TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

DMA(Directmemory access)

2018.07.06 ~ 2018.07.17

강사 : Innova Lee(이상훈)

강사 메일 : gcccompil3r@gmail.com

학생 : 문지희

학생 메일 : mjh8127@naver.com

목차

1. DMA 의 뜻

DMA(Direct Access Memory)

: 주변장치들이 메모리에 직접 접근하여 읽거나 쓸 수 있도록 하는 기능으로, 장치들 사이의 데이터가 CPU 를 거치지 않는다.

↔ PIO(Programmed Input/Output)

: 주변 기기와 CPU 사이에서 데이터를 주고 받는 방식. PIO 방식은 많은양의 데이터를 주고받게되면 CPU 가 처리해야할 태스크의 양이 많아지게 되어 효율성이 떨어진다는 단점이 있다. PIO 방식의 단점을 극복하기 위해 DMA 와 Interrupt 를 고안하였다.

DMA 의 장점

: 장치 컨트롤러가 데이터의 한 블록을 이동시키는데 CPU 의 개입이 필요 없어 많은 양의 데이터를 이동시킬 때 부담이 없다. CPU 에서는 데이터 이동이 완료되면 단 한번의 인터럽트를 발생하여 데이터가 전송되는 동안 다른 작업을 수행할 수 있게 되어 효율성이 높아지는 장점이 있다.

DMA 의 주요 특징

- CPU 에 독립된 데이터 전송

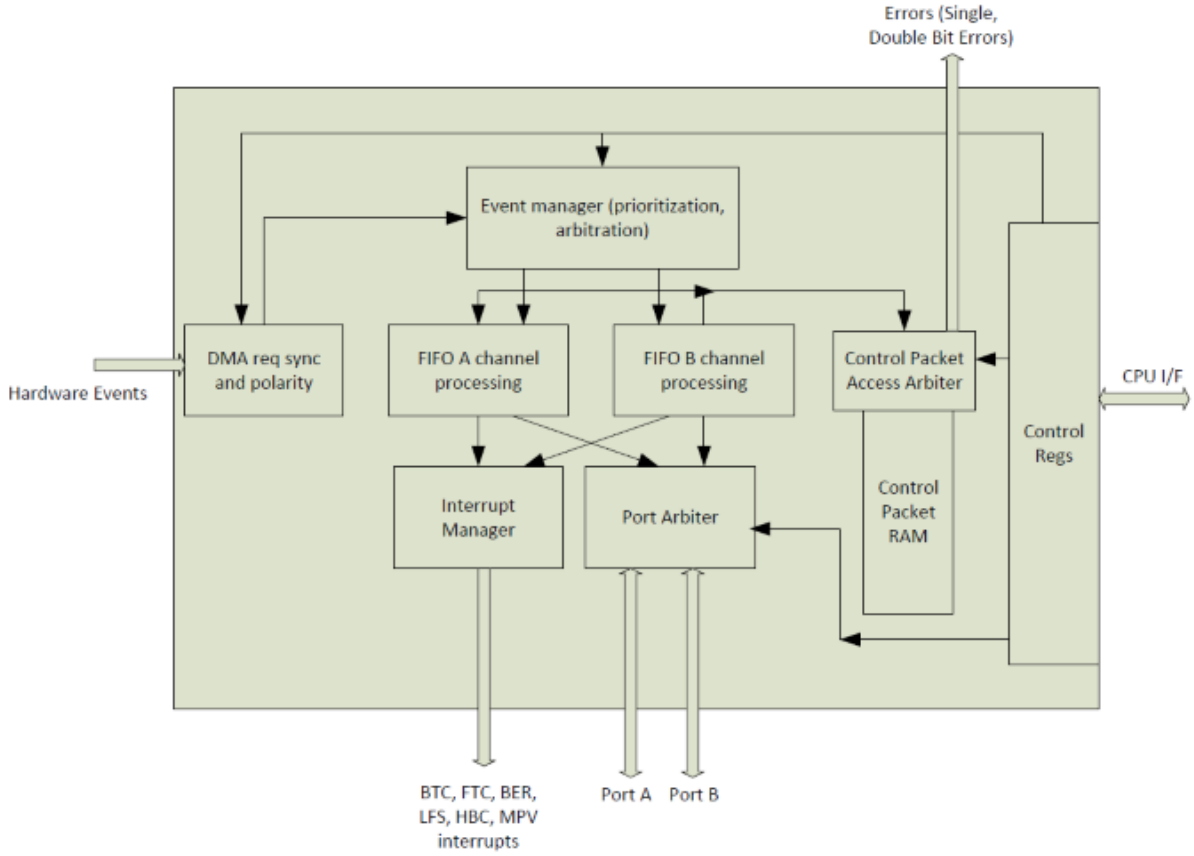
- 최대 2 개의 채널에 동시 전송 지원(시스템 처리량의 효율이 높아진다). 각 채널은 FIFO 중 하나에 데이터를 저장하는 동안 읽기 쓰기 엑세스를 위해 Port A 또는 Port B 또는 둘 다 사용할 수 있도록 구성할 수 있음.

- FIFO(First Input First Output)의 버퍼가 각 4 개의 엔트리, 64 비트의 너비를 가져 최대 32 바이트의 데이터를 수용 가능하다.

- 2 개의 마스터 포트가 있고 각각의 포트는 64 비트의 너비를 가진다.

2. DMA Block Diagram

Figure 20-1. DMA Block Diagram



Port Arbiter

Port A 나 Port B 를 통해 DMA 데이터의 읽기, 쓰기를 한다. 두 Port 모두 64 비트의 너비를 가진다.

두 포트는 어느 장치에서 어느 장치로 데이터를 전송하느냐에 따라 사용이 달라진다. 아래 Table20-1 을 참조하여 **PARx** 레지스터(Section)를 설정해야한다. PAR 레지스터는 PAR0 ~ PAR3 까지 4 개의 레지스터가 존재하는데, 선택한 채널에 따라 사용하는 레지스터가 달라진다.

DMA Ports	System Resources
Port A	L2 Flash L2 SRAM EMIF
Port B	All peripherals, that is, MibSPI registers, DCAN registers All peripherals memories, that is, MibSPI RAM, DCAN RAM

Example 1

: L2 Flash · L2 SRAM · EMIF → 주변 장치 레지스터 · 주변 장치 메모리

PARx 레지스터의 각 채널에 0 x1(Port A read, Port B write)을 쓴다

Example 2

: 주변 장치 레지스터 · 주변 장치 메모리 → L2 Flash · L2 SRAM · EMIF

PARx 레지스터의 각 채널에 0 x0(Port A write, Port B read)을 쓴다.

Example 3

: L2 Flash → L2 SRAM

PARx 레지스터의 각 채널에 0 x2(Port A)을 쓴다.

Example 4

: 주변 장치에서 → 다른 주변 장치

PARx 레지스터의 각 채널에 0 x3(Port B)을 쓴다.

Interrupt Manager

DMA 를 사용하게되면 데이터를 전송하고 CPU 에 인터럽트를 보내게 되는데 인터럽트의 종류는 데이터 블록의 어느 시점에서 인터럽트를 보내는 지에 따라 나뉜다.

Figure 20-2. Example of a DMA Transfer Using Frame Trigger Source

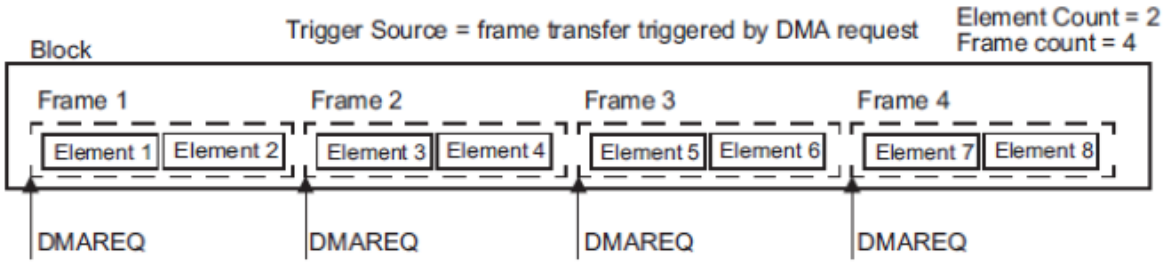
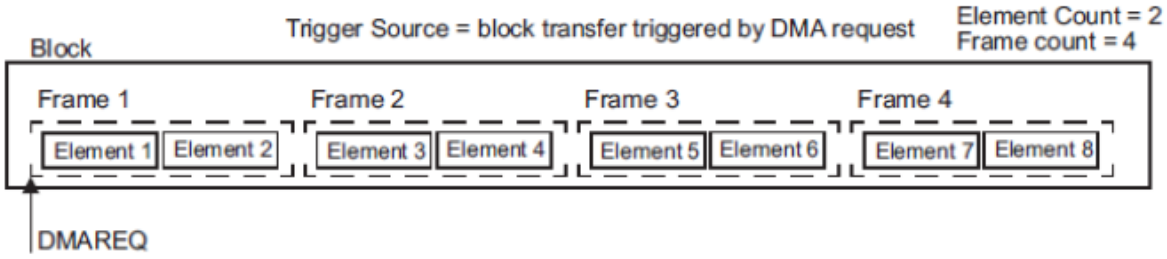


Figure 20-3. Example of a DMA Transfer Using Block Trigger Source



DMA 컨트롤러는 세 가지의 데이터를 참조한다.

Element : 프로그래밍 된 데이터 타입에 따라 8bit , 16bit, 32bit 값을 가짐. Source(read) , destination(write)에 대해 개별적으로 선택할 수 있음. element 전송은 중단될 수 없다.

Frame : 하나 이상의 element 을 하나의 유닛으로 전송된다. frame 전송은 element 전송 사이에 전송이 중단될 수 있음..

Block : 하나 이상의 frame 이 하나의 유닛으로 전송된다. 각 채널은 하나의 데이터 블록을 전송할 수 있다.

인터럽트의 종류는 아래와 같다.

• **Frame transfer complete (FTC)** : 프레임의 마지막 요소 이후에 인터럽트가 발생합니다. 양도되었습니다.

• **Last frame transfer started (LFS)** : 마지막 인터럽트의 첫 번째 요소 앞에 인터럽트가 발생합니다. 블록 전송의 프레임이 시작되었습니다.

• **First half of block complete (HBC)** : 블록의 절반 이상이 인터럽트 인 경우 양도. - 프레임 수 n 이 홀수 인 경우, HBC 인터럽트는 프레임의 끝에서 생성됩니다. (n + 1) / 2 프레임의 수가 블록에 남아 있습니다. - 프레임 수 n 이 짝수이면, HBC 인터럽트는 프레임의 끝에서 생성됩니다 n / 2 개의 프레임 수가 블록에 남아 있습니다.

• **Block transfer complete (BTC)** : 마지막 프레임의 마지막 요소 다음에 인터럽트가 발생합니다 이전 되었습니다.

• **External imprecise error on read** : 버스 오류 (인터럽트와의 잘못된 거래)가 발생하면 인터럽트가 발생할 수 있습니다. ok 응답)이 감지됩니다. 부정확 한 읽기 오류는 ESM 모듈에 연결됩니다.

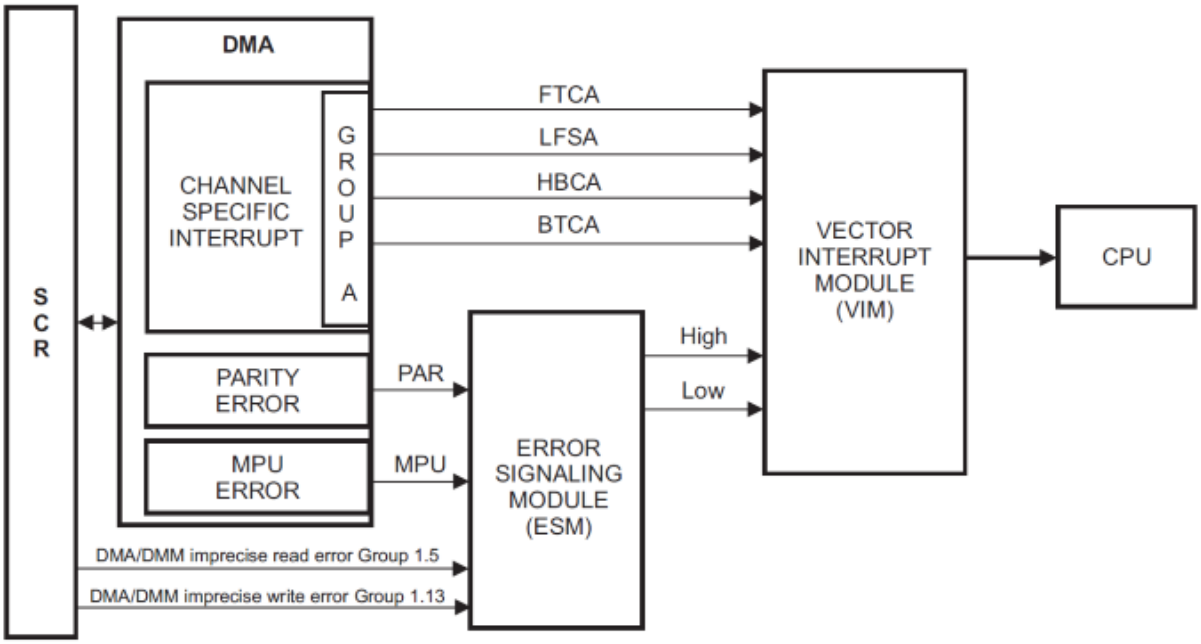
• **External imprecise error on write** : 버스 오류 (인터럽트와의 불법 거래)가 발생하면 인터럽트가 발생할 수 있습니다. ok 응답)이 감지됩니다. 부정확 한 쓰기 오류는 ESM 모듈에 연결됩니다.

• **Memory Protection Unit error (MPU)** : DMA 가 액세스가 저하 된 것을 감지하면 인터럽트가 발생합니다. DMA 의 MPU 레지스터에 프로그램 된 메모리 영역 외부. MPU 인터럽트 ESM 모듈에 연결됩니다.

• **Parity error (PAR)** : DMA 가 패리티 오류를 감지 할 때 인터럽트가 발생합니다. 제어 패킷. PAR 인터럽트는 ESM 모듈에 연결됩니다.

DMA 는 제어 패킷 처리, 패리티 인터럽트 및 메모리 보호를 위해 5 개의 인터럽트 라인을 출력한다.

Figure 20-14. DMA Interrupts



DMA 모듈의 각 채널 특정 인터럽트는 두 개의 서로 다른 CPU 를 개별적으로 지원하기 위해 그룹 A 또는 B 로 라우팅된다.

Group A - 인터럽트 (FTC, LFS, HBC 및 BTC)는 ARM CPU 로 라우팅된다.

Group B - 인터럽트 (FTC, LFS, HBC 및 BTC)는 라우팅되지 않는다.

사용자 소프트웨어는 Group A 인터럽트 만 구성해야한다.

FIFO buffer

DMA FIFO 는 4 레벨 깊이와 64 비트 폭의 너비를 가진다. (최대 4 × 64 비트의 데이터 = 32 바이트를 저장할 수 있음). 데이터 packing 및 unpacking 에 사용된다. DMA FIFO 에는 두 가지 상태가 있다.

- EMPTY : FIFO 에 데이터가 없다.
- FULL : FIFO 가 채워지거나 요소 수가 0 에 도달했다. 판독 동작은 정지되어야한다.

FIFO 가 비어 있을 때만 DMA 채널을 전환 할 수 있다(FIFO 가 비어 있을 때 채널 간의 중재가 수행됨). DMA 에는 2 개의 FIFO, FIFO A, FIFO B 가 있으며, 각 FIFO 는 동시에 최대 2 개의 채널을 실행할 수 있는 기능을 제공한다.

FIFO 버퍼는 포트 제어 레지스터의 우회 기능이 존재한다. (자세한 내용은 Section 20.3.1.51 참조)

packing & unpacking

DMA 컨트롤러는 read element size 와 write element size 가 다를 때 필요한 데이터 패킹 및 언 패킹을 자동으로 수행한다. read element size 와 write element size 보다 작으면 데이터 packing 을 요구한다. read element size 와 write element size 보다 클 경우 데이터 unpacking 이 필요하다. read element size 와 write element size 가 같을 때, 읽기 도중 패킹이 수행되지 않으며 쓰기 도중 언팩이 수행되지 않는다.

Event Manager(prioritization, arbitration)

Arbitration - 우선순위가 더 높은 채널을 서비스하거나 해당 채널이 사용 중지 된 경우 채널이 Arbitration 되었다고 한다.

Arbitration Boundary - 최대 32 바이트를 전송 완료 했을 때 마다 Arbitration Boundary 에 도달했다고 한다. FIFO 는 Arbitration Boundary 에서 비어있음. DMA 는 Arbitration Boundary 를 활용하여 채널의 우선순위를 다시 지정한다. Arbitration Boundary 내에서 전송이 중단 될 수 없다.

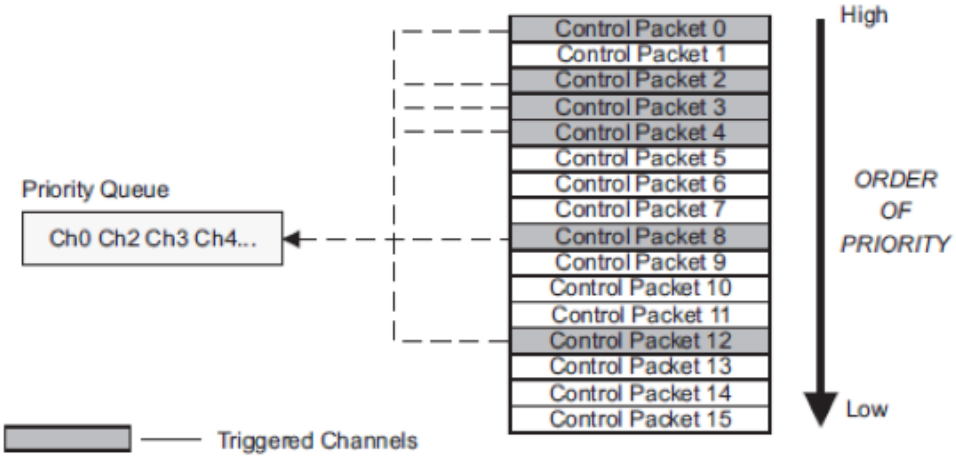
유저는 우선순위 큐(Priority Queue)에 arbitration 한 다음 채널을 할당하여 요청을 처리할 수 있다.

포트에는 두가지 우선순위 큐가 있는데 high priority queue, low priority queue 가 있는데, 각 큐는 fixed priority 또는 rotating priority 를 따르도록 구성할 수 있다.

Fixed priority 는 채널 번호가 낮을수록 우선순위가 높아진다. (Figure 20-9)

Rotating priority 는 **라운드 로빈 방식**을 기반으로 한다. 처음에는 우선순위 목록이 고정 된 우선순위 체계에 따라 정렬된다. 우선순위가 낮은 대기열의 채널이 서비스 되기 전에, 우선순위가 높은 대기열에 할당된 채널은 항상 우선순위 구성표에 따라 우선 처리된다. Table 20-2 는 다른 우선순위 방식에 따라 어떻게 arbitration 이 수행되는지 보여준다.

Figure 20-9. Fixed Priority Scheme

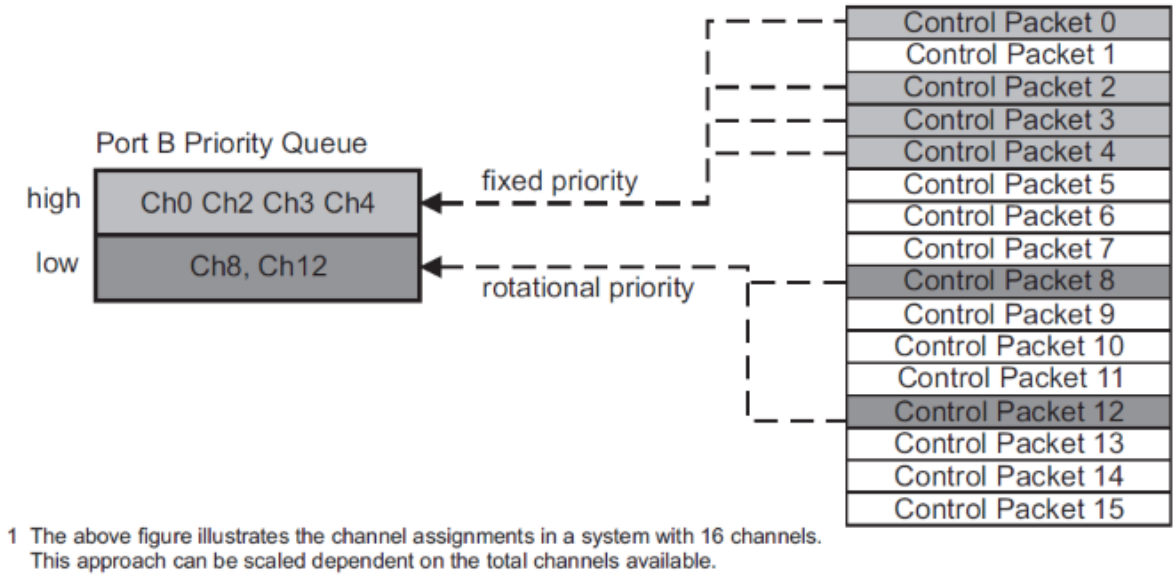


The above figure illustrates that by default Lower the channel number, higher the Priority.

Queue	Priority Scheme	Remark
High priority	Fixed	채널은 채널 번호에 따라 오름차순으로 서비스된다. **채널번호가 낮을수록 우선순위가 높아진다**. 보류중인 채널이 있을 때마다 채널이 arbitration 된다. 우선순위가 더 높은 보류중인 채널이 없다면 **transfer count 가 0** 이 될 때까지 채널은 전부 서비스되고, 다음으로 우선순위가 채널이 서비스된다. 대기열에 대기중인 채널이 없는 경우 DMA 는 낮은 대기열(Low Queue) 채널을 서비스하기 위해 전환한다.
	Rotating	**채널은 라운드 로빈 방식**을 사용하여 중재됩니다. FIFO 가 비어 있을 때 arbitration 이 수행된다. 대기열에 대기중인 채널이 없는 경우 DMA 는 낮은 대기열(Low Queue) 채널을 서비스하기 위해 전환한다.
Low priority	Fixed	채널은 채널 번호에 따라 오름차순으로 서비스됩니다. **채널 번호가 낮을수록 우선순위가 높아진다**. 더 높은 우선 순위의 보류중인 채널이 있을 때마다 채널이 arbitration된다. 보류중인 채널이 없다면 **transfer count가 0** 이 될 때까지 채널이 서비스되고, 그 다음으로 보류중인 채널 중에서 우선순위가 높은 채널이 서비스된다. DMA 가 Low Queue 채널을 서비스 하는 동안 High Queue 에 보류중인 채널이 있다면 DMA 는 **arbitration bountary 이후**에 High Queue 채널을 서비스하게 된다.
	Rotating	채널은 **라운드 로빈 방식**을 사용하여 중재 됩니다. FIFO 가 비어 있을 때 arbitration 이 수행된다.

Table 20-2

Figure 20-11. Example Channel Assignments



1 The above figure illustrates the channel assignments in a system with 16 channels. This approach can be scaled dependent on the total channels available.

Figure 20-11 처럼 최적의 시스템 성능을 위해, **우선 순위가 높은 채널은 fixed priority, 우선순위가 낮은 채널은 rotational priority 방식으로 사용**한다.

DMA 는 active high 및 active low 하드웨어 요청을 모두 지원한다. 이것은 DMAREQPS1 및 DMAREQPS0 레지스터를 통해 구성됩니다. request polarity 의 선택은 프로그램 시작시에 이루어 져야함. 채널을 따라 요청 극성을 active high 에서 active low 로 변경하려면 다음과 같이 해야 한다.

1. HWCHENA 비트를 사용하여 극성을 변경할 채널을 비활성화함.

2. 요청 라인을 비활성 하이 상태로 설정할 수 있도록 주변 장치를 비활성화한다(기본적으로 요청이 활성 높음).

3. GCTRL 레지스터를 사용하여 소프트웨어 리셋을 DMA 에 적용한다.

4. 채널에 대한 요청 극성을 프로그래밍한다.

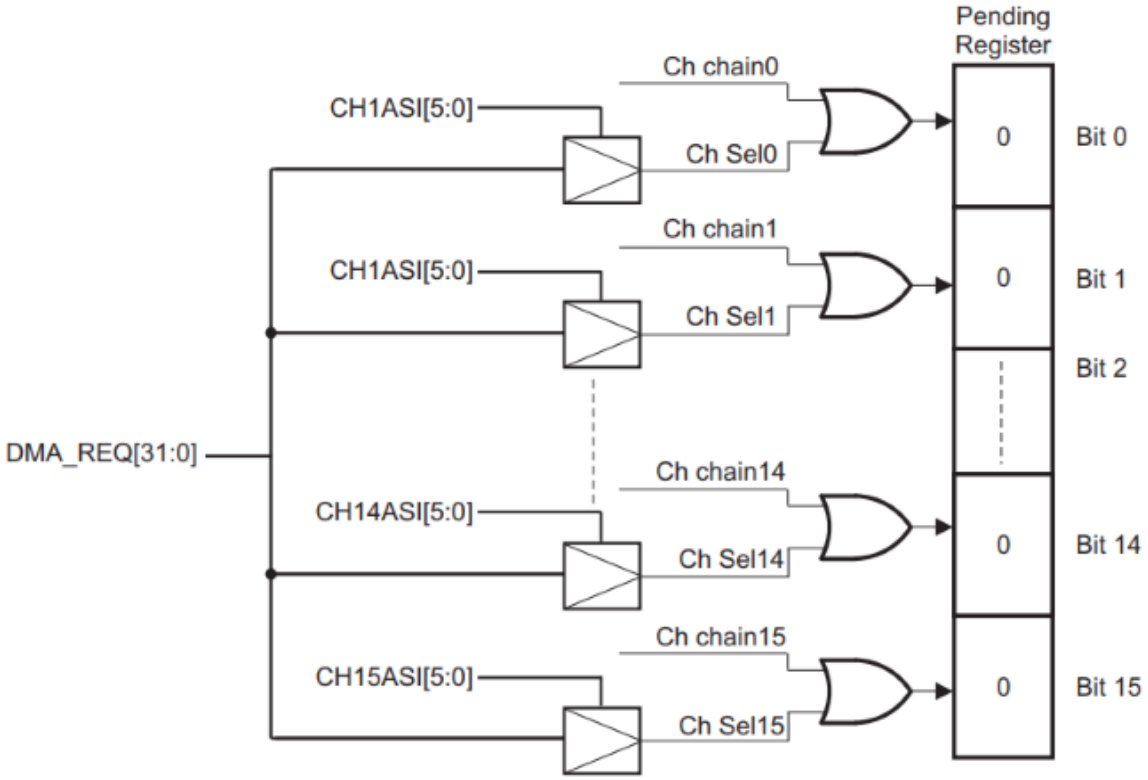
5. DMA 채널을 다시 활성화한다.

6. DMA 이벤트를 트리거하는 주변 장치를 다시 활성화한다.

channel chaining 이라는 외부 DMA 요청없이 단일 또는 다중 채널을 트리거하는 데 사용되는 것이 있는데, 하나의 제어 패킷을 다른 패킷과 연결하여 가능하다. 채널 제어 레지스터 (섹션 20.3.2.4)의 체인 [5 : 0] 필드는 체인 제어 패킷을 프로그래밍하는 데 사용된다. 연결 제어 패킷은 보류중인 레지스터 내에서 조정 규칙을 따릅니다. 예를 들어, CH1, CH2, CH4, CH5 가 함께 트리거되고 CH3 이 CH1 로 연결되면 체인 연결에도 불구하고 서비스되는 채널의 순서는 CH1 -> CH2 -> CH3 -> CH4 -> CH5 입니다. 채널 체인 기능을 설정하려면 첫 번째 DMA 요청을 트리거하기 전에 모든 채널에 대해 채널 제어 레지스터 (20.3.2.4 절)를 활성화 해야 한다.

그림 20-16 은 내부 전송 요청이 필요한 전송을 완료 한 후 생성되어 보류중인 레지스터에 저장되는 방법을 보여준다. 이 예시에서 CH1 은 CH0 에 연결됩니다. CH0 이 트리거되면 CH1 은 트리거되지 않은 경우에도 채널 보류 레지스터 (20.3.1.2 절)에서 보류 중으로 캡처 된다.

Figure 20-16. Example of Channel Chaining



Control Packet Access Arbiter & Control Packet RAM

* 패킷(packet) : 정보 기술에서 패킷 방식의 컴퓨터 네트워크가 전달하는 데이터의 형식화된 블록이다. 패킷을 지원하지 않는 컴퓨터 통신 연결은 단순히 바이트, 문자열, 비트를 독립적으로 연속하여 데이터를 전송한다. 데이터가 패킷으로 형식이 바뀔 때, 네트워크는 장문 메시지를 더 효과적이고 신뢰성 있게 보낼 수 있다. 패킷은 데이터의 한 단위라고 할 수 있다.

제어 패킷(Control packet)은 각 채널에 대한 소스 주소, 대상 주소, 전송 횟수 및 제어 속성과 같은 전송정보를 저장.

각 로직채널에 상응하는 것은 고정된 순서대로 매핑된 control packet 이다. 예를들어 control packet0

은 채널 0 에 대한 채널 정보를 저장한다. DMA 요청은 각 독립된 채널(Section 20.2.7 에 설명)에 매핑된다.

DMA 요청과 채널 사이의 매핑 규칙은 Figure 20-4 에 나와있다.

각 제어 패킷은 9 개의 필드를 포함한다. 처음 6 개의 필드는 기본적인 제어패킷을 구성하고 DMA 설정동안 프로그래밍 가능하다. 마지막 세 필드는 작업 제어 패킷을 구성하며 CPU 에서만 읽을 수 있다

Figure 20-4. DMA Request Mapping and Control Packet Organization

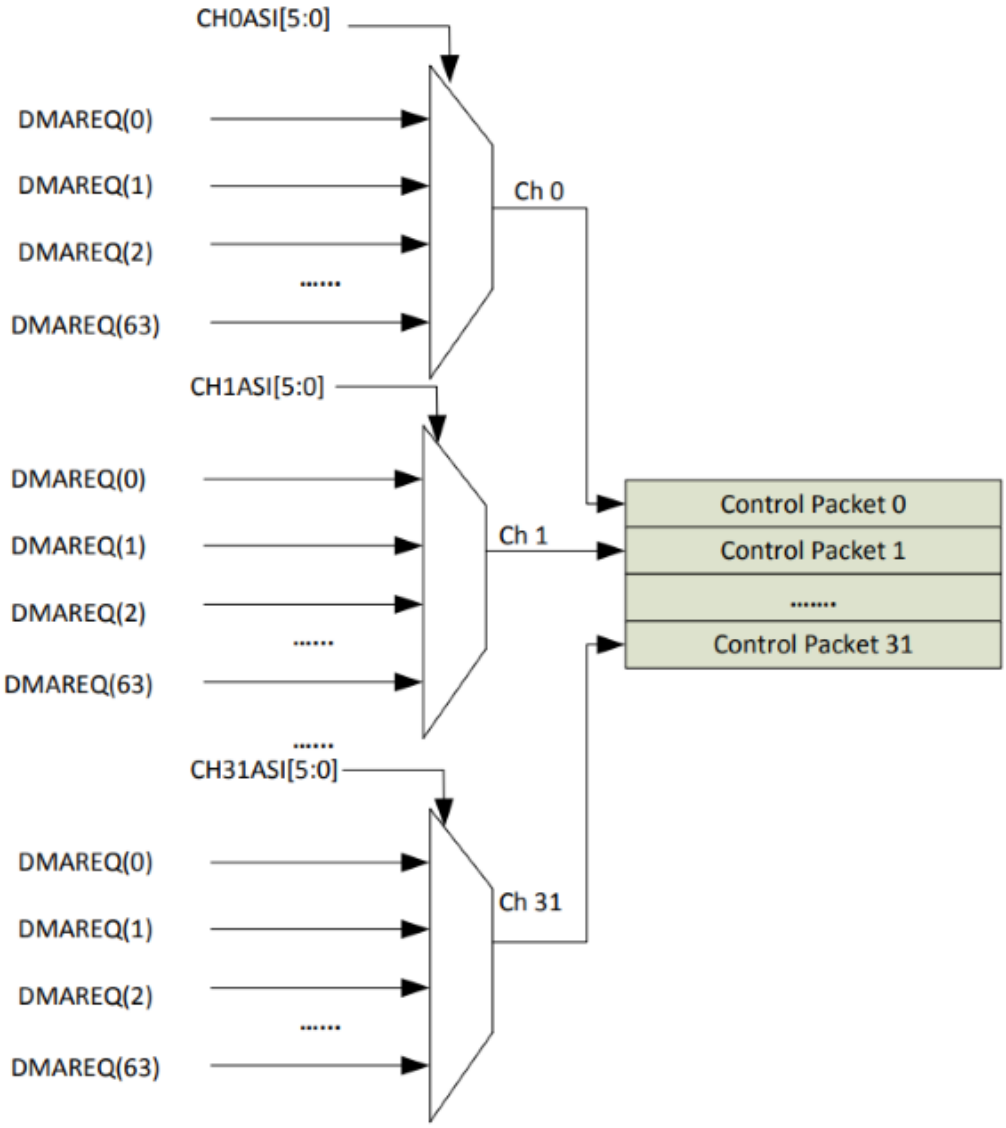


Figure 20-5 는 제어패킷의 구성을 보여준다. 기본 제어 패킷에는 소스 주소, 대상 주소, 전송 횟수, element/frame 의 오프셋 값 및 채널 구성과 같은 채널 정보가 들어있다.

소스 주소, 대상 주소 및 전송 횟수에도 각각의 작업이미지가 있다. 작업 이미지의 세 필드는 작동 제어 패킷을 구성하며 쓰기 액세스에서 CPU 가 액세스 할 수 없다.

처음에 DMA 채널이 트랜잭션을 위해 선택되면 다음과 같은 과정이 이루어진다.

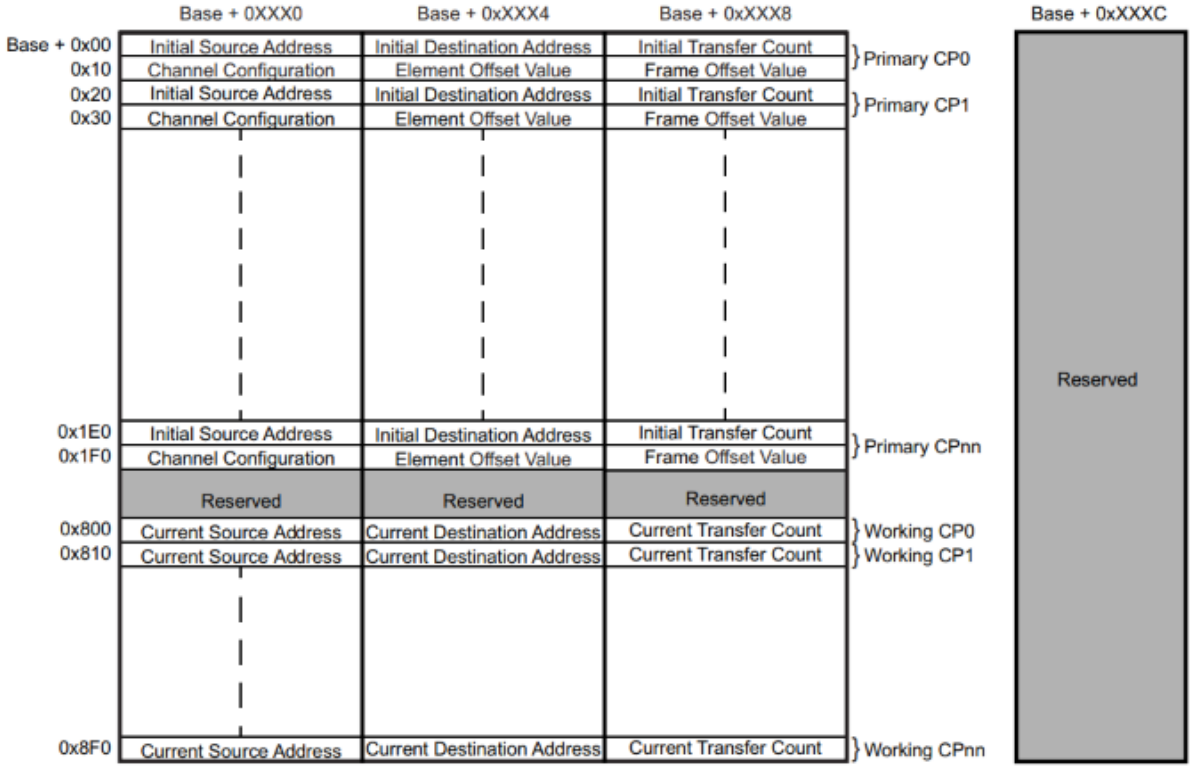
1. DMA 상태 머신이 주 제어 패킷(Primary control packet)을 읽음.

2. 채널이 arbitrate 되면 현재 source 주소, destination 주소, 전송 횟수가 각각의 작업 이미지로 복사된다.

3. 채널이 DMA 에 의해 다시 서비스되면, 상태머신은 전체 블록 전송이 끝날 때 까지 DMA 트랜잭션을 계속하기 위해 주 제어 패킷(primary control packet)과 작업 제어 패킷(working control packet)을 읽는다.

동일한 채널이 다시 요청되면, 상태 시스템은 주 제어 패킷만을 읽고 위의 과정을 반복한다.

Figure 20-5. Control Packet Organization and Memory Map



Initial Source Address : DMA 전송의 destination 주소 32bit 를 저장.

Initial Source Address : DMA 전송의 source 주소 32bit 를 저장.

Initial Transfer Count : 전송 횟수 필드는 Frame 전송 카운트 값과 element 전송 카운트 값으로 구성된다. 각 카운트 값은 13 비트 폭을 가진다. 하나의 블록을 전송하여 최대 512MB 의 데이터를 전송할 수 있다. element 와 frame 수는 source 데이터 구조에 따라 프로그래밍 된다.

총 전송 사이즈 계산 :

The total transfer size is calculated as:

$$T_{sz} = E_{rsz} \bullet E_{tc} \bullet F_{tc}$$

where

T_{sz} = Total Transfer Size

E_{rsz} = Read Element Size

E_{tc} = Element Transfer Count

F_{tc} = Frame Transfer Count

Element count 나 frame count 가 0 일 경우는 총 전송횟수가 0 으로 간주한다. 0 으로 설정된 카운터는 DNA 트랜잭션이 시작되지 않음.

Elemnet/Frame Offset Value :

RAM 과 주소 레지스터에 다른 타입의 구조체로 생성할 수 있는 4 개의 offset 값이 있다.

an element offset value for source and destination

a frame offset value for source and destination

source 및/또는 destination 에 대한 element 오프셋 값은, 각 element 가 source 및/또는 destination 전송된 후에 추가될 오프셋을 정의한다.

source 및/또는 destination 에 대한 frame 오프셋 값은, element 카운트가 0 이 된 후에 source 및/또는 destination 주소에 추가 될 오프셋을 정의한다.

Current Source Address : 현재 source 주소 필드는 DMA 트랜잭션 동안 현재 작업 source 주소를 포함한다. indexing mode/addressing mode 가 증가 한 이후에 현재 소스 주소가 증가한다.

Current destination Address : 현재 destination 주소 필드는 DMA 트랜잭션 동안 현재 작업 destination 주소를 포함한다.

Current Transfer Count : 현재 전송 카운트는 블록에 보내질 element 의 남아있는 수를 저장한다. Source location 에서 element 를 읽었을 때 하나씩 감소된다.

*index : 데이터를 기록할 경우 그 데이터의 이름, 데이터 크기 등의 속성과 그 기록 장소 등을 표로 표시하는 것. 즉 참조용의 데이터를 색인표 또는 인덱스라 한다.

*indexing : 데이터를 찾아내기 위한 색인을 지정하는 것. 이것은 메모리 내의 표나 직접 액세스 기억 장치(direct access store) 내의 파일로부터 데이터를 검색하기 위하여 사용되는 방법이다.

ECC

제어 패킷 RAM 은 Single Error Correction Double Error Detection (SECDED) 체계를 사용하여 보호된다. 이 방식은 DMA 제어 패킷 RAM 에 저장된 128 비트의 데이터마다 총 9 개의 ECC 검사 비트를 사용하여 구현됨. ECC 제어 레지스터(Section 20.3.1.62)에서 ECC 기능을 활성화, 비활성화 할 수 있다.

ECC 검사를 하게 되면 두 가지의 오류가 발생할 수 있다.

1. 단일 비트 오류 : CPU 또는 DMA 논리에 의해 제어 패킷에 대한 읽기 중에 단일 비트 오류가 발생하고 DMASECCCTRL 레지스터의 EDCAMODE [3 : 0]이 0xA 이면 오류가 자동으로 수정됨. 레지스터의 SBEFLG 비트도 1 로 설정되어 단일 비트 오류가 수정되었음을 나타낸다. DMAECCSBE 레지스터는 오류 주소를 나타내도록 업데이트하고, DMASECCCTRL 레지스터의 SBE_EVT_EN [3 : 0]이 0xA 이면 오류가 ESM 에도 표시된다.

2. 이중 비트 오류 : CPU 또는 DMA 논리에 의해 제어 패킷에 대한 읽기 중에 이중 비트 오류가 발생하고 DMAPECR 레지스터의 ECC_ENA [3 : 0]이 0xA 이면 오류가 ESM 에 표시된다. EDFLG 비트가 설정되고 오류 주소가 DMAPAR 레지스터에 저장된다.

Control Regs

DMA Control Register는 Table 20-7 에 요약되어 있다. Control Register의 기본 주소는 FFFF F000h 이다. Control packet은 Table 20-8 에 요약되어 있고 control packet 의 기본 주소는 FFF8 0000h 이다.

DMA 컨트롤러는 프로그램 메모리와 데이터 메모리를 구분하지 않는다. DMA 컨트롤러는 프로그래밍을 통해 물리적 주소 맵 내의 모든 공간으로 4Gbyte 만큼을을 전송할 수 있다. DMA 컨트롤러는 제어 패킷에서 source 및 destination 에 대한 물리주소를 프로그래밍 함으로써, 4Gbyte 의 물리 주소맵의 어느 공간으로든 전송할 수 있다.

3. 함수정리

void dmaEnable(void);

GCTRL 레지스터를 조정하여 DMA 를 활성화 시킬 것인지와 debug 모드를 조정할 수 있다.

debug 의 4 가지 모드
1. Ignore suspend.
2. Finish current block transfer.
3. Finish current frame transfer.
4. Immediately stop at an DMA arbitration boundary and continue after suspend.

void dmaDisable(void);

GCTRL 레지스터를 조정하여 데이터 전송 중이 아닐 때 DMA 를 비활성화 시킴.

void dmaSetCtrlPacket(dmaChannel_t channel, g_dmaCTRL g_dmaCTRLPKT);

Control packet 을 설정하는 함수이다. g_dmaCTRLPKT 의 매개변수로 dmaSetCtrlPacket 내의 Control Packet Memory Map 의 primary Control Packet 0 를 채워주게된다.
또한 channel 에 해당하는 PARx 레지스터에서 Port A, Port B 의 사용을 결정한다.

- dmaChannel_t channel : DMA_CH0 ~ DMA_CH31 까지의 32 개의 채널이 존재한다. 0~31 의 채널 중 하나를 선택.

- g_dmaCTRL g_dmaCTRLPKT : g_dmaCTRLPKT 는 아래의 구조체를 포함한다.

```
typedef struct  dmaCTRLPKT
{
    uint32 SADD;       /* Initial source address           */
    uint32 DADD;       /* Initial destination address      */
    uint32 CHCTRL;     /* Next channel to be triggered + 1 */
    uint32 FRCNT;      /* Frame   count                    */
    uint32 ELCNT;      /* Element count                    */
    uint32 ELDOFFSET;  /* Element destination offset       */
    uint32 ELSOFFSET;  /* Element source offset            */
    uint32 FRDOFFSET;  /* Frame destination offset         */
    uint32 FRSOFFSET;  /* Frame source offset              */
    uint32 PORTASGN;   /* DMA port                         */
    uint32 RDSIZE;     /* Read element size                */
    uint32 WRSIZE;     /* Write element size               */
    uint32 TTYPE;      /* Trigger type - frame/block       */
    uint32 ADDMODERD;  /* Addressing mode for source       */
    uint32 ADDMODEWR;  /* Addressing mode for destination  */
    uint32 AUTOINIT;   /* Auto-init mode                   */
} g_dmaCTRL;
```

void dmaSetChEnable(dmaChannel_t channel, dmaTriggerType_t type);

DMA trigger type 은 HW 와 SW 가 있는데 전송 타입을 정하고, 매개변수로 설정한 타입의 0~31 사이의 채널을 활성화시킨다.

DMA Request 에는 3 가지의 DMA 전송 방법이 존재한다.
1. **Software request** : SW 채널 활성화 레지스터와 SW 채널 상태 레지스터에 기록함으로써 데이터가 전송된다.(Section 20.3.1.7) 소프트웨어 요청은 채널 제어 레지스터의 TTYPE 비트 설정에 따라 블록 또는 프레임 전송을 트리거 할 수 있다. (Section 20.3.2.4).)
2. **Hardware request** : DMA 컨트롤러는 최대 48 개의 DMA 요청라인을 처리한다. 하드웨어 요청은 채널 제어 레지스터의 TTYPE 비트 설정에 따라 프레임 또는 블록 전송을 트리거 할 수 있다.(Section 20.3.2.4)
3. **Triggered by other control packet** : control packet 이 프로그래밍 된 번호의 전송을 완료하면 다른 채널을 트리거하여 전송을 시작한다.

dmaChannel_t channel : 0~31 까지의 32 개의 채널 중 하나를 선택

dmaTriggerType_t type : type 이 DMA_HW 이면 HWCHENAS 레지스터를 조정하고, type 이 DMA_SW 이면 SWCHENAS 를 조정한다.

void dmaReqAssign(dmaChannel_t channel, dmaRequest_t reqline);

channel 에 대한 DMA 요청할당을 선택한다.

dmaChannel_t channel : 0~31 까지의 32 개의 채널 중 하나를 선택

dmaRequest_t reqline : DMA_REQ0 ~ DMA_REQ47 까지의 48 개의 request line 을 선택한다. 사용하는 모듈에 따라 사용되는 request line 이 다른데 이는 Table 20-3 을 참조하면 된다.

Ex) SCI3 를 사용 할 때 : receive - DMAREQ[30], transmit - DMAREQ[31]

void dmaSetPriority(dmaChannel_t channel, dmaPriorityQueue_t priority);

채널의 우선순위를 설정하는 함수이다. reqline 은 LOWPRIORITY, HIGHPRIORITY 두 종류가 있는데 LOWPRIORITY 이면 LOW Queue 에 채널의 우선순위가 설정되고, HIGHPRIORITY 이면 HIGH Queue 에 채널의 우선순위가 설정된다.

reqline 이 High Queue 인지 Low Queue 인지에 따라 활성화하는 레지스터가 다른데
Low Priority 일 때에는 CHPRIOR 레지스터의 channel 에 해당하는 비트를 설정하고
High Priority 일 때에는 CHPRIOS 레지스터의 channel 에 해당하는 비트를 설정한다.

dmaChannel_t channel : 0~31 까지의 32 개의 채널 중 하나를 선택

dmaPriorityQueue_t priority : priority 에는 LOWPRIORITY, HIGHPRIORITY 두가지가 존재한다. LOW Queue 를 사용할 것인지 HIGH Queue 를 사용 할 것인지 선택.

void dmaEnableInterrupt(dmaChannel_t channel, dmaInterrupt_t inttype, dmaIntGroup_t group);

GCHIENAS 레지스터에서 해당 channel 의 비트를 조정하여 인터럽트를 활성화한다.

dmaChannel_t channel : 0~31 까지의 32 개의 채널 중 하나를 선택
dmaInterrupt_t inttype : inttype 에는

FTC (Frame transfer complete Interrupt),
LFS (Last frame transferstarted Interrupt),
HBC (First half of block complete Interrupt),
BTC (Block transfer complete Interrupt)

이 4 가지가 존재한다.
FTC 일 때에는 FTCINTENAS 레지스터와 FTCMAP 레지스터를 조정 하게되고,
LFS 는 LFSINTENAS 레지스터와 LFSMAP 레지스터를,
HBC 는 HBCINTENAS 레지스터와 HBCMAP 레지스터를,
BTC 는 BTCINTENAS 레지스터와 BTCMAP 레지스터를 조정한다.

dmaIntGroup_t group : dmaIntGroup_t 에는 DMA_INTA, DMA_INTB 가 존재한다.

DMA 모듈의 각 채널 특정 인터럽트는 2 개의 CPU 를 지원하기 위해 Group A 와 Group B 로 라우팅된다.

Group A - 인터럽트(FTC, LFS, HBC, BTC)는 ARM CPU 로 라우팅된다.

Group B - 인터럽트(FTC, LFS, HBC, BTC)는 라우팅되지 않는다.

사용자 소프트웨어는 Group A 인터럽트만 구성해아함.

*라우팅(routing) : 목적지까지 갈 수 있는 여러 경로 중 한 가지 경로를 설정해 주는 과정

void dmaDisableInterrupt(dmaChannel_t channel, dmaInterrupt_t inttype);

inttype 의 타입에 따라 FTCINTENAR, LFSINTENAR, HBCINTENAR, BTCINTENAR 레지스터를 조정하여 해당 채널의 인터럽트를 비활성화 한다.

void dmaDefineRegion(dmaMPURegion_t region, uint32 start_add, uint32 end_add);

DMA 컨트롤러는 보호 메커니즘을 통해 메모리 영역을 보호하며 해당 주소 범위에 대한 액세스를 제한하여 실수로 DMA 컨트롤러에 의해 액세스 되는 것을 방지할 수 있다.

메모리 보호 메커니즘은 주어진 메모리 영역에 대한 액세스 권한, 영역에 대한 시작 및 끝 주소, 보호된 영역에 대한 액세스 위반 통지로 구성된다.

보호 할 각 영역의 시작 및 끝 주소를 DMAMPRxS 와 DMAMPRxE 에 입력하면 된다.(Section 20.3.1.64) 이렇게 하게 되면 부적절한 액세스로부터 보호 될 수 있다.

액세스 권한의 4 가지 종류
1. Full access
2. Read only access
3. Write only access
4. No access

메모리 보호 위반을 감지하면 위반을 유발 한 DMA 채널이 중지되고 사용 가능한 다음 DMA 채널이 서비스된다.

region 에 해당하는 레지스터에서 시작주소와 끝 주소를 적용한다.

1B0h	DMAMPCTRL1	DMA Memory Protection Control Register 1	Section 20.3.1.64
1B4h	DMAMPST1	DMA Memory Protection Status Register 1	Section 20.3.1.65
1B8h	DMAMPR0S	DMA Memory Protection Region 0 Start Address Register	Section 20.3.1.66
1BCh	DMAMPR0E	DMA Memory Protection Region 0 End Address Register	Section 20.3.1.67
1C0h	DMAMPR1S	DMA Memory Protection Region 1 Start Address Register	Section 20.3.1.68
1C4h	DMAMPR1E	DMA Memory Protection Region 1 End Address Register	Section 20.3.1.69
1C8h	DMAMPR2S	DMA Memory Protection Region 2 Start Address Register	Section 20.3.1.70
1CCh	DMAMPR2E	DMA Memory Protection Region 2 End Address Register	Section 20.3.1.71
1D0h	DMAMPR3S	DMA Memory Protection Region 3 Start Address Register	Section 20.3.1.72
1D4h	DMAMPR3E	DMA Memory Protection Region 3 End Address Register	Section 20.3.1.73
1D8h	DMAMPCTRL	DMA Memory Protection Control Register	Section 20.3.1.74
1DCh	DMAMPST2	DMA Memory Protection Status Register 2	Section 20.3.1.75
1E0h	DMAMPR4S	DMA Memory Protection Region 4 Start Address Register	Section 20.3.1.76
1E4h	DMAMPR4E	DMA Memory Protection Region 4 End Address Register	Section 20.3.1.77
1E8h	DMAMPR5S	DMA Memory Protection Region 5 Start Address Register	Section 20.3.1.78
1ECh	DMAMPR5E	DMA Memory Protection Region 5End Address Register	Section 20.3.1.79
1F0h	DMAMPR6S	DMA Memory Protection Region 6 Start Address Register	Section 20.3.1.80
1F4h	DMAMPR6E	DMA Memory Protection Region 6 End Address Register	Section 20.3.1.81
1F8h	DMAMPR7S	DMA Memory Protection Region 7 Start Address Register	Section 20.3.1.82
1FCh	DMAMPR7E	DMA Memory Protection Region 7 End Address Register	Section 20.3.1.83

dmaMPURegion_t region : DMA_REGION0 ~ DMA_REGION7 까지 존재한다.
uint32 start_add : 시작주소
uint32 end_add : 끝주소

void dmaEnableRegion(dmaMPURegion_t region, dmaRegionAccess_t access, dmaMPUInt_t intenable);

dmaMPURegion_t region : DMA_REGION0 ~ DMA_REGION7

DMA_REGION0 ~ DMA_REGION3 은 DMAMPCTRL1 에서 관리하고,
DMA_REGION4 ~ DMA_REGION7 은 DMAMPCTRL2 에서 관리한다.
DMAMPCTRL1, DMAMPCTRL2 의 비트를 조정하여 영역을 활성화하거나, 영역에 대한 권한을 설정하고, 인터럽트를 활성화 또는 비활성화 한다.

dmaRegionAccess_t access : 권한의 종류에는 4 가지가 있다.
1. FULLACCESS
2. READONLY
3. WRITEONLY
4. NOACCESS

dmaMPUInt_t intenable :

INTERRUPTA_ENABLE : 선택한 영역에 대해 Group A 인터럽트 활성화
INTERRUPTB_ENABLE : 선택한 영역에 대해 Group B 인터럽트 활성화 (잠금단계 장치일 경우 이 옵션을 사용 x)
INTERRUPT_DISABLE : 선택한 영역에 대한 인터럽트 사용하지 않음

void dmaDisableRegion(dmaMPURegion_t region);

region 에 따라 DMAMPCTRL1 이나 DMAMPCTRL2 를 조정하여

void dmaEnableECC(void);
void dmaDisableECC(void);

ECC : Error Correction Code, 작업 수행 중 발생할 수 있는 오류를 검출하고 정정한다.

ECC 기능을 활성화/비활성화 하는 함수이다.
함수 내부를 보면

dmaREG->DMAPCR = 0x5U;

명령어가 있는데 DMAPCR 레지스터는 데이터 시트에 존재하지 않음.

그러므로 ECC 기능을 활성화하는 DMAPECR 레지스터를 조작하도록 수정해야한다.

dmaREG->DMAPECR = 0x0U;

DMAPECR 레지스터에 0x5U 값을 넣으면 ECC check 를 비활성화 하는 것 이므로 0x5 가 아닌 값을 넣어주어야 한다. 대신 DMAPECR 레지스터는 16 비트, 8 비트는 다른 기능을 하므로 피해야 한다.

uint32 dmaGetReq(dmaChannel_t channel);

channel 에 매핑된 request line 의 숫자를 리턴함

boolean dmaIsBusy(void);

GCTRL 레지스터를 보고 데이터를 전송 중인지의 유무를 파악하여 전송 중이라면 FALSE, 전송 중이지 않다면 TRUE 를 반환함.

boolean dmaIsChannelActive(dmaChannel_t channel);

매개변수인 channel 이 활성화 되어있으면 TRUE 를 반환하고 활성화 되어있지 않다면 FALSE 를 반환한다.

boolean **dmaGetInterruptStatus**(dmaChannel_t channel, dmaInterrupt_t inttype);

인터럽트의 타입에는 FTC, LFS, HBC, BTC 가 존재함.
FTC : Frame transfer complete
LFS : Last frame transfer started Interrupt
HBC : First half of block complete Interrupt
BTC : Block transfer complete Interrupt

매개변수 channel 의 인터럽트가 보류 중이 아니라면 0 을 반환하고 인터럽트가 보류 중이라면 1 을 반환한다.

void dmaGroupANotification(dmaInterrupt_t inttype, uint32 channel);

해당 channel 에서 인터럽트가 발생했을 때 취하는 프로그램을 작성하면 된다.

4. 예제 분석

소스코드 파일 위치 : HalCoGen - Help - Examples - TMS570LC43x - example_sci_dma.c

```
#include "HL_sys_common.h"
#include "HL_system.h"
#include "HL_sys_dma.h"
#include "HL_sci.h"
#include "stdio.h"

#define size 100
#define LOOPBACKMODE 1

/* Tx and Rx data buffer */
uint8_t TX_DATA[size], RX_DATA[size] = {0};

/* Addresses of SCI 8-bit TX/Rx data */
//  __little_endian__ 과 LITTLE_ENDIAN__ 이 0으로 정의되어있음.
#if ((__little_endian__ == 1) || (__LITTLE_ENDIAN__ == 1))
#define SCI3_TX_ADDR ((uint32_t)(&(sciREG3->TD)))
#define SCI3_RX_ADDR ((uint32_t)(&(sciREG3->RD)))
#define SCI4_TX_ADDR ((uint32_t)(&(sciREG4->TD)))
#define SCI4_RX_ADDR ((uint32_t)(&(sciREG4->RD)))
#else
#define SCI3_TX_ADDR ((uint32_t)(&(sciREG3->TD))+ 3)
#define SCI3_RX_ADDR ((uint32_t)(&(sciREG3->RD))+ 3)
#define SCI4_TX_ADDR ((uint32_t)(&(sciREG4->TD))+ 3)
#define SCI4_RX_ADDR ((uint32_t)(&(sciREG4->RD))+ 3)
#endif

//  DMA reqest line, Table 20-3 DMA Request Line Connection 참조
#define DMA_SCI3_TX  DMA_REQ31
#define DMA_SCI3_RX  DMA_REQ30
#define DMA_SCI4_TX  DMA_REQ43
#define DMA_SCI4_RX  DMA_REQ42
```

MIBSPI1 / MIBSPI3 / SCI3 / MIBSPI5	MIBSPI1[14] / MIBSPI3[14] / SCI3 receive / MIBSPI5[1](1)	DMAREQ[30]
MIBSPI1 / MIBSPI3 / SCI3 / MIBSPI5	MIBSPI1[15] / MIBSPI3[15] / SCI3 transmit / MIBSPI5[0](2)	DMAREQ[31]
SCI4 / ePWM6 / MIBSPI2 / MIBSPI4 / GIOB	SCI4 receive / ePWM6_SOCA / MIBSPI2[12] / MIBSPI4[12] / GIOB[2]	DMAREQ[42]
SCI4 / ePWM6 / MIBSPI2 / MIBSPI4 / GIOB	SCI4 transmit / ePWM6_SOCB / MIBSPI2[13] / MIBSPI4[13] / GIOB[3]	DMAREQ[43]

```
#define SCI_SET_TX_DMA     (1<<16)
#define SCI_SET_RX_DMA     (1<<17)
#define SCI_SET_RX_DMA_ALL (1<<18)
/* USER CODE END */

uint8   emacAddress[6U] =   {0xFFU, 0xFFU, 0xFFU, 0xFFU, 0xFFU, 0xFFU};
uint32  emacPhyAddress  =   0U;
```

```
void main(void)
{
uint32 sciTxData, sciRxData; //데이터를 받는 곳
    int i;
    g_dmaCTRL g_dmaCTRLPKT1, g_dmaCTRLPKT2;

    /*Load source data*/
    for (i=0; i<size; i++)
    {
        TX_DATA[i] = i;
    }

    /*Initialize SCI*/
    sciInit();

#if LOOPBACKMODE == 1
    /* Enable SCI loopback */
    sciEnableLoopback(sciREG3, Digital_Lbk);
    while (((sciREG3->FLR & SCI_TX_INT) == 0U) || ((sciREG3->FLR & 0x4) == 0x4))
    {
    } /* Wait */

    /*Assign DMA request SCI3 transmit to Channel 0*/
    dmaReqAssign(DMA_CH0, DMA_SCI3_TX); //DMA_REQ31

    /*Assign DMA request SCI3 receive to Channel 1*/
    dmaReqAssign(DMA_CH1, DMA_SCI3_RX);  // DMA_REQ30

    sciTxData = SCI3_TX_ADDR;
    sciRxData = SCI3_RX_ADDR;

#else
    while (((sciREG3->FLR & SCI_TX_INT) == 0U) || ((sciREG3->FLR & 0x4) == 0x4))
    {
    } /* Wait */

    /*Assign DMA request SCI3 transmit to Channel 0*/
    dmaReqAssign(DMA_CH0, DMA_SCI3_TX);

    /*Assign DMA request SCI4 receive to Channel 1*/
    dmaReqAssign(DMA_CH1, DMA_SCI4_RX);

    sciTxData = SCI3_TX_ADDR;
    sciRxData = SCI4_RX_ADDR;

#endif

    /*Configure control packet for Channel 0*/
    // g_dmaCTRL 구조체의 요소들을 정하는 것.
    g_dmaCTRLPKT1.SADD      = (uint32_t)TX_DATA;  //source address
    g_dmaCTRLPKT1.DADD      = sciTxData;          // destination address
    g_dmaCTRLPKT1.CHCTRL    = 0;                   // channel control
    g_dmaCTRLPKT1.FRCNT    = size;                 // frame count
    g_dmaCTRLPKT1.ELCNT     = 1;                  // element count
    g_dmaCTRLPKT1.ELDOFFSET = 0;                   // element destination offset
    g_dmaCTRLPKT1.ELSOFFSET = 0;                   // element destination offset
    g_dmaCTRLPKT1.FRDOFFSET = 0;                   // frame destination offset
    g_dmaCTRLPKT1.FRSOFFSET = 0;                   // frame destination offset
```

```
    g_dmaCTRLPKT1.PORTASGN  = PORTA_READ_PORTB_WRITE;
    g_dmaCTRLPKT1.RDSIZE    = ACCESS_8_BIT;         // read size
    g_dmaCTRLPKT1.WRSIZE    = ACCESS_8_BIT;         // write size
    g_dmaCTRLPKT1.TTYPE     = FRAME_TRANSFER;       // transfer type
    g_dmaCTRLPKT1.ADDMODERD = ADDR_INC1;            // address mode read
    g_dmaCTRLPKT1.ADDMODEWR = ADDR_FIXED;           // address mode write
    g_dmaCTRLPKT1.AUTOINIT  = AUTOINIT_OFF;         // autoinit

    /*Configure control packet for Channel 1*/
    g_dmaCTRLPKT2.SADD      = sciRxData;            //source address
    g_dmaCTRLPKT2.DADD      = (uint32_t)RX_DATA;    //destination address
    g_dmaCTRLPKT2.CHCTRL    = 0;                    //channel control
    g_dmaCTRLPKT2.FRCNT     = size;                 //frame count
    g_dmaCTRLPKT2.ELCNT     = 1;                    //element count
    g_dmaCTRLPKT2.ELDOFFSET = 0;                    //element destination offset
    g_dmaCTRLPKT2.ELSOFFSET = 0;                    // element destination offset
    g_dmaCTRLPKT2.FRDOFFSET = 0;                    // frame destination offset
    g_dmaCTRLPKT2.FRSOFFSET = 0;                    // frame destination offset
    g_dmaCTRLPKT2.PORTASGN  = PORTB_READ_PORTA_WRITE;
    g_dmaCTRLPKT2.RDSIZE    = ACCESS_8_BIT;         // read size
    g_dmaCTRLPKT2.WRSIZE    = ACCESS_8_BIT;         // write size
    g_dmaCTRLPKT2.TTYPE     = FRAME_TRANSFER;       // transfer type
    g_dmaCTRLPKT2.ADDMODERD = ADDR_FIXED;           // address mode read
    g_dmaCTRLPKT2.ADDMODEWR = ADDR_INC1;            // address mode write
    g_dmaCTRLPKT2.AUTOINIT  = AUTOINIT_OFF;         // autoinit

    /*Set control packet for channel 0 and 1*/
    dmaSetCtrlPacket(DMA_CH0, g_dmaCTRLPKT1);
    dmaSetCtrlPacket(DMA_CH1, g_dmaCTRLPKT2);

    /*Set dma channel 0 and 1 to trigger on hardware request*/
    dmaSetChEnable(DMA_CH0, DMA_HW); //전송 타입과 활성화 시킬 채널 세팅
    dmaSetChEnable(DMA_CH1, DMA_HW);

    /*Enable DMA*/
    dmaEnable();

#if LOOPBACKMODE == 1
    /*Enable SCI3 Transmit and Receive DMA Request*/
    sciREG3->SETINT |= SCI_SET_TX_DMA | SCI_SET_RX_DMA | SCI_SET_RX_DMA_ALL;
    //위에서 16비트, 17비트, 18비트가 1로 세팅.  DMA를 receive, transfer

#else
    /*Enable SCI3 Transmit and SCI4 Receive DMA Request*/
    sciREG3->SETINT |= SCI_SET_TX_DMA;
    sciREG4->SETINT |= SCI_SET_RX_DMA | SCI_SET_RX_DMA_ALL;
#endif

    while(dmaGetInterruptStatus(DMA_CH1, BTC) != TRUE); // 보류중인 BTC인터럽트가
없으면 while문은 계속 만족




    for(i=0; i<size; i++)
    {
        if(RX_DATA[i] != TX_DATA[i])
        {
```

```
            break;
        }
    }
    if(i<size)
    {
        printf("Fail\n");
    }
    else
    {
        printf("Pass\n");
    }
    while(1);

}
```