before the DMA channel is enabled. In this case, the UART transmitter DMA request will be lost, and the user application must issue a DMA request to start DMA transfers by setting the FORCE bit in the DMAxREQ register.

22.8 DMA CHANNEL ARBITRATION AND OVERRUNS

Each DMA channel has a fixed priority. Channel 0 is the highest, and Channel 7 is the lowest. When a DMA transfer is requested by the source, the request is latched by the associated DMA channel. The DMA controller acts as an arbitrator. If no other transfer is underway or pending, the controller grants bus resources to the requesting DMA channel. The DMA controller ensures that the no other DMA channel is granted any resource until the current DMA channel completes its operation.

If multiple DMA requests arrive or are pending, the priority logic within the DMA controller grants resources to the highest priority DMA channel for completing its operation. All other DMA requests remain pending until the selected DMA transfer is complete. If another DMA request arrives while the current DMA transfer is underway, it is also prioritized with any pending DMA requests, ensuring that the highest priority request is always serviced after the current DMA transfer has completed.

As the DMA channels are prioritized, it is possible that a DMA request will not be immediately serviced and will become pending. The request will remain pending until all higher priority channels have been serviced. A data overrun occurs, If another interrupt arrives before the DMA controller has cleared the original DMA request, and the interrupt is the same type as the pending interrupt.

A data overrun is defined as the condition where new data has arrived in a peripheral data buffer before the DMA could move the prior data. Some DMA-ready peripherals can detect data overruns and issue a CPU interrupt (if the corresponding peripheral error interrupt is enabled), as shown in Table 22-5.

Table 22-5: Overrun Handling by DMA-Ready Peripherals

DMA-Ready Peripheral	Data Overrun Handling
Serial Peripheral Interface (SPI)	Data waiting to be moved by the DMA channel is not overwritten by additional incoming data. Subsequent incoming data is lost and the SPI Receive Overflow (SPIROV) bit is set in the SPI Status (SPIxSTAT) register. Also the SPIx Fault interrupt is generated if the SPI Error Interrupt Enable (SPIxEIE) bit is set in the Interrupt Enable Control (IECx) register in the interrupt controller.
UART	Data waiting to be moved by the DMA channel is not overwritten by additional incoming data. Subsequent incoming data is lost and the Overflow Error (OERR) bit is set in the UART Status (UxSTA) register. Also, the UARTx Error interrupt is generated if the UART Error Interrupt Enable (UxEIE) bit is set in the Interrupt Enable Control (IECx) register in the interrupt controller.
Data Converter Interface (DCI)	Data waiting to be moved by the DMA channel is overwritten by additional incoming data and the Receive Overflow (ROV) bit is set in the DCI Status (DCISTAT) register. Also the DCI Fault interrupt is generated if the DCI Error Interrupt Enable (DCIEIE) bit is set in the Interrupt Enable Control (IEC0) register in the interrupt controller.
10-bit/12-bit Analog-to-Digital Converter (ADC)	Data waiting to be moved by the DMA channel is overwritten by additional incoming data. The overrun condition is not detected by the ADC.
Other DMA-Ready Peripherals	No data overrun can occur.

Data overruns are only detectable in hardware when the DMA is moving data from a peripheral to DPSRAM. DMA data transfers from DPSRAM to a peripheral (based on, for example, a buffer empty interrupt) will always execute. Any consequential DPSRAM data overruns must be detected using software. The duplicate DMA request is ignored and the pending request remains pending. As usual, the DMA channel clears the DMA request when the transfer is eventually completed. If the CPU does not intervene in the meantime, the data transferred will be the latest (overrun) data, and the prior data will be lost.

The user application can handle an overrun error in different ways, depending on the nature of the data source. Data recovery and resynchronization of the DMAC with its data source/sink is a task that is highly application dependent. For streaming data, such as that from a CODEC (via the DCI peripheral), the application can ignore the lost data. After fixing the source of the problem (if possible), the DMA interrupt handler should attempt to resynchronize the DMAC and DCI so that data is again buffered correctly. The user application should react fast enough to prevent any further overruns occurring.

By the time the peripheral overrun interrupt is entered, the pending DMA request will have already moved the overrun data value to the address where the lost data should have gone. That data can be moved to its correct address, and a null data value inserted into the missing data slot. The DPSRAM address of the channel can then be adjusted accordingly. Subsequent DMA requests to the faulted channel then initiate transfers as normal to the corrected DPSRAM address. For applications where the data cannot be lost, the peripheral overrun interrupt will need to abort the current block transfer, reinitialize the DMA channel and request a data resend before it is lost.

22.9 DEBUGGING SUPPORT

To improve user visibility into DMA operation during debugging, the DMA controller includes several status registers that can provide information on which DMA channel executed last (LSTCH<3:0> bits in the DMACS1 register), which DPSRAM address offset it was accessing (DSADR<15:0> bits in the DSADR register) and from which buffer (PPSTx bits in the DMACS1 register).

22.10 DATA WRITE COLLISIONS

The CPU and DMA channel may simultaneously read or read/write to any DPSRAM or DMA-ready peripheral data register. The only constraint is that the CPU and DMA channel should not simultaneously write to the same address. Under normal circumstances, this situation should never arise. However, if for some reason it does, then it will be detected and flagged, and a DMA Fault trap will be initiated. The CPU write will also be allowed to take priority, though that is mainly to provide predictable behavior and is otherwise of little practical consequence.

It is also permissible for the DMA channel to write to a location during the same bus cycle that the CPU is reading it, and vice versa. However, it should be noted that the resultant reads are of the old data, not the data written during that bus cycle. Also note that this situation is considered normal operation and does not result in any special action being taken.

In the event of a simultaneous write to the same DPSRAM address by the CPU and DMA channel, the XWCOLx bit is set in the DMA Controller Status 0 (DMACS0) register. In the event of a simultaneous write to the same peripheral address by the CPU and DMA channel, the PWCOLx bit is set in the DMA Controller Status 0 (DMACS0) register. All collision status flags are logically ORed together to generate a common DMAC Fault trap. The XWCOLx and PWCOLx flags are automatically cleared when the user application clears the DMAC Error Status bit (DMACERR) in the Interrupt Controller (INTCON1) register.

Subsequent DMA requests to a channel that has a write collision error are ignored while the XWCOLx or PWCOLx remain set.

Under write collision conditions, either XWCOLx or PWCOLx could be set due to write collision, but not both. Setting both flags is used as a unique means to flag a rare manual trigger event error without adding more Status bits (see **22.6.10 “Manual Transfer Mode”**).

Example 22-13 illustrates DMA controller trap handling with DMA Channel 0 transferring data from the DPSRAM to the peripheral (UART), and DMA Channel 1 transferring data from the peripheral (ADC) to the DPSRAM.

Example 22-13: DMA Controller Trap Handling:

```
void __attribute__((__interrupt__)) _DMACError(void)
{
    static unsigned int ErrorLocation;

    // Peripheral Write Collision Error Location
    if(DMACS0 & 0x0100)
    {
        ErrorLocation = DMA0STA;
    }

    // DMA RAM Write Collision Error Location
    if(DMACS0 & 0x0002)
    {
        ErrorLocation = DMA1STA;
    }

    DMACS0 = 0;                          //Clear Write Collision Flag
    INTCON1bits.DMACERR = 0;             //Clear Trap Flag
}
```

22.11 OPERATION IN POWER-SAVING MODES

22.11.1 Sleep Mode

The DMA is disabled during the Sleep power-saving mode. Prior to entering Sleep mode, it is recommended (though not essential) that all DMA channels either be allowed to complete the block transfer that is currently underway, or be disabled.

22.11.2 Idle Mode

The DMA is a second bus master within the system and can, therefore, continue to transfer data when the CPU has entered the Idle power-saving mode. Provided the peripheral being serviced by the DMA channel is configured for operation during Idle mode, data may be transferred to and from the peripheral and DPSRAM. When the block transfer is complete, the DMA channel issues an interrupt (if enabled) and wakes up the CPU. The CPU then runs the interrupt service handler.

Each peripheral includes a Stop in Idle control bit. When set, this control bit disables the peripheral while the CPU is in Idle mode. If the DMAC is being used to transfer data in and/or out of the peripheral, engaging the Stop in Idle feature within the peripheral will, in effect, also disable the DMA channel associated with the peripheral.

22.12 DESIGN TIPS

22.12.1 Interfacing DMA with DCI

When the DCI has multiple audio channels per frame (for example, left and right channels from a stereo codec), all samples are sequentially interleaved and transferred by the DMA channel as shown in Figure 22-21.

Typically, however, the user application prefers the data on a per-channel basis, which means that the program must provide an additional algorithm to either sort the transferred data, or index it by jumping unwanted data. Either approach requires additional code and results in more execution time.

The DCI does not support Peripheral Indirect Addressing. However, it is still possible to assemble the data by audio channel with a special DCI configuration combined with two DMA channels. When the Buffer Length control bits (BLEN<1:0>) in the DCI Control 2 (DCICON2) register are set to '01' (instead of '00'), and two DMA Channels are used to transfer received data from the DCI to DPSRAM, the received audio data is sorted by channel. In this case, when the DCI generates a DMA request it goes to both DMA channels every time two words are buffered (one right sample and one left sample). When the DMA request occurs, one DMA channel transfers data from the DCI RX Buffer 0 (RXBUF0) register, while the other DMA channel transfers data from the DCI RX Buffer 1 (RXBUF1) register. In effect, the transferred data is sorted by audio channel, as shown in Figure 22-22.

For this example to work, DMA Channel 1 initializes the DMA1PAD register to the RXBUF1 address (instead of the RXBUF0 address as stated in **22.5.1 "DMA Channel to Peripheral Association Setup"**).

Figure 22-21: Typical Data Transfer for the DCI

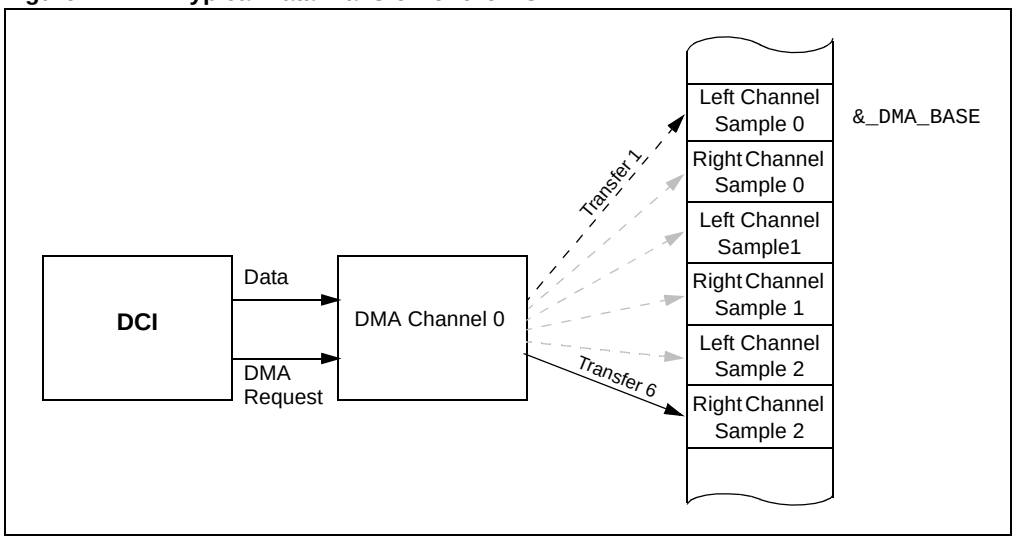
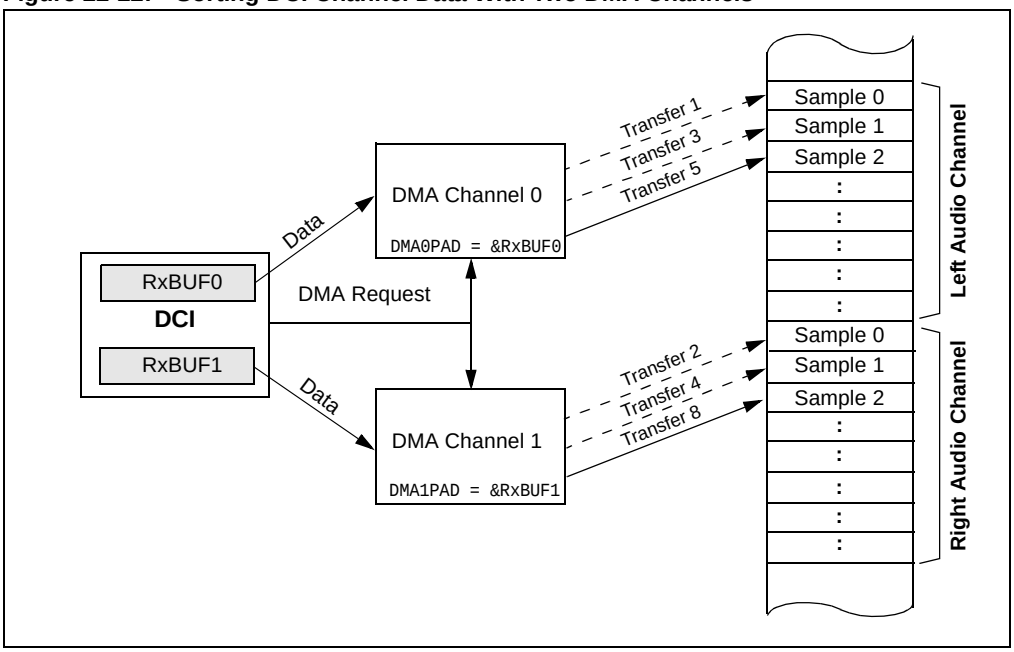


Figure 22-22: Sorting DCI Channel Data With Two DMA Channels



22.13 REGISTER MAPS

Table 22-6: DMA Register Map

File Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
DMA0CON	0380	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA0REQ	0382	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA0STA	0384	STA<15:0>																0000
DMA0STB	0386	STB<15:0>																0000
DMA0PAD	0388	PAD<15:0>																0000
DMA0CNT	038A	—	—	—	—	—	—	CNT<9:0>										0000
DMA1CON	038C	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA1REQ	038E	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA1STA	0390	STA<15:0>																0000
DMA1STB	0392	STB<15:0>																0000
DMA1PAD	0394	PAD<15:0>																0000
DMA1CNT	0396	—	—	—	—	—	—	CNT<9:0>										0000
DMA2CON	0398	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA2REQ	039A	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA2STA	039C	STA<15:0>																0000
DMA2STB	039E	STB<15:0>																0000
DMA2PAD	03A0	PAD<15:0>																0000
DMA2CNT	03A2	—	—	—	—	—	—	CNT<9:0>										0000
DMA3CON	03A4	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA3REQ	03A6	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA3STA	03A8	STA<15:0>																0000
DMA3STB	03AA	STB<15:0>																0000
DMA3PAD	03AC	PAD<15:0>																0000
DMA3CNT	03AE	—	—	—	—	—	—	CNT<9:0>										0000
DMA4CON	03B0	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA4REQ	03B2	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA4STA	03B4	STA<15:0>																0000
DMA4STB	03B6	STB<15:0>																0000
DMA4PAD	03B8	PAD<15:0>																0000
DMA4CNT	03BA	—	—	—	—	—	—	CNT<9:0>										0000
DMA5CON	03BC	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA5REQ	03BE	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000

Legend: — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

Table 22-6: DMA Register Map (Continued)

File Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
DMA5STA	03C0	STA<15:0>																0000
DMA5STB	03C2	STB<15:0>																0000
DMA5PAD	03C4	PAD<15:0>																0000
DMA5CNT	03C6	—	—	—	—	—	—	CNT<9:0>										0000
DMA6CON	03C8	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA6REQ	03CA	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA6STA	03CC	STA<15:0>																0000
DMA6STB	03CE	STB<15:0>																0000
DMA6PAD	03D0	PAD<15:0>																0000
DMA6CNT	03D2	—	—	—	—	—	—	CNT<9:0>										0000
DMA7CON	03D4	CHEN	SIZE	DIR	HALF	NULLW	—	—	—	—	—	AMODE<1:0>		—	—	MODE<1:0>		0000
DMA7REQ	03D6	FORCE	—	—	—	—	—	—	—	—	IRQSEL<6:0>							0000
DMA7STA	03D8	STA<15:0>																0000
DMA7STB	03DA	STB<15:0>																0000
DMA7PAD	03DC	PAD<15:0>																0000
DMA7CNT	03DE	—	—	—	—	—	—	CNT<9:0>										0000
DMACS0	03E0	PWCOL7	PWCOL6	PWCOL5	PWCOL4	PWCOL3	PWCOL2	PWCOL1	PWCOL0	XWCOL7	XWCOL6	XWCOL5	XWCOL4	XWCOL3	XWCOL2	XWCOL1	XWCOL0	0000
DMACS1	03E2	—	—	—	—	LSTCH<3:0>				PPST7	PPST6	PPST5	PPST4	PPST3	PPST2	PPST1	PPST0	0000
DSADR	03E4	DSADR<15:0>																0000
INTCON1	0080	NSTDIS	—	—	—	—	—	—	—	—	—	DMACERR	—	—	—	—	—	0000
IFS0	0084	—	DMA1IF	—	—	—	—	—	—	—	—	—	DMA0IF	—	—	—	—	0000
IFS1	0086	—	—	—	—	—	—	—	DMA2IF	—	—	—	—	—	—	—	—	0000
IFS2	0088	—	DMA4IF	—	—	—	—	—	—	—	—	—	DMA3IF	—	—	—	—	0000
IFS3	008A	—	—	DMA5IF	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IFS4	008C	—	—	—	—	—	—	—	—	—	—	DMA7IF	DMA6IF	—	—	—	—	0000
IEC0	0094	—	DMA1IE	—	—	—	—	—	—	—	—	—	DMA0IE	—	—	—	—	0000
IEC1	0096	—	—	—	—	—	—	—	DMA2IE	—	—	—	—	—	—	—	—	0000
IEC2	0098	—	DMA4IE	—	—	—	—	—	—	—	—	—	DMA3IE	—	—	—	—	0000
IEC3	009A	—	—	DMA5IE	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IEC4	009C	—	—	—	—	—	—	—	—	—	—	DMA7IE	DMA6IE	—	—	—	—	0000
IPC1	00A6	—	—	—	—	—	—	—	—	—	—	—	—	—	DMA0IP<2:0>			4444
IPC3	00AA	—	—	—	—	—	DMA1IP<2:0>			—	—	—	—	—	—	—	—	4444
IPC6	00B0	—	—	—	—	—	—	—	—	—	—	—	—	—	DMA2IP<2:0>			4444
IPC9	00B6	—	—	—	—	—	—	—	—	—	—	—	—	—	DMA3IP<2:0>			4444

Legend: — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

Table 22-6: DMA Register Map (Continued)

File Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
IPC11	00BA	—	—	—	—	—	DMA4IP<2:0>			—	—	—	—	—	—	—	—	4444
IPC15	00C2	—	—	—	—	—	—	—	—	—	DMA5IP<2:0>			—	—	—	—	4444
IPC17	00C6	—	—	—	—	—	—	—	—	—	DMA7IP<2:0>			—	DMA6IP<2:0>			4444

Legend: — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

22.14 RELATED APPLICATION NOTES

This section lists application notes that are related to the use of Direct Memory Access. These application notes may not be written specifically for the dsPIC33F Product Family, but the concepts are pertinent and could be used with modification and possible limitations. The current application notes related to the DMA module include:

Title	Application Note #
No related application notes at this time.	

Note: Please visit the Microchip web site (www.microchip.com) for additional Application Notes and code examples for the dsPIC33F family of devices.

22.15 REVISION HISTORY

Revision A (December 2006)

This is the initial release of this document.

Revision B (July 2008)

This revision incorporates the following content updates:

- Figures:
  - Data Transfer from ADC with Register Indirect Addressing (see Figure 22-11).
  - Data Transfer from ADC with Peripheral Indirect Addressing (see Figure 22-12).
- Additional minor corrections such as language and formatting updates are incorporated in the entire document.