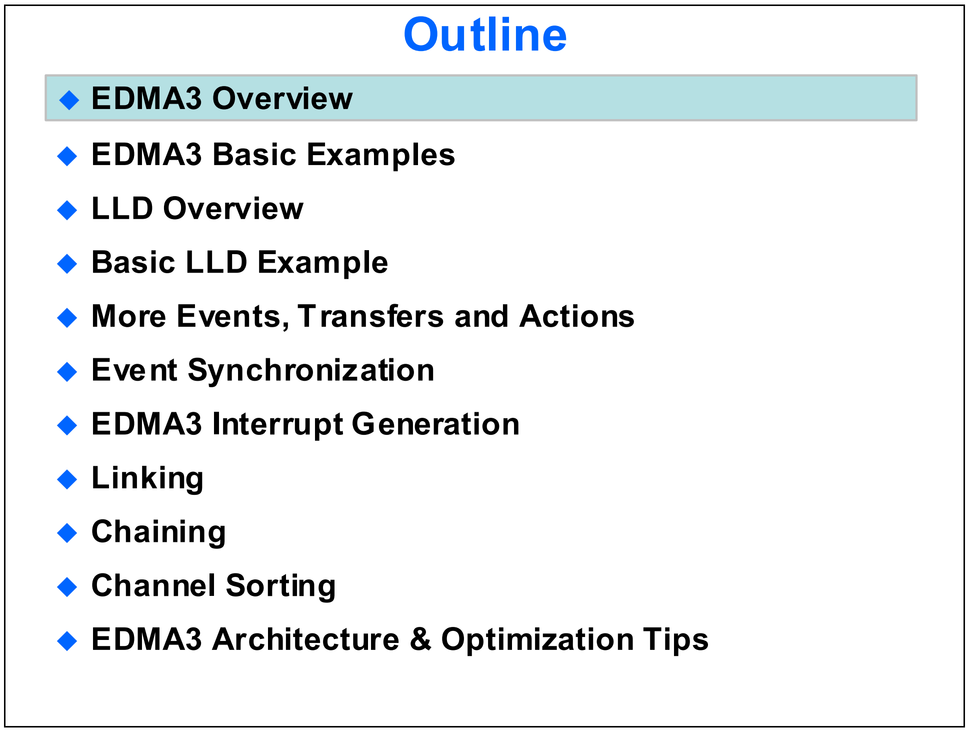
EDMA3 Low Level Driver

Introduction

In this chapter, you will learn the basic concepts of using the EDMA3 (Enhanced DMA version 3) along with the Low Level Driver (LLD) APIs to program various types of transfers. If you are just planning to do block copies of memory without synchronization, we'd recommend you explore the ACPY3 library. However, if you are building an I/O driver (such as a combo of a peripheral like the serial port and EDMA3) that requires synchronization to a peripheral event, then LLD is your best choice.

Objectives

- Provide an introduction to the EDMA3 hardware
- Compare/contrast ACPY3 and LLD
- Analyze LLD examples for basic transfers, interrupt generation, linking, channel sorting, chaining, etc.



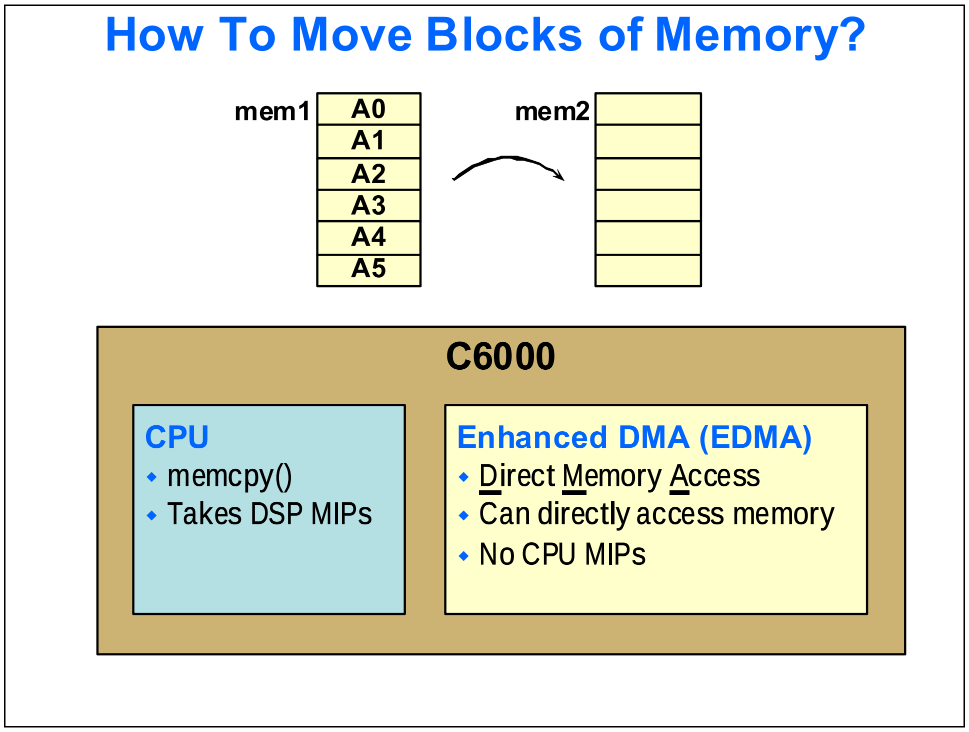
Outline

- **EDMA3 Overview**
- EDMA3 Basic Examples
- LLD Overview
- Basic LLD Example
- More Events, Transfers and Actions
- Event Synchronization
- EDMA3 Interrupt Generation
- Linking
- Chaining
- Channel Sorting
- EDMA3 Architecture & Optimization Tips

Module Topics

EDMA3 Overview

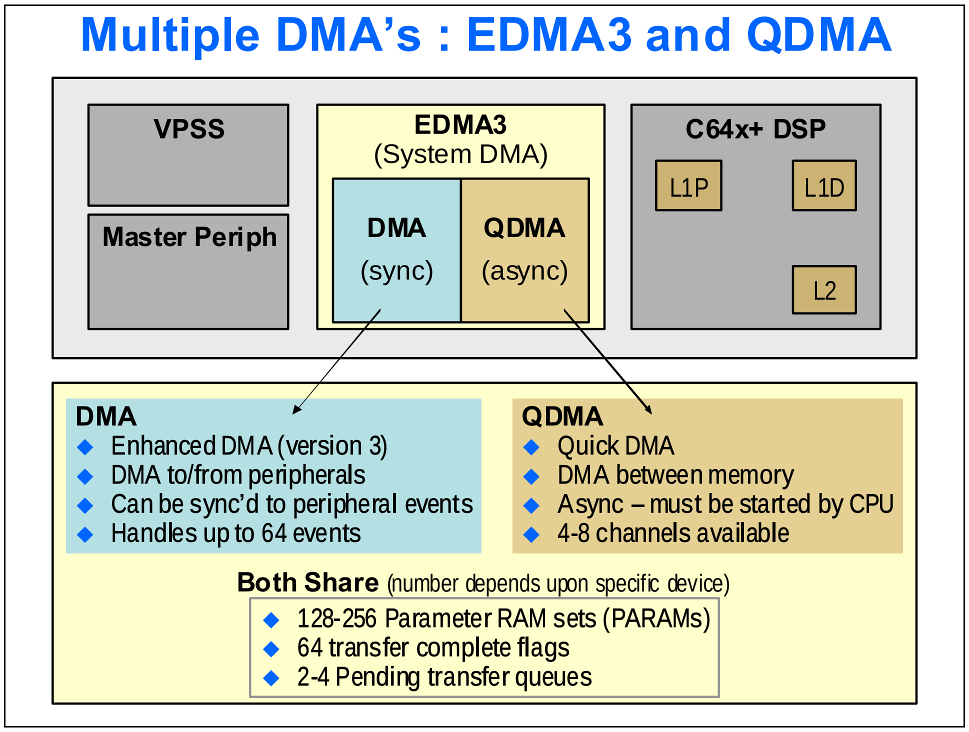
Options for Moving Blocks of Memory



You have several options when you need to move a block of memory. From a high level, you can either choose to use the CPU (like a standard memcpy()), or use the provided EDMA3 peripheral. memcpy() uses a load/store to move the data which can tax your CPU and keep it from doing more important jobs like executing an algorithm.

The EDMA3 peripheral performs this load/store using its own buses and therefore takes no CPU cycles away from your application.

EDMA3 and QDMA


Multiple DMA's : EDMA3 and QDMA
VPSS
Master Periph
EDMA3
(System DMA)
DMA
(sync)
QDMA
(async)
C64x+ DSP
L1P
L1D
L2
DMA
Enhanced DMA (version 3)
DMA to/from peripherals
Can be sync'd to peripheral events
Handles up to 64 events
QDMA
Quick DMA
DMA between memory
Async – must be started by CPU
4-8 channels available
Both Share (number depends upon specific device)
128-256 Parameter RAM sets (PARAMs)
64 transfer complete flags
2-4 Pending transfer queues

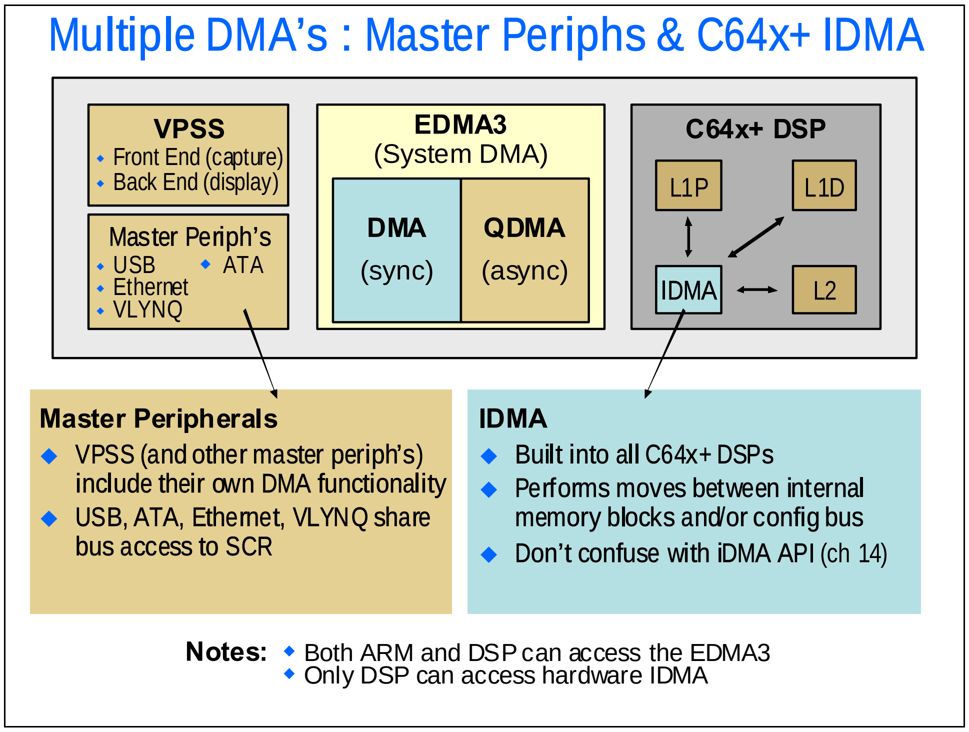
The System DMA (EDMA3) actually consists of two separate DMAs – synchronous DMA (DMA) and the asynchronous DMA (QDMA or Quick DMA). "Quick" refers to the speed at which you can configure the transfer – NOT to the speed of the transfer (a stumbling block for people new to the EDMA3 peripheral). You can perform similar block memory moves with either the DMA or the QDMA – however only the DMA supports options such as synchronizing the transfer to peripheral events (e.g. the serial port receive register having data ready).

ACPY3, as mentioned previously, uses the QDMA to perform its transfers. A transfer using the QDMA uses a trigger word (one of the configuration parameters in the parameter RAM set such as CCNT) to start the transfer – when the parameter "word" is written, it "triggers" the transfer to start. This is all hidden to the user when programming EDMA3 via the ACPY3 APIs (shown later). Again, if all you need to do is simply move a block of memory from SRC to DST, then ACPY3 is probably the method to use – either in an algorithm or your application.

However, when building a driver that contains a peripheral (e.g. the serial port), using the synchronous capability of the DMA via the LLD library is required. Most of this chapter will focus on how to configure and use LLD to program synchronous transfers.

Both DMAs (DMA and QDMA) use EDMA3 resources such as Parameter RAM sets (PARAM or PSETs) which contain the configuration registers (like src, dst, count, etc) and transfer complete codes (TCCs). Check your datasheet to determine the exact number for your device. All of these resources are shared amongst the two DMAs.

All Master Peripherals have their own DMA



Multiple DMA's : Master Periphs & C64x+ IDMA

Master Peripherals

- VPSS (and other master periph's) include their own DMA functionality
- USB, ATA, Ethernet, VLYNQ share bus access to SCR

IDMA

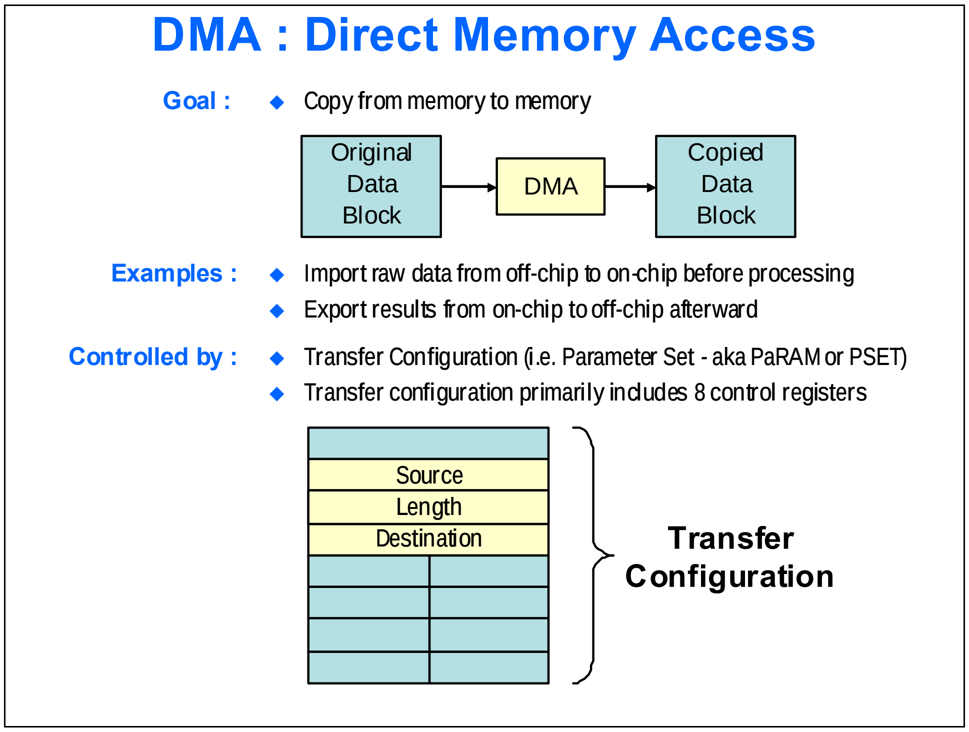
- Built into all C64x+ DSPs
- Performs moves between internal memory blocks and/or config bus
- Don't confuse with iDMA API (ch 14)

Notes:
- Both ARM and DSP can access the EDMA3
- Only DSP can access hardware IDMA

Depending on your specific device, you will have master peripherals such as video ports (VPSS), USB, Ethernet MAC, HPI, etc. All of these peripherals have their own DMA embedded in the peripheral in order to initiate transfers to the buses. Slave peripherals (such as a serial port) do NOT have a DMA built into them.

Another somewhat confusing name is the IDMA. Don't confuse this with the iDMA algorithm interface standard for requesting DMA resources from an algorithm to an application. All CAPS IDMA is the "Internal DMA" located in the C64x megamodule. Its sole purpose is to move blocks of memory between internal memories (L1, L2) and from L1/L2 to the peripheral configuration registers. How to use the IDMA is outside the scope of this chapter – however there is pretty good documentation on it available.

How Does a DMA Work?



DMA : Direct Memory Access

Goal :
- Copy from memory to memory

Original Data Block
DMA
Copied Data Block

Examples :
- Import raw data from off-chip to on-chip before processing
- Export results from on-chip to off-chip afterward

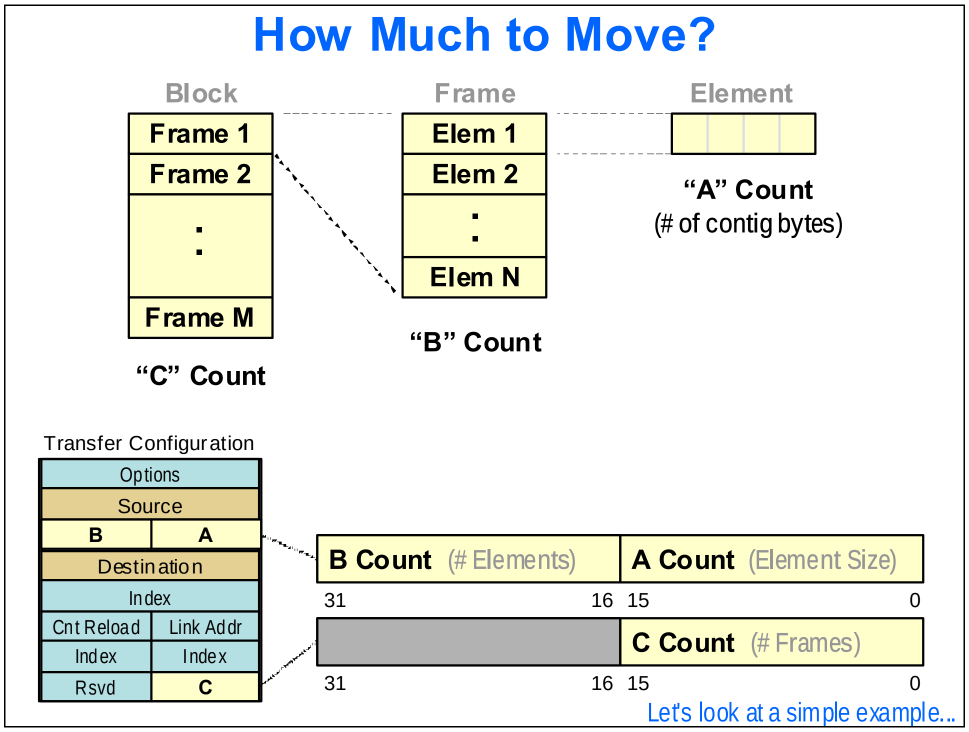
Controlled by :
- Transfer Configuration (i.e. Parameter Set - aka PaRAM or PSET)
- Transfer configuration primarily includes 8 control registers

Source
Length
Destination
Transfer Configuration

The goal of any DMA is to transfer either memory to memory or between peripherals and memory. To program a transfer, you must specify certain transfer configuration parameters such as source address (SRC), destination address (DST) and the length of the transfer (count). A Transfer Configuration is also called a Parameter RAM Set or PARAM Set or PSET. Each PARAM Set contains 8 32-bit registers which define the behavior of the transfer.

The bulk of this chapter will focus on how to program this transfer configuration using the Low Level Driver (LLD) for EDMA3.

How Does a DMA Work?



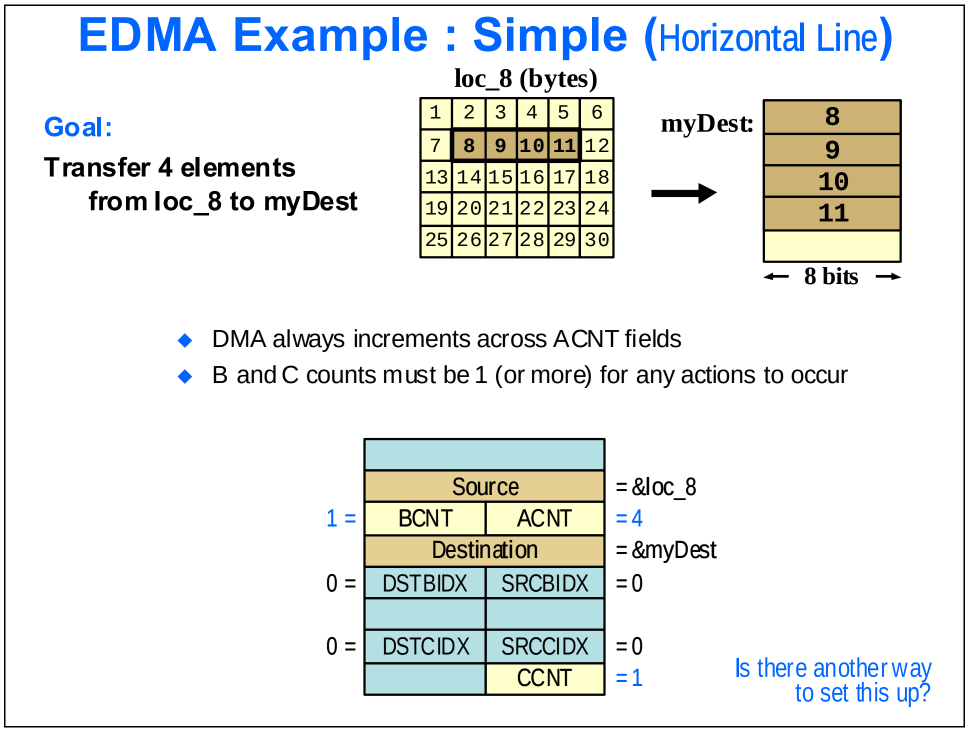
The length of a transfer is specified by three count values: A Count (ACNT) – the number of contiguous bytes (1-64K), B Count (BCNT) – the number of ACNTs (1-64K), and C Count (CCNT), the number of BCNT frames (1-64K) which grouped together can be called a block. Each count value has a range of 1-64K – plenty for just about any imaginable transfer.

The combination of count values provides up to a 3-dimensional (3D) transfer. The reason they were split like this will become more apparent in later slides – mainly to handle indexing and syncing between ACNT and BCNT transfers.

NOTE: ACNT * BCNT * CCNT must NOT equal ZERO or nothing will be transferred. So, all minimum count values are 1. This is a quick way to check to make sure your values are correct. Multiply the 3 numbers together and this will be the total size of the transfer in bytes.

EDMA3 Basic Examples

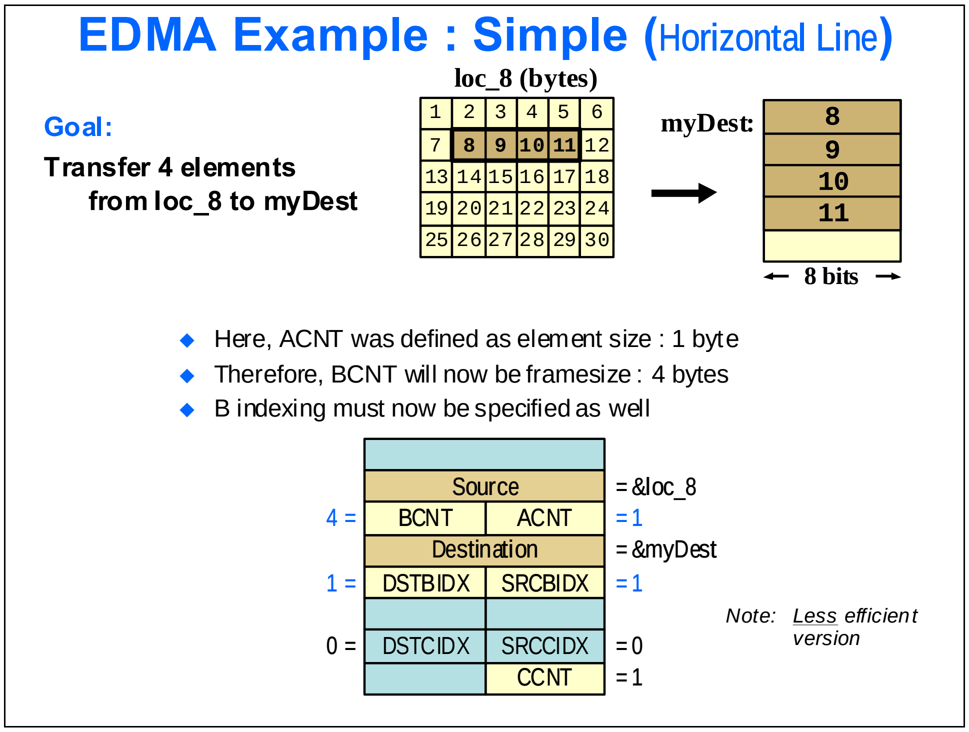
Example 1 – Simple Horizontal Line Xfr



Transfers can be viewed in several ways with different combinations of ACNT, BCNT and CCNT. Which one you choose is highly dependent upon your application and the type of transfer you require. Hopefully through these examples and ones later, you'll have the proper information to make an educated choice.

In this example, our goal is to transfer 4 contiguous bytes that represent a line in our 2D grid. Please note that bytes 1-30 are CONTIGUOUS in memory – they are just shown as a 2D grid to portray a video display. The SRC and DST locations are shown. Next, we must determine the count values. We have two options: (1) ACNT = 4 bytes and BCNT = 1; (2) ACNT = 1 byte and BCNT = 4. For each ACNT bytes, the EDMA3 will request a transfer to the internal buses. So, limiting the number of requests is actually more efficient. So, the best answer is option #1: ACNT = 4 bytes and BCNT = 1. (Note, CCNT must be 1 also – never zero – or nothing will be transferred).

When BCNT and CCNT are both 1, this is called a 1D transfer. This nomenclature shows up when using ACPY3 APIs, so that is why it is mentioned here. If BCNT is greater than 1, it is called a 2D transfer. And, you guessed it, when CCNT is greater than 1, it is called a 3D transfer. We'll see both of those coming up.



EDMA Example : Simple (Horizontal Line)

Goal:

Transfer 4 elements from loc_8 to myDest

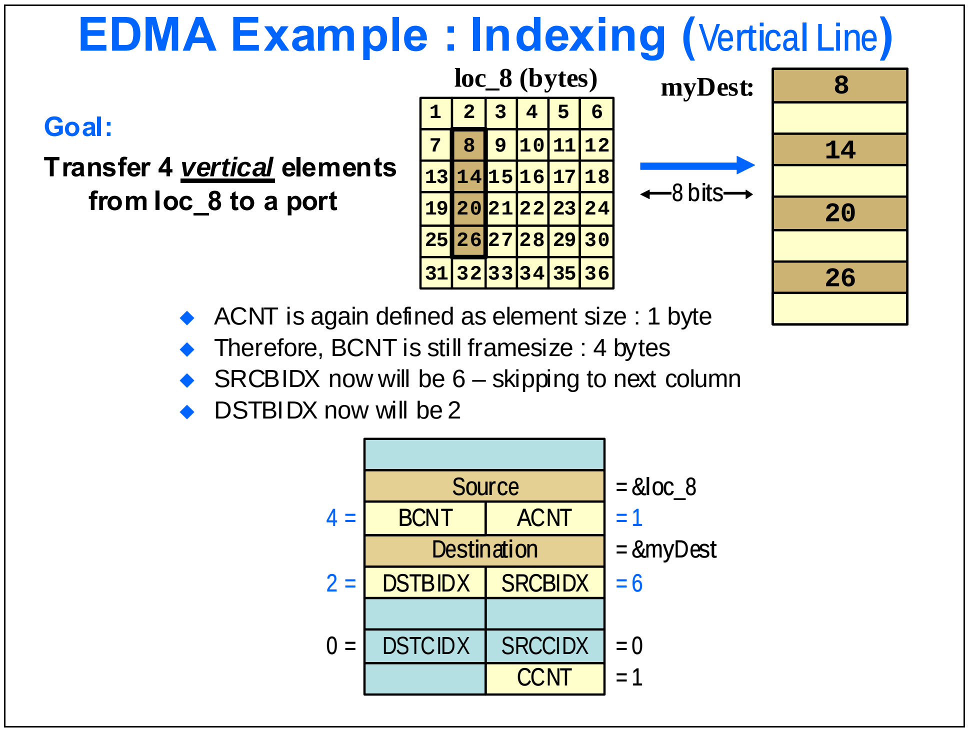
- Here, ACNT was defined as element size : 1 byte
- Therefore, BCNT will now be framesize : 4 bytes
- B indexing must now be specified as well

	Source		= &loc_8
4 =	BCNT	ACNT	= 1
	Destination		= &myDest
1 =	DSTBIDX	SRCBIDX	= 1
0 =	DSTCIDX	SRCCIDX	= 0
		CCNT	= 1

Note: *Less efficient version*

This shows the other case with ACNT = 1 and BCNT = 4. As described on the previous page, this is less efficient in terms of the number of bus requests, but it will still work. Also, if the destination was NOT a memory block, but a peripheral that could only handle one byte at a time (like the XMT side of a serial port), then you would have no option other than to use this method.

Example 2 – Indexing (Vertical Line)



EDMA Example : Indexing (Vertical Line)

Goal:

Transfer 4 *vertical* elements from loc_8 to a port

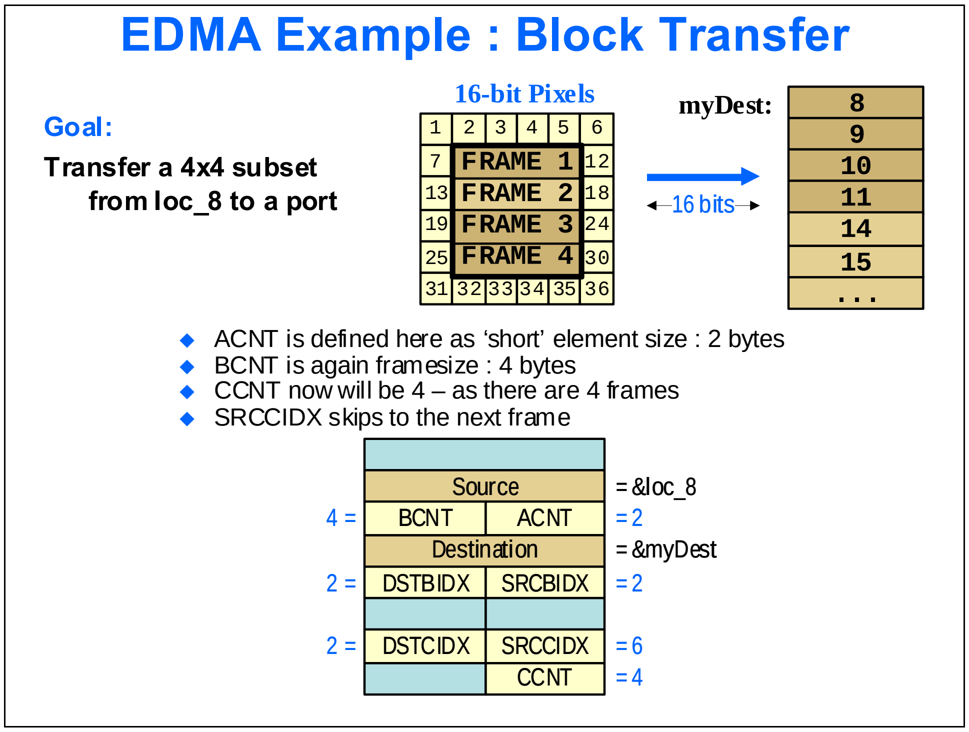
- ACNT is again defined as element size : 1 byte
- Therefore, BCNT is still framesize : 4 bytes
- SRCBIDX now will be 6 – skipping to next column
- DSTBIDX now will be 2

We are now transferring a vertical line. ACNT is the number of contiguous bytes to transfer. Well, in this case, there is only ONE byte to transfer and then we need to “hop” or index an amount to get to the next value in the array. So, ACNT will be 1. Notice the two ‘BIDX values. B index is used when ACNT goes to zero (and BCNT is decremented). We will use ‘BIDX for the SRC to jump from loc_8 to loc_14. We will use DSTBIDX to jump every other spot at myDest. If you hold a pointer at loc_8 and count to loc_14, you get 6 bytes – so SRCBIDX is 6 bytes. For DSTBIDX, it is simply 2 bytes that we want the destination pointer to jump to each time.

So, the index values (SRCBIDX, DSTBIDX) denote the number of BYTES to jump. What if these were short (16-bit) values instead of 8-bit? How would the index values change? Well, they would double in size. SRCBIDX would be 12 and DSTBIDX would be 4. Indexing can be very useful when channel sorting (i.e. you have stereo coming in LRLRLR and you want to channel sort them to LLLL RRRR). This is covered later in this chapter.

One other question. Why couldn’t ACNT be 4 instead with a BCNT of 1? We are transferring the same amount of data. I think the answer is obvious, but you must employ ‘BIDX between ACNT transfers to get the “hops” necessary to perform the operation. So, in this case, you’re stuck with ACNT being 1.

Example 3 – Indexing (Block Transfer)



EDMA Example : Block Transfer

Goal:

Transfer a 4x4 subset from loc_8 to a port

- ACNT is defined here as ‘short’ element size : 2 bytes
- BCNT is again framesize : 4 bytes
- CCNT now will be 4 – as there are 4 frames
- SRCCIDX skips to the next frame

	Source		= &loc_8
4 =	BCNT	ACNT	= 2
	Destination		= &myDest
2 =	DSTBIDX	SRCBIDX	= 2
2 =	DSTCIDX	SRCCIDX	= 6
		CCNT	= 4

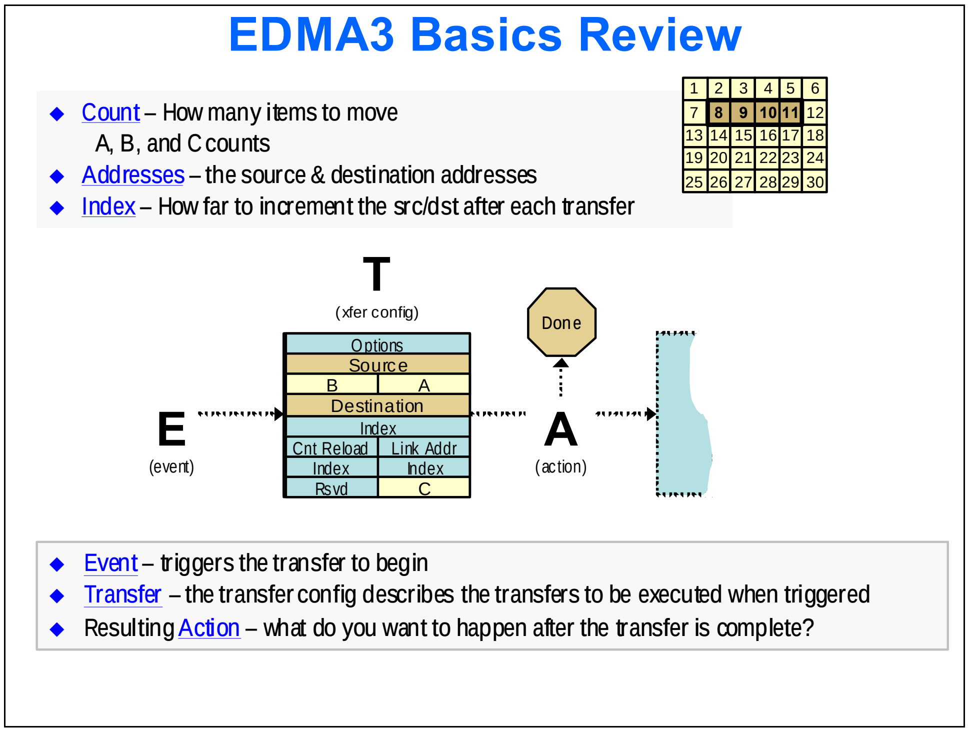
This is taking our examples one step further into a 3D transfer. In this case, we want to transfer a BLOCK of pixels as shown, starting at loc_8 and ending at loc_29. Notice that the data size has changed to 16 bits instead of 8 (kinda tricky, but fun). So, ACNT is now 2 bytes (16-bit pixels). A frame size is 4 ACNT transfers, so BCNT = 4. When ACNT goes to zero and BCNT decrements (when SRCBIDX is employed), what should the value be? Well, they are contiguous, so SRCBIDX is the size of ACNT = 2.

When you get to the end of the frame, BCNT is zero and CCNT decrements. When CCNT decrements, ‘CIDX is employed. From the first byte of loc_11 (the starting of the last 16-bit value in Frame 1) to the first byte of loc_14, how many bytes should we jump to hop from the end of the first frame to the beginning of Frame 2? If you count: loc_12 (2 bytes), loc_13 (2 bytes), loc_14 (2 bytes) – that totals 6 bytes for SRCCIDX.

There are two types of synchronization within EDMA3 – A-sync and AB-sync (discussed further in LLD Synchronization later on). If you are syncing to a peripheral, you almost always use A-sync because most peripherals can only handle one data item at a time (like a serial port). In this case, CIDX is calculated from the LAST ACNT byte transferred when BCNT reaches zero. On the SRC side, that would be from the first byte of loc_11 to the first byte of loc_14 – which is 6 bytes (as shown). Likewise, DSTCIDX would use the same sync and therefore would hop from where loc_11 is stored at myDest to two bytes further – hence DSTCIDX would be 2.

If AB-sync was chosen, ‘CIDX is calculated from the first byte in the first frame to the first byte of the second frame. So, for SRCCIDX, that would 12 bytes. On the DSTCIDX side, the bump would be 4 16-bit values = 8 bytes. The reason for this difference is how the transfer (A or AB) is sent to the transfer controller of the DMA. It goes off the last address it received.

EDMA3 Review



EDMA3 Basics Review

- Count – How many items to move
  A, B, and C counts
- Addresses – the source & destination addresses
- Index – How far to increment the src/dst after each transfer

- Event – triggers the transfer to begin
- Transfer – the transfer config describes the transfers to be executed when triggered
- Resulting Action – what do you want to happen after the transfer is complete?

So, the basic elements of a transfer configuration are count (A, B, C), src/dst addresses and indexes (if required).

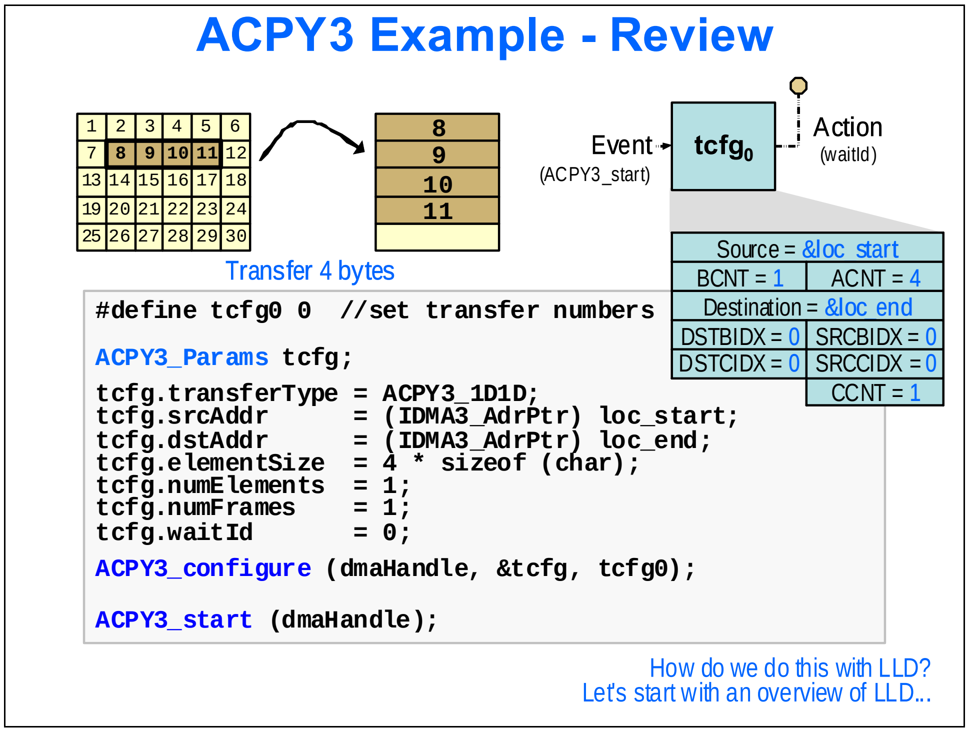
You can view the EDMA3 as having 3 major components in terms of programming a transfer. First, an event "E" must occur to start the transfer – this could be a manual start, an event sync from a peripheral (saying I have data or I need data) or via something called chaining (where the completion of one channel kicks off a different channel to run – explained later).

The "T" for transfer configuration defines how the transfer behaves. Most of the transfer parameters have been covered so far except for Options, Link and Cnt Reload – which will be explained in future slides.

"A" stands for "Action". What do you want to happen when the current transfer completes? Do you want to link to a different configuration PSET? Do you want to interrupt the CPU? Do you want to chain to another channel or a combo of these? These issues will all be covered in the upcoming slides. If you keep this picture in mind, it is a good brain map to understand the fundamentals of how the EDMA3 operates and is programmed.

LLD Overview

ACPY3 Example


1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
8
9
10
11
Transfer 4 bytes
Event
(ACPY3_start)
tcfg$_0$
Action
(waitId)

ACPY3 Example - Review

Source = &loc_start	
BCNT = 1	ACNT = 4
Destination = &loc_end	
DSTBIDX = 0	SRCBIDX = 0
DSTCIDX = 0	SRCCIDX = 0
	CCNT = 1

```
#define tcfg0 0  //set transfer numbers

ACPY3_Params tcfg;

tcfg.transferType = ACPY3_1D1D;
tcfg.srcAddr      = (IDMA3_AdrPtr) loc_start;
tcfg.dstAddr      = (IDMA3_AdrPtr) loc_end;
tcfg.elementSize  = 4 * sizeof (char);
tcfg.numElements  = 1;
tcfg.numFrames    = 1;
tcfg.waitId       = 0;

ACPY3_configure (dmaHandle, &tcfg, tcfg0);

ACPY3_start (dmaHandle);
```

How do we do this with LLD?
Let's start with an overview of LLD...

Well, this is actually NOT a review because we haven't covered ACPY3 at all yet. However, this will give you a sense of what the APIs look like if you decide to use ACPY3 for simple memory-memory transfers with NO synchronization to peripherals.

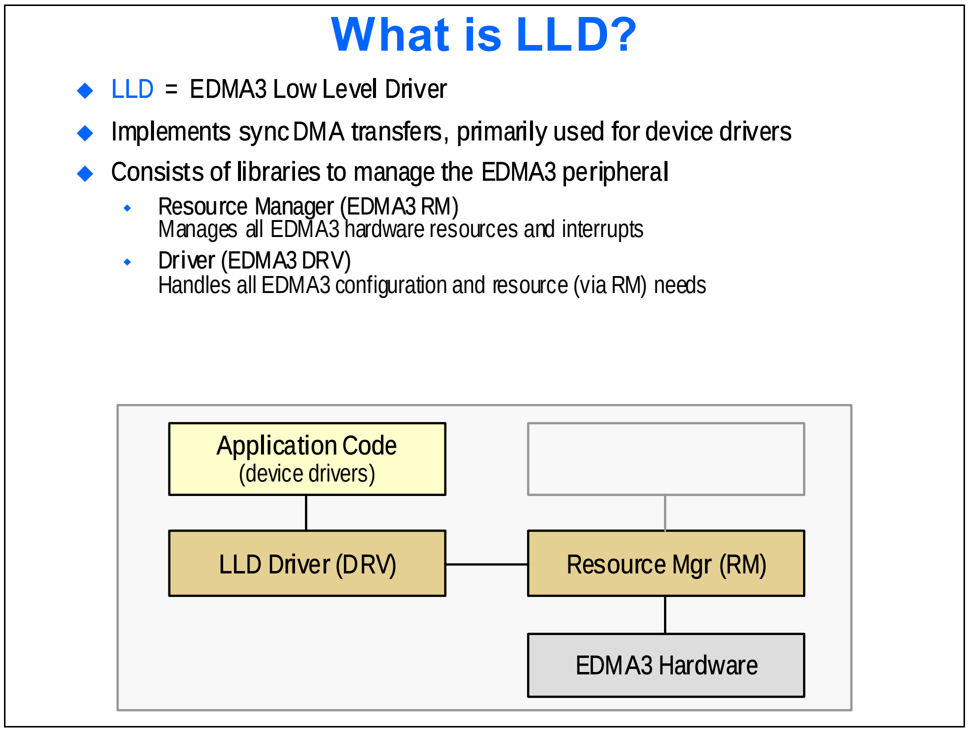
You can see we're using the same horizontal line transfer as before and the transfer params are shown in the upper RH corner. With ACPY3, you define a transfer number (in this case 0) which you can use to poll on to see if the transfer is finished later. You set up a list of parameters in a structure called tcfg as shown. Instead of ACNT, BCNT and CCNT, ACPY3 uses elementSize (ACNT), numElements (BCNT) and numFrames (CCNT).

The dmaHandle is provided via DMAN3 (DMA Manager) which is not shown. ACPY3_configure takes the parameters you specified and initializes the proper PSET with these values. To start a transfer, use ACPY3_start with the dmaHandle.

Using ACPY3 is simple and straightforward. The benefits are that it is easy to use with relatively few APIs to understand. The limitation is that you cannot sync a peripheral to any transfers using ACPY3. The reason for this is that ACPY3 uses the QDMA which only supports asynchronous (manual start) transfers.

Hopefully this gives enough of a sense of how ACPY3 works because we're about to contrast it to LLD which is lower level and has several more APIs because you have the ability to program every bit and every feature of EDMA3 using LLD…

What is the Low Level Driver (LLD) ?



What is LLD?

- LLD = EDMA3 Low Level Driver
- Implements sync DMA transfers, primarily used for device drivers
- Consists of libraries to manage the EDMA3 peripheral
  - Resource Manager (EDMA3 RM)
    Manages all EDMA3 hardware resources and interrupts
  - Driver (EDMA3 DRV)
    Handles all EDMA3 configuration and resource (via RM) needs

Application Code
(device drivers)
LLD Driver (DRV)
Resource Mgr (RM)
EDMA3 Hardware

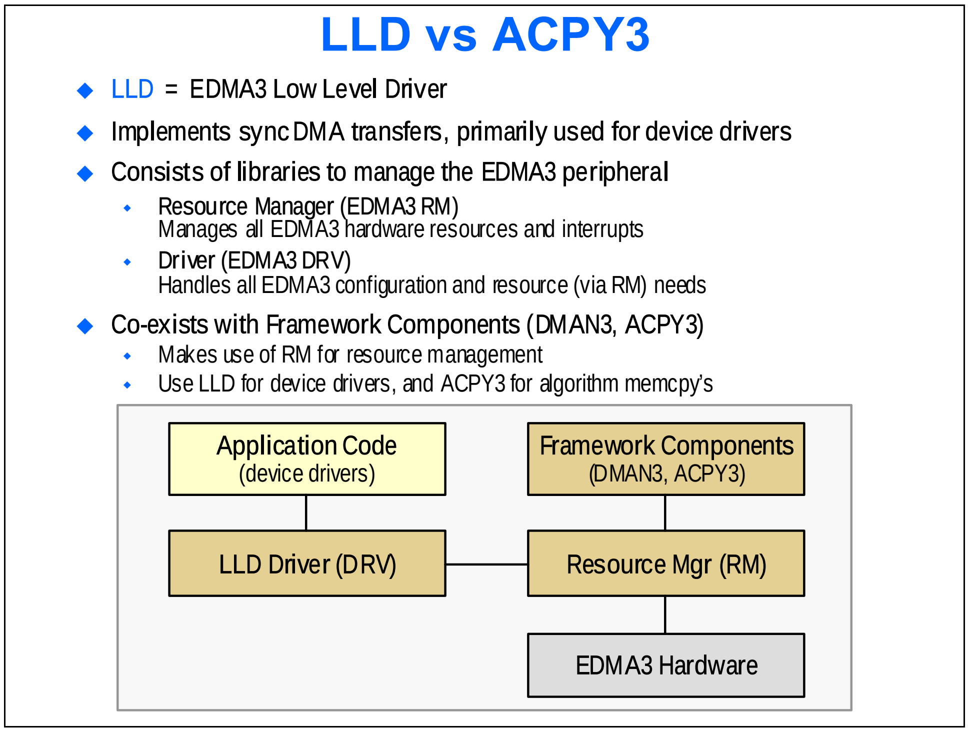
LLD is a set of libraries that allow you to program every feature and option of the EDMA3 peripheral. It is primarily used by driver authors to build I/O drivers that use EDMA3. Application authors can also use LLD to manage transfers within their code. There are actually two libraries – one for the LLD (DRV layer) and one that is more closely tied to the hardware specification (RM).

Most calls you need to make are from the DRV layer. However, there are a few calls during early initialization that are RM calls only. These will be clearly explained in a few more slides.

So, the device driver author or application author uses APIs to call the DRV layer which in turns calls the RM layer to talk directly to the peripheral registers that configure the EDMA3 peripheral registers and hardware.

If you are using framework components, i.e. ACPY3 and DMAN3…it looks like the following…

LLD vs. ACPY3



LLD vs ACPY3

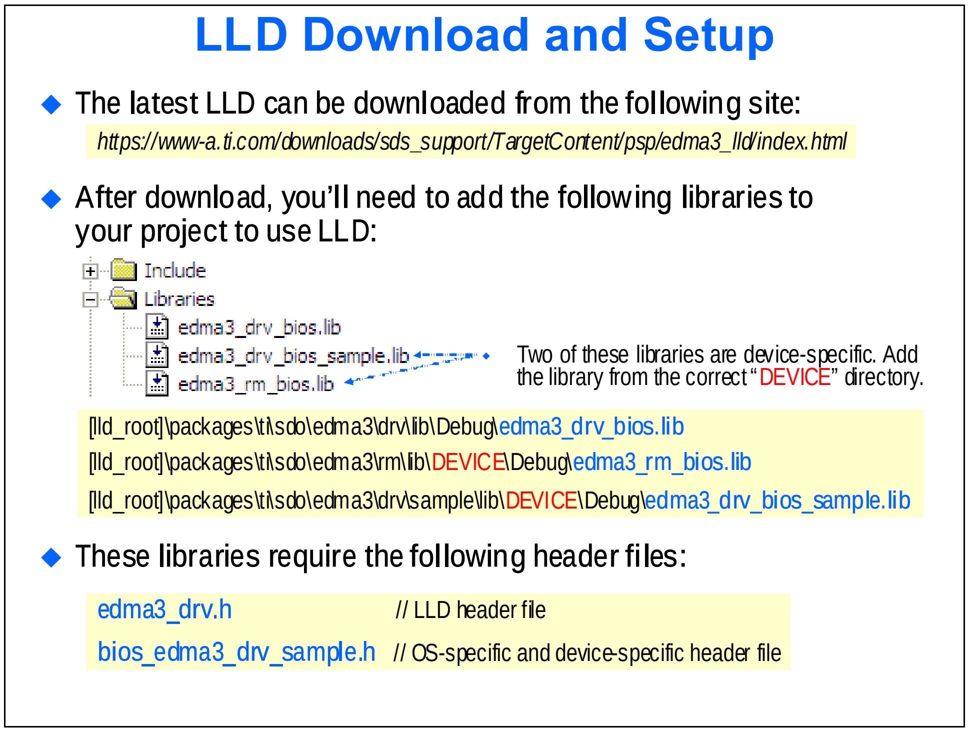
- LLD = EDMA3 Low Level Driver
- Implements sync DMA transfers, primarily used for device drivers
- Consists of libraries to manage the EDMA3 peripheral
  - Resource Manager (EDMA3 RM)
    Manages all EDMA3 hardware resources and interrupts
  - Driver (EDMA3 DRV)
    Handles all EDMA3 configuration and resource (via RM) needs
- Co-exists with Framework Components (DMAN3, ACPY3)
  - Makes use of RM for resource management
  - Use LLD for device drivers, and ACPY3 for algorithm memcpy's

ACPY3 uses the same resource manager (RM) that LLD uses. It makes sense because you wouldn't want driver authors and application developers and algorithms to request more resources than the EDMA3 has. All of these methods require you to request a channel and TCC from the RM which can return an error code that allows you to manage resource allocation.

Again, as we've already documented, use ACPY3 for algorithms and some application code when the need is to only do a memory to memory transfer without synchronization. Use LLD when synchronization, indexing, linking and chaining is desired.

The rest of this chapter is totally dedicated to using the LLD.

ACPY3 Example



LLD Download and Setup

- The latest LLD can be downloaded from the following site:
  https://www-a.ti.com/downloads/sds_support/TargetContent/psp/edma3_lld/index.html
- After download, you'll need to add the following libraries to your project to use LLD:

Include
Libraries
edma3_drv_bios.lib
edma3_drv_bios_sample.lib
edma3_rm_bios.lib

Two of these libraries are device-specific. Add the library from the correct "DEVICE" directory.

```
[lld_root]\packages\ti\sdo\edma3\drv\lib\Debug\edma3_drv_bios.lib
[lld_root]\packages\ti\sdo\edma3\rm\lib\DEVICE\Debug\edma3_rm_bios.lib
[lld_root]\packages\ti\sdo\edma3\drv\sample\lib\DEVICE\Debug\edma3_drv_bios_sample.lib
```

- These libraries require the following header files:

```
edma3_drv.h              // LLD header file
bios_edma3_drv_sample.h  // OS-specific and device-specific header file
```

Sometimes, just getting all of the stuff (libraries, header files, etc.) in the right place can be the most frustrating part of the effort to get any software or library to work properly. When we first started working with the LLD, this was one of the more difficult hurdles to get over. This slide is probably a culmination of man-weeks of effort – but it boils down to just these few items.

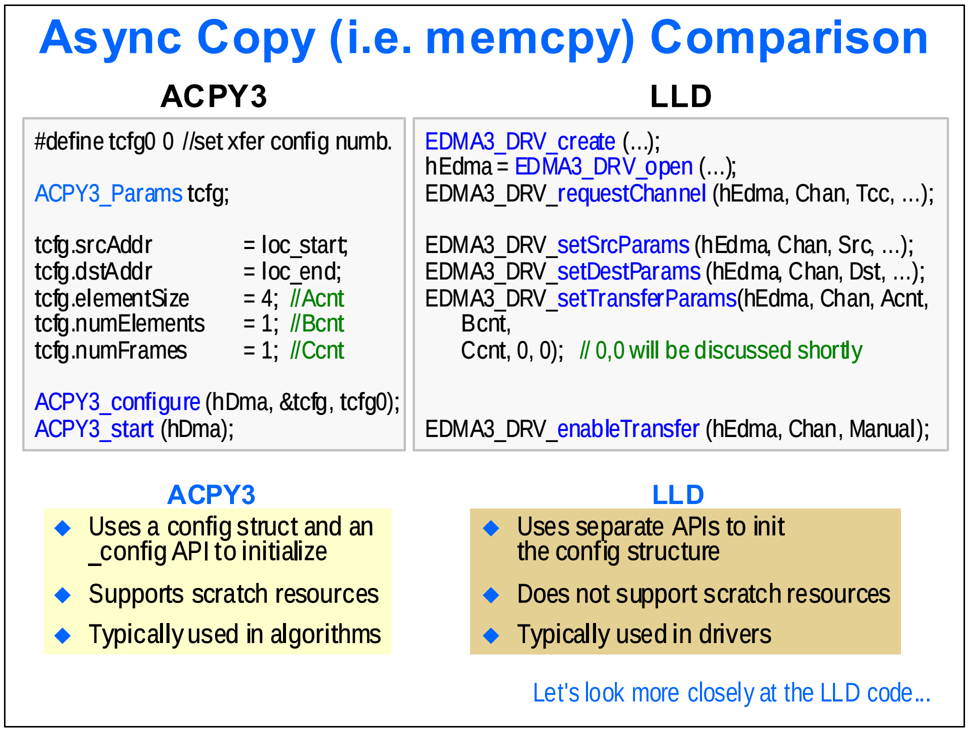
LLD is part of a bigger package of content called PSP (Platform Support Package) which is a bundle of already created drivers for various peripherals and devices. The link to the latest LLD is shown (note: the latest version at the time of this writing was version 6 – there might be newer versions available now). You will need to sign up with a userid and password, but that's typical. Download the software and place it in an obvious directory – our preference is C:\TI\. But you can use whatever you like.

LLD uses the DSP/BIOS RTOS within the libraries. The location of each library is shown on the slide above. You will need to add three libraries to your project to get your code to work properly. As previously stated, two of them are quite obvious – the _drv and _rm libraries. The one that is WAY NOT obvious is the _sample.lib. At first, you think "sample" means it is like an "example" library. Bad naming convention and this one thing spun the authors of this chapter into la-la land for way too long. In the LLD world, SAMPLE = DEVICE. This is the device-specific library that describes the actual hardware on the chosen device – i.e. 6455, 6437 and 6747 (or whatever devices are supported). This one piece of knowledge will save you hours if not days. You're welcome.

As always, libraries require header files. Those are listed at the bottom of this slide.

LLD Configuration and Initialization

Example Code – ACPY3 vs. LLD



Async Copy (i.e. memcpy) Comparison

ACPY3

```
#define tcfg0 0 //set xfer config numb.

ACPY3_Params tcfg;

tcfg.srcAddr        = loc_start;
tcfg.dstAddr        = loc_end;
tcfg.elementSize    = 4;  //Acnt
tcfg.numElements    = 1;  //Bcnt
tcfg.numFrames      = 1;  //Ccnt

ACPY3_configure (hDma, &tcfg, tcfg0);
ACPY3_start (hDma);
```

LLD

```
EDMA3_DRV_create (...);
hEdma = EDMA3_DRV_open (...);
EDMA3_DRV_requestChannel (hEdma, Chan, Tcc, ...);

EDMA3_DRV_setSrcParams (hEdma, Chan, Src, ...);
EDMA3_DRV_setDestParams (hEdma, Chan, Dst, ...);
EDMA3_DRV_setTransferParams(hEdma, Chan, Acnt,
    Bcnt,
    Ccnt, 0, 0);  // 0,0 will be discussed shortly


EDMA3_DRV_enableTransfer (hEdma, Chan, Manual);
```

ACPY3
- Uses a config struct and an _config API to initialize
- Supports scratch resources
- Typically used in algorithms

LLD
- Uses separate APIs to init the config structure
- Does not support scratch resources
- Typically used in drivers

Let's look more closely at the LLD code...

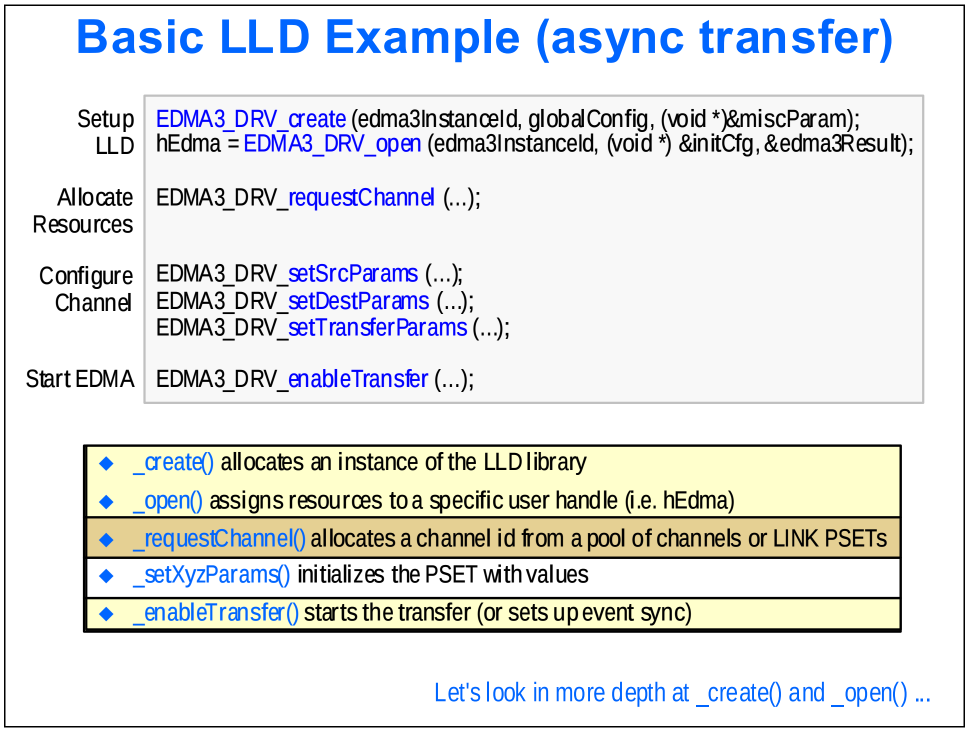
The goal here is to draw a parallel or comparison between how ACPY3 works and LLD – and how the APIs from each library line up. Some of similar and some are different. If you think about ACPY3 being a higher level interface – fewer APIs and based on structures – and you think LLD is "lower level" and gives you more options which means more APIs, then you have a sense of how they differ.

We've seen a similar example of the ACPY3 code before. You define a structure with the parameters you want, _configure it (write the values to the hardware) and then manually start it with _start. Done.

For LLD, you have a few more APIs to accomplish the same thing. First, _create and _open are used to eventually provide a handle to the EDMA3 hardware. You then request a channel (get a channel ID) that you use for the rest of the initialization. Instead of using a structure, LLD has separate APIs for SRC address, DST address and count values as shown on the slide. To trigger the transfer, you use _enableTransfer and the type of transfer you prefer – manual (async start) or SYNC (sync'd start). You will see these LLD APIs more in future slides.

For simple async transfers (which is shown above), LLD doesn't look too exciting. However, once we probe into more capabilities of the EDMA3, the power you have as a programmer will become apparent.

Example – LLD (async transfer)



Basic LLD Example (async transfer)

Setup LLD	EDMA3_DRV_create (edma3InstanceId, globalConfig, (void *)&miscParam);
	hEdma = EDMA3_DRV_open (edma3InstanceId, (void *) &initCfg, &edma3Result);

Allocate Resources	EDMA3_DRV_requestChannel (...);

Configure Channel	EDMA3_DRV_setSrcParams (...);
	EDMA3_DRV_setDestParams (...);
	EDMA3_DRV_setTransferParams (...);

Start EDMA	EDMA3_DRV_enableTransfer (...);

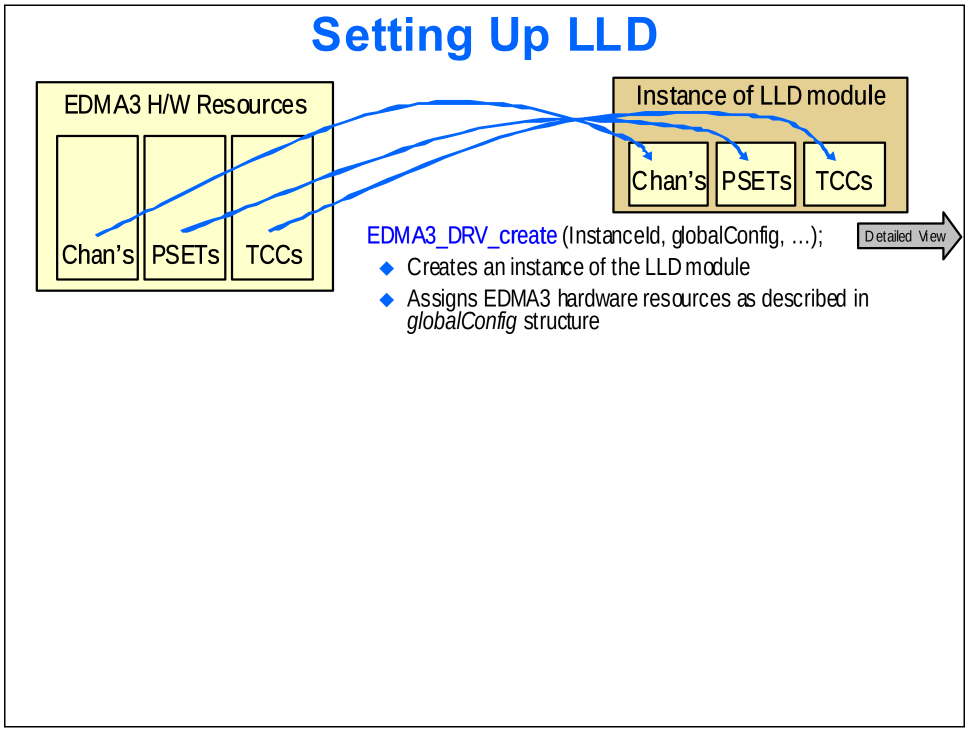
- _create() allocates an instance of the LLD library
- _open() assigns resources to a specific user handle (i.e. hEdma)
- _requestChannel() allocates a channel id from a pool of channels or LINK PSETs
- _setXyzParams() initializes the PSET with values
- _enableTransfer() starts the transfer (or sets up event sync)

Let's look in more depth at _create() and _open() ...

This example shows the set of APIs to program an EDMA3 transfer using LLD. Of course, not all of the parameters are shown – we'll get into more details soon. If you look at the left top of this slide, you can see it broken into 4 pieces – Setup (getting a handle to the EDMA), Allocating Resources (channel id and TCC value), Configuration (setting the src/dst, count values and other options) and Start triggers the execution to occur.

On the next few slides, we're look more in depth at the _create and _open APIs…

Setting Up the LLD (_create API)



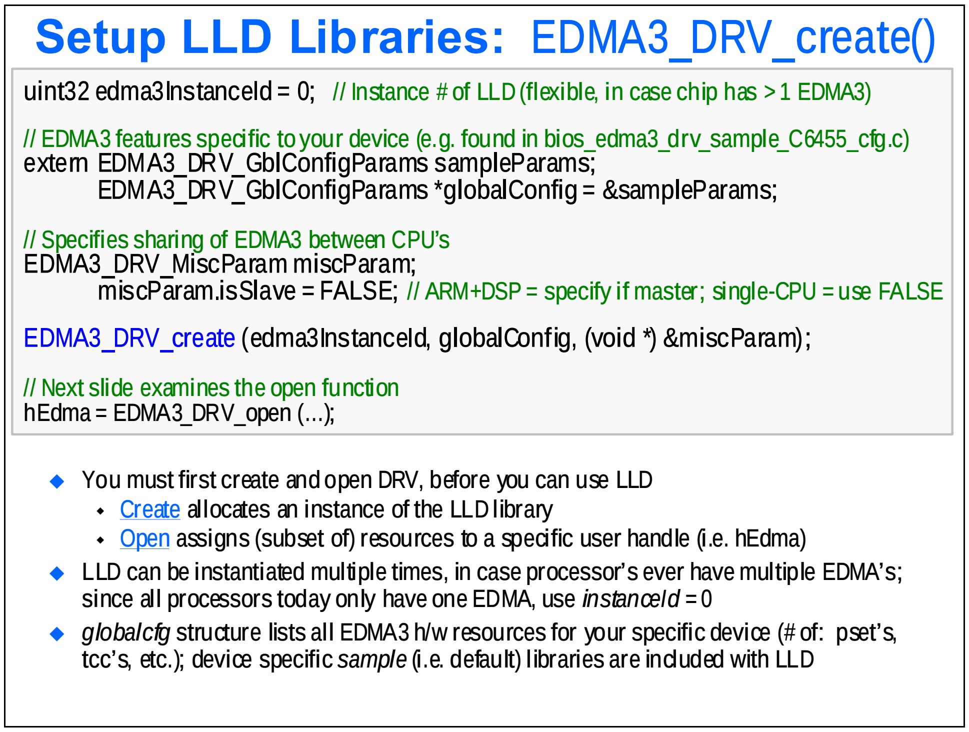
Setting Up LLD

EDMA3_DRV_create (InstanceId, globalConfig, ...);

- Creates an instance of the LLD module
- Assigns EDMA3 hardware resources as described in *globalConfig* structure

We start with the actual devices resources on the left-hand side: channels (or events), PSETs (Parameter RAM sets) and TCCs (transfer complete codes). These are actually the physical hardware resources on your device. _create creates an instance or copy of these resources based upon the globalConfig structure for your specific device. From this instanceid, using the _open API, we can receive a handle to the EDMA3 resources.

Sidebar - _create Details



Setup LLD Libraries: EDMA3_DRV_create()

```
uint32 edma3InstanceId = 0;   // Instance # of LLD (flexible, in case chip has > 1 EDMA3)

// EDMA3 features specific to your device (e.g. found in bios_edma3_drv_sample_C6455_cfg.c)
extern EDMA3_DRV_GblConfigParams sampleParams;
       EDMA3_DRV_GblConfigParams *globalConfig = &sampleParams;

// Specifies sharing of EDMA3 between CPU's
EDMA3_DRV_MiscParam miscParam;
       miscParam.isSlave = FALSE; // ARM+DSP = specify if master; single-CPU = use FALSE

EDMA3_DRV_create (edma3InstanceId, globalConfig, (void *) &miscParam);

// Next slide examines the open function
hEdma = EDMA3_DRV_open (...);
```

- You must first create and open DRV, before you can use LLD
  - Create allocates an instance of the LLD library
  - Open assigns (subset of) resources to a specific user handle (i.e. hEdma)
- LLD can be instantiated multiple times, in case processor's ever have multiple EDMA's; since all processors today only have one EDMA, use *instanceId* = 0
- *globalcfg* structure lists all EDMA3 h/w resources for your specific device (# of: pset's, tcc's, etc.); device specific *sample* (i.e. default) libraries are included with LLD

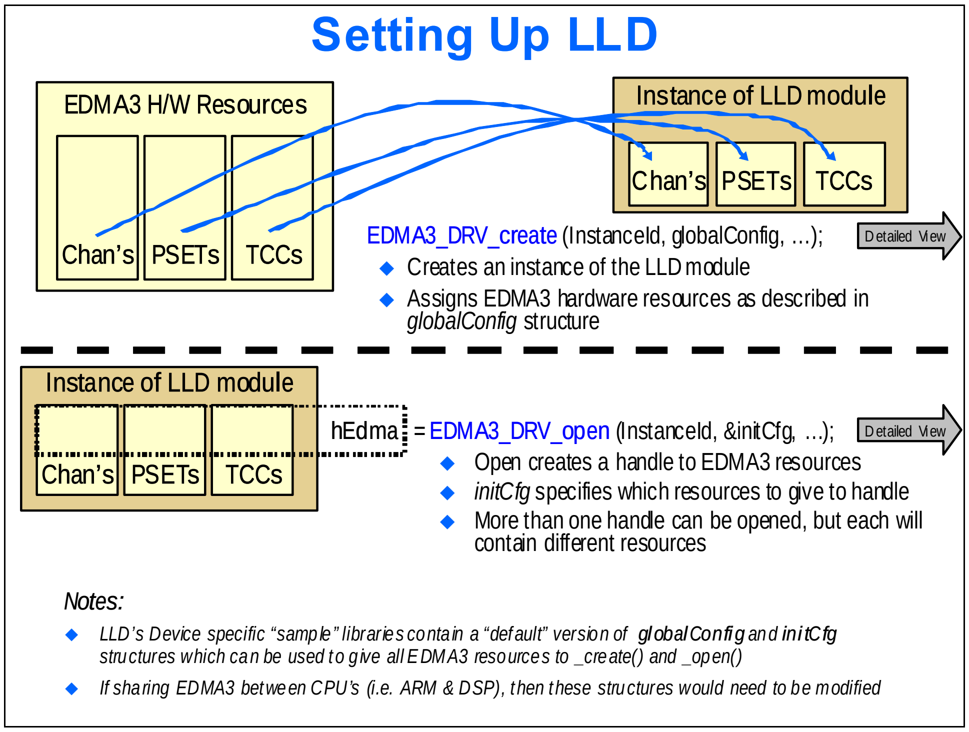
This code can basically be copied as is. The init code for any library can sometimes be a little bothersome and cause more questions than answers. The _create API requires 3 parameters: (1) edma3InstanceId (which is a return value); (2) globalConfig (which contains the specific resources on your device); (3) miscParam structure (which defines if this EDMA is a master or slave).

The instanceId is basically a handle to the instance of the LLD library for your specific device. The contents of your specific EDMA3 (PSETs, events and TCCs) are found in the _sample_DEVICE_cfg.c file as shown in the comment – in a structure called sampleParams. Remember, sample is a poor name – and that sample = DEVICE. Kwirky but true. Confusing, but true. That's how the globalConfig structure gets initialized and finally used in the _create API. If you had more than one EDMA3 on your device (which is rare if nonexistent), you might have one instanceId pointing to one EDMA3 and another instanceId pointing to the other EDMA3.

The miscParam structure only has one element we care about - .isSlave. Most processors that are single core (i.e. NO ARM + DSP) will use the value FALSE.

That's it. _create is done. We will use the instanceId along with the _open API which is discussed next.

LLD - _open API



Setting Up LLD

EDMA3_DRV_create (InstanceId, globalConfig, ...);

- Creates an instance of the LLD module
- Assigns EDMA3 hardware resources as described in *globalConfig* structure

hEdma = EDMA3_DRV_open (InstanceId, &initCfg, ...);

- Open creates a handle to EDMA3 resources
- *initCfg* specifies which resources to give to handle
- More than one handle can be opened, but each will contain different resources

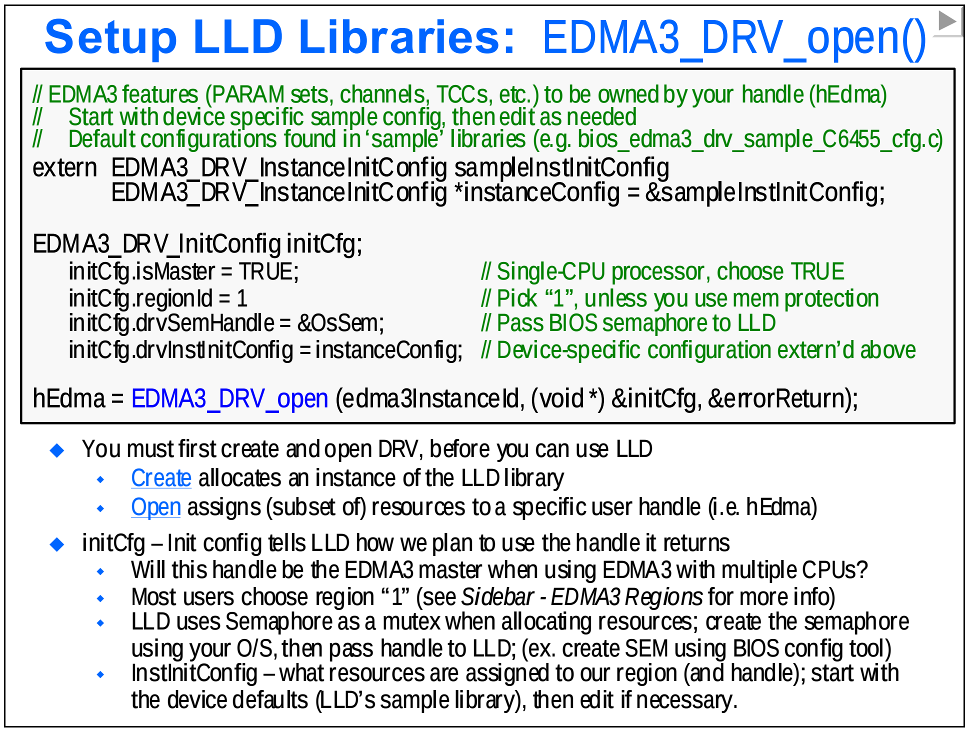
Notes:

- *LLD's Device specific "sample" libraries contain a "default" version of* ***globalConfig*** *and* ***initCfg*** *structures which can be used to give all EDMA3 resources to _create() and _open()*
- *If sharing EDMA3 between CPU's (i.e. ARM & DSP), then these structures would need to be modified*

The _create API is a copy of the entire set of resources on your specific device. With _open, you can assign all of these resources (which is done the majority of the time) or a subset of these resources. So, you could have multiple hEdma handles to different resources of one EDMA3 peripheral. initCfg would then need to be modified if multiple handles are requested. However, the most common use case is to assign them all to one EDMA handle.

More details on the code follow…

LLD - _open Details



Setup LLD Libraries: EDMA3_DRV_open()

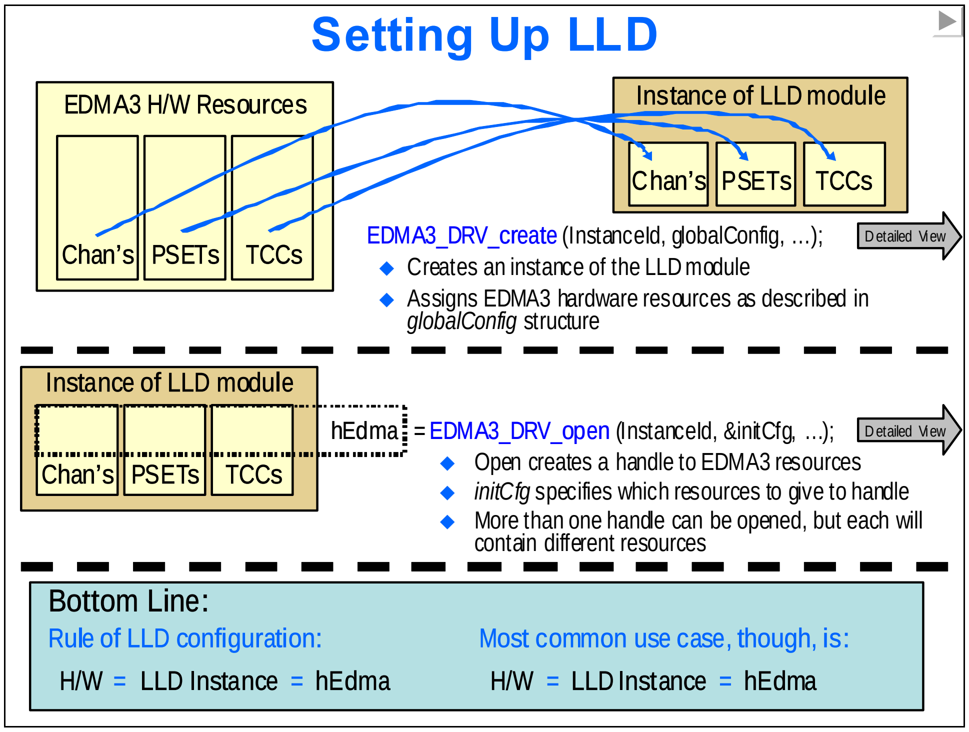
```
// EDMA3 features (PARAM sets, channels, TCCs, etc.) to be owned by your handle (hEdma)
//    Start with device specific sample config, then edit as needed
//    Default configurations found in 'sample' libraries (e.g. bios_edma3_drv_sample_C6455_cfg.c)
extern  EDMA3_DRV_InstanceInitConfig sampleInstInitConfig
        EDMA3_DRV_InstanceInitConfig *instanceConfig = &sampleInstInitConfig;

EDMA3_DRV_InitConfig initCfg;
    initCfg.isMaster = TRUE;                  // Single-CPU processor, choose TRUE
    initCfg.regionId = 1                      // Pick "1", unless you use mem protection
    initCfg.drvSemHandle = &OsSem;            // Pass BIOS semaphore to LLD
    initCfg.drvInstInitConfig = instanceConfig;  // Device-specific configuration extern'd above

hEdma = EDMA3_DRV_open (edma3InstanceId, (void *) &initCfg, &errorReturn);
```

- You must first create and open DRV, before you can use LLD
  - Create allocates an instance of the LLD library
  - Open assigns (subset of) resources to a specific user handle (i.e. hEdma)
- initCfg – Init config tells LLD how we plan to use the handle it returns
  - Will this handle be the EDMA3 master when using EDMA3 with multiple CPUs?
  - Most users choose region "1" (see *Sidebar - EDMA3 Regions* for more info)
  - LLD uses Semaphore as a mutex when allocating resources; create the semaphore using your O/S, then pass handle to LLD; (ex. create SEM using BIOS config tool)
  - InstInitConfig – what resources are assigned to our region (and handle); start with the device defaults (LLD's sample library), then edit if necessary.

_open has three parameters: (1) instanceId which was created with the _create call; (2) initCfg which specifies the resources allocated to our EDMA handle (hEdma); (3) a return error code. The resources assigned to initCfg again come from the "sample" code found in _sample_DEVICE_cfg.c. Yes, poor naming again, but there you go. Also within the initCfg structure, you must specify isMaster (typically TRUE), regionId (use "1" as a default – more on regions in a few slides), a semaphore used as a mutex for resource protection within the driver, and the instanceConfig which contains the device-specific resources.

The result is a handle to the EDMA3 resources that your future LLD APIs will access, configure and use.

Setting Up LLD – Options


Setting Up LLD

EDMA3_DRV_create (InstanceId, globalConfig, ...);

- Creates an instance of the LLD module
- Assigns EDMA3 hardware resources as described in *globalConfig* structure

hEdma = EDMA3_DRV_open (InstanceId, &initCfg, ...);

- Open creates a handle to EDMA3 resources
- *initCfg* specifies which resources to give to handle
- More than one handle can be opened, but each will contain different resources

Bottom Line:

Rule of LLD configuration:

H/W = LLD Instance = hEdma

Most common use case, though, is:

H/W = LLD Instance = hEdma

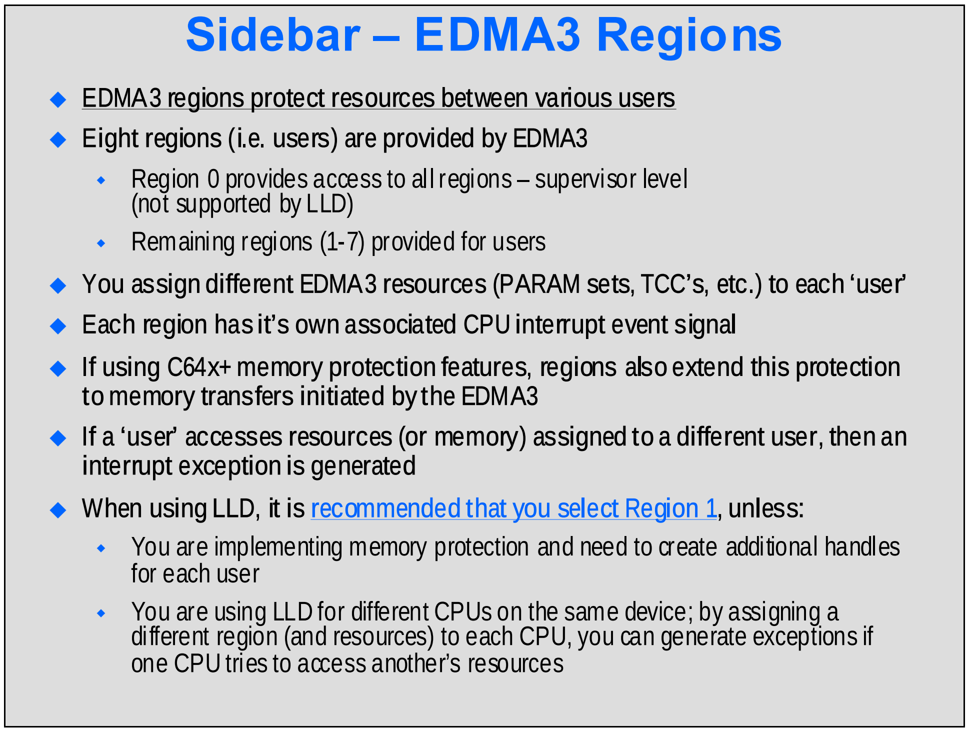
As you can see, this has been a build up to this final slide which has all the init APIs together and the common use case specified. In the bottom line area, the first equation should read:

H/W >= LLD Instance >= hEdma

For some reason, PowerPoint played a trick and wouldn't show this properly. In other words, the instance (copy) of the hardware will be the same or smaller than the actual hardware on the device (by modifying the _cfg.c file and thereby changing the sampleParams structure). In the same way, hEdma could point to only a subset of the resources contained in the instanceId by modifying the sampleInstInitConfig structure in _cfg.c file.

However, the most common use case is to NOT modify these files and have the hEdma handle point to ALL of the resources. And, that's what the authors did when they built the examples that you have access to.

EDMA3 Regions – Sidebar



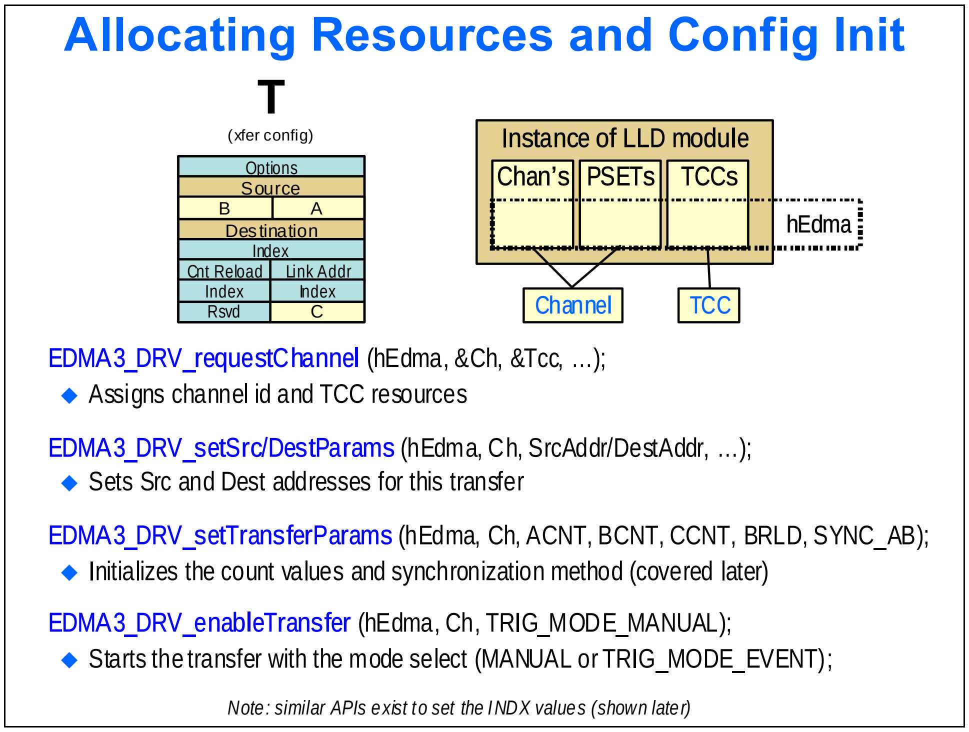
Sidebar – EDMA3 Regions

- EDMA3 regions protect resources between various users
- Eight regions (i.e. users) are provided by EDMA3
  - Region 0 provides access to all regions – supervisor level (not supported by LLD)
  - Remaining regions (1-7) provided for users
- You assign different EDMA3 resources (PARAM sets, TCC's, etc.) to each 'user'
- Each region has it's own associated CPU interrupt event signal
- If using C64x+ memory protection features, regions also extend this protection to memory transfers initiated by the EDMA3
- If a 'user' accesses resources (or memory) assigned to a different user, then an interrupt exception is generated
- When using LLD, it is recommended that you select Region 1, unless:
  - You are implementing memory protection and need to create additional handles for each user
  - You are using LLD for different CPUs on the same device; by assigning a different region (and resources) to each CPU, you can generate exceptions if one CPU tries to access another's resources

Sidebars are usually for extra details that you may or may not find interesting or helpful. In a complex system that uses exception handling and multiple CPUs accessing the same EDMA3 peripheral, regions are extremely important. Otherwise, you can use Region 1 and be just fine.

The text above says just that, so we'll just let it speak for itself.

Allocating Resources and Config Init



Allocating Resources and Config Init

EDMA3_DRV_requestChannel (hEdma, &Ch, &Tcc, …);
- Assigns channel id and TCC resources

EDMA3_DRV_setSrc/DestParams (hEdma, Ch, SrcAddr/DestAddr, …);
- Sets Src and Dest addresses for this transfer

EDMA3_DRV_setTransferParams (hEdma, Ch, ACNT, BCNT, CCNT, BRLD, SYNC_AB);
- Initializes the count values and synchronization method (covered later)

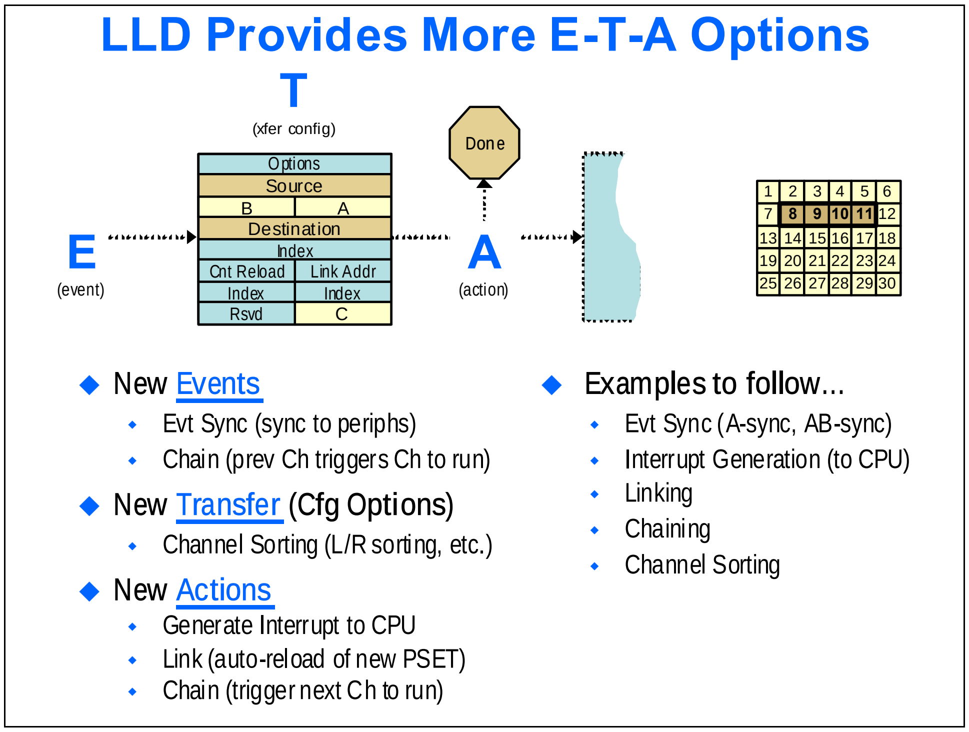
EDMA3_DRV_enableTransfer (hEdma, Ch, TRIG_MODE_MANUAL);
- Starts the transfer with the mode select (MANUAL or TRIG_MODE_EVENT);

Note: similar APIs exist to set the INDX values (shown later)

The APIs shown allow you to: (1) request a channel and TCC resources; (2) set the Src/Dst parameters; (3) set the count values and type of sync used (A or AB); (4) start the transfer either manually (async) or via a trigger (event sync).

We will address each of these in detail throughout the following example code.

LLD – More Events, Transfers and Actions



LLD Provides More E-T-A Options

T
(xfer config)
Options
Source
B
A
Destination
Index
Cnt Reload
Link Addr
Index
Index
Rsvd
C
E
(event)
Done
A
(action)
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

- New Events
  - Evt Sync (sync to periphs)
  - Chain (prev Ch triggers Ch to run)
- New Transfer (Cfg Options)
  - Channel Sorting (L/R sorting, etc.)
- New Actions
  - Generate Interrupt to CPU
  - Link (auto-reload of new PSET)
  - Chain (trigger next Ch to run)
- Examples to follow…
  - Evt Sync (A-sync, AB-sync)
  - Interrupt Generation (to CPU)
  - Linking
  - Chaining
  - Channel Sorting

This is somewhat of a review of what we’ve seen before, but it does set the stage for the following set of slides which will go through programming many of the common features and capabilities of the EDMA3.

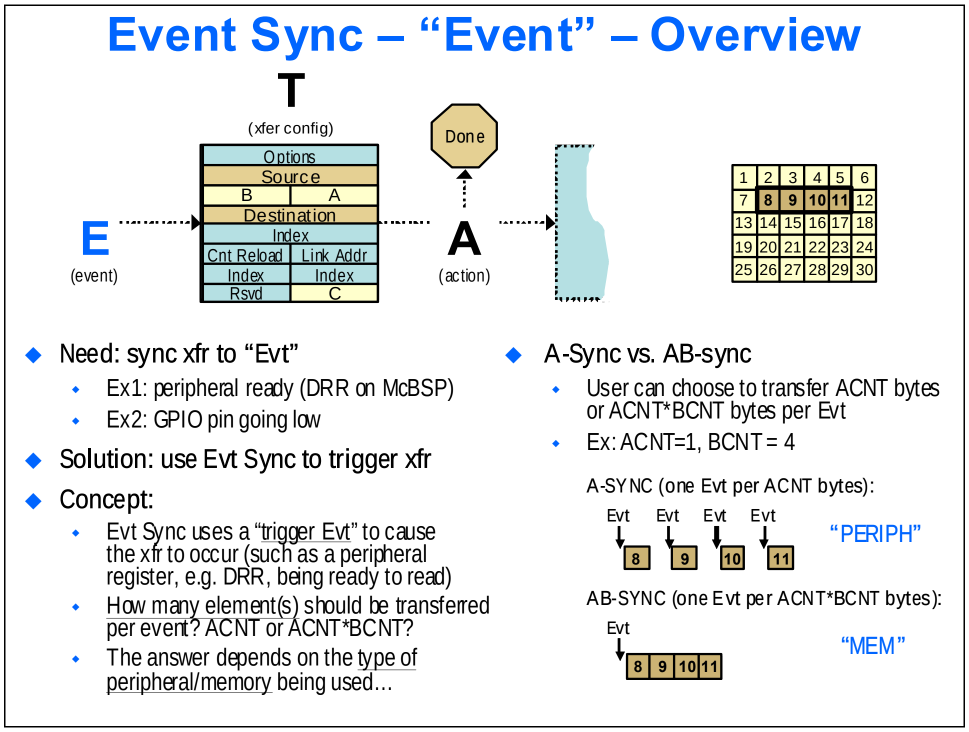
Events trigger transfers to start – this can be done manually (like an async start) or via an event which could be either a peripheral sync or a chaining event (covered later).

Transfer options include using indexing to channel sort incoming data – again, covered in an example later.

Actions refer to what happens when the transfer completes – either generating a CPU interrupt, linking to another PSET (auto reload) and/or chaining (triggering another channel to run).

In the following slides, we plan to cover each one with the theory/methodology of the concept followed by example code showing how it works with the LLD APIs.

"Event" – Event Synchronization



Event Sync – "Event" – Overview

- Need: sync xfr to "Evt"
  - Ex1: peripheral ready (DRR on McBSP)
  - Ex2: GPIO pin going low
- Solution: use Evt Sync to trigger xfr
- Concept:
  - Evt Sync uses a "trigger Evt" to cause the xfr to occur (such as a peripheral register, e.g. DRR, being ready to read)
  - How many element(s) should be transferred per event? ACNT or ACNT*BCNT?
  - The answer depends on the type of peripheral/memory being used…
- A-Sync vs. AB-sync
  - User can choose to transfer ACNT bytes or ACNT*BCNT bytes per Evt
  - Ex: ACNT=1, BCNT = 4

  A-SYNC (one Evt per ACNT bytes):

  AB-SYNC (one Evt per ACNT*BCNT bytes):

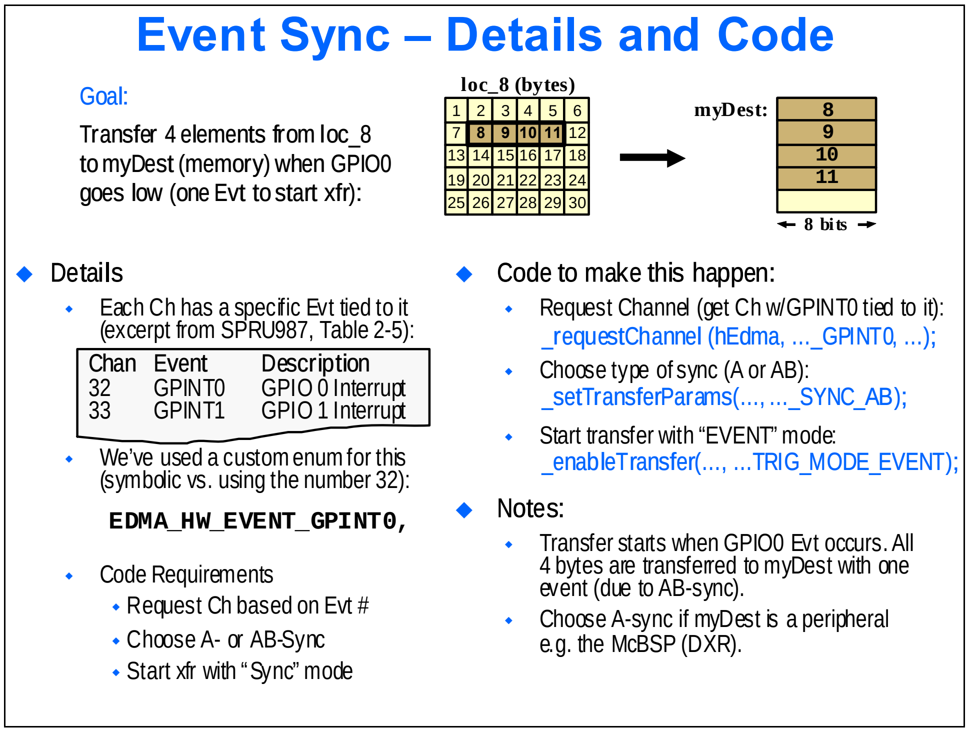
Most peripherals (if not all) have a sync event tied to the EDMA. Just like most peripherals have an interrupt source that you can use to interrupt the CPU, sync events trigger the DMA to execute a transfer. This transfer could simply be a single data item or multiple values or an entire block of memory. Typical sync events would be the DRR register of the serial port being "ready" to read or a GPIO pin going low. Refer to your datasheet for the specific list of sync events on your device.

You have a choice to transfer ACNT bytes (A-sync) or ACNT * BCNT bytes (AB-sync). Usually, peripherals require A-sync transfers because they are limited in size (e.g. the serial port can't provide 16 32-bit data values at one time – it is normally one 16-bit value). AB-sync is used in various ways, but primarily for blocks of data with one single event that transfers the entire block (GPIO pin goes low, transfer A*B to this dest, etc.). The good news is that you can architect this any way you like as a programmer.

If you view the example and the diagrams that go with it, you can see that A-sync moves ACNT bytes (in this case one byte) per event. Using AB-sync, the entire line (ACNT * BCNT) is moved per event. ABC-sync does not exist.

In the following slide, we'll look at how you program this transfer using the LLD APIs.

Event Sync – Code Details



loc_8 (bytes)
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
myDest:
8
9
10
11
8 bits

Event Sync – Details and Code

Goal:

Transfer 4 elements from loc_8 to myDest (memory) when GPIO0 goes low (one Evt to start xfr):

- Details
  - Each Ch has a specific Evt tied to it (excerpt from SPRU987, Table 2-5):

Chan	Event	Description
32	GPINT0	GPIO 0 Interrupt
33	GPINT1	GPIO 1 Interrupt

  - We've used a custom enum for this (symbolic vs. using the number 32):

```
EDMA_HW_EVENT_GPINT0,
```

  - Code Requirements
    - Request Ch based on Evt #
    - Choose A- or AB-Sync
    - Start xfr with "Sync" mode
- Code to make this happen:
  - Request Channel (get Ch w/GPINT0 tied to it): _requestChannel (hEdma, …_GPINT0, …);
  - Choose type of sync (A or AB): _setTransferParams(…, …_SYNC_AB);
  - Start transfer with "EVENT" mode: _enableTransfer(…, …TRIG_MODE_EVENT);
- Notes:
  - Transfer starts when GPIO0 Evt occurs. All 4 bytes are transferred to myDest with one event (due to AB-sync).
  - Choose A-sync if myDest is a peripheral e.g. the McBSP (DXR).

This is the same example we looked at before. However, the only change is that we want to transfer the horizontal line when GPIO0 goes low (event triggered).

Each event has a specific channel (0-63) tied to it. Each device will have a slightly different list of events and channel numbers – so refer to your datasheet for the specifics. The example shown is for the 'C6455 device. The authors of this chapter used enums in a file to make it easier for the user (and us) not to make a mistake (you can see this in the downloadable examples).

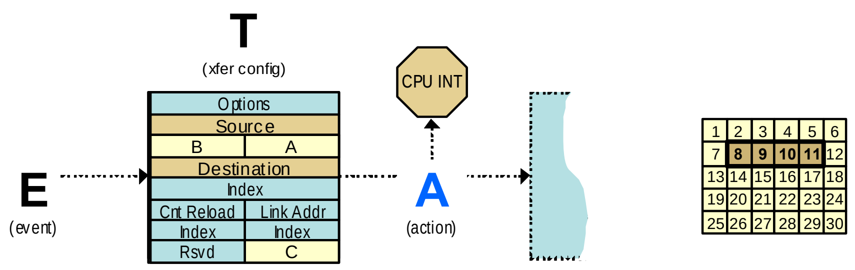
Once you know the event you want, use that event name in the _requestChannel API so that the exact channel with that event tied to it will be returned.

Next, you need to choose the type of sync mode – A or AB. In this example, ACNT = 1, and we want to transfer ACNT * BCNT with one event, so AB-sync is specified. Also, we're transferring memory to memory, so typically AB-sync is used for this.

There are two ways to start a transfer – manually and via a sync event. The enumeration for this value is symbolically shown as "TRIG_MODE_EVENT" using the _enableTransfer API. So, you can set all this up in your initialization code and the transfer will not occur until the GPIO0 event occurs.

Action – EDMA3 Interrupt Generation

Interrupt Generation – "Action" - Overview



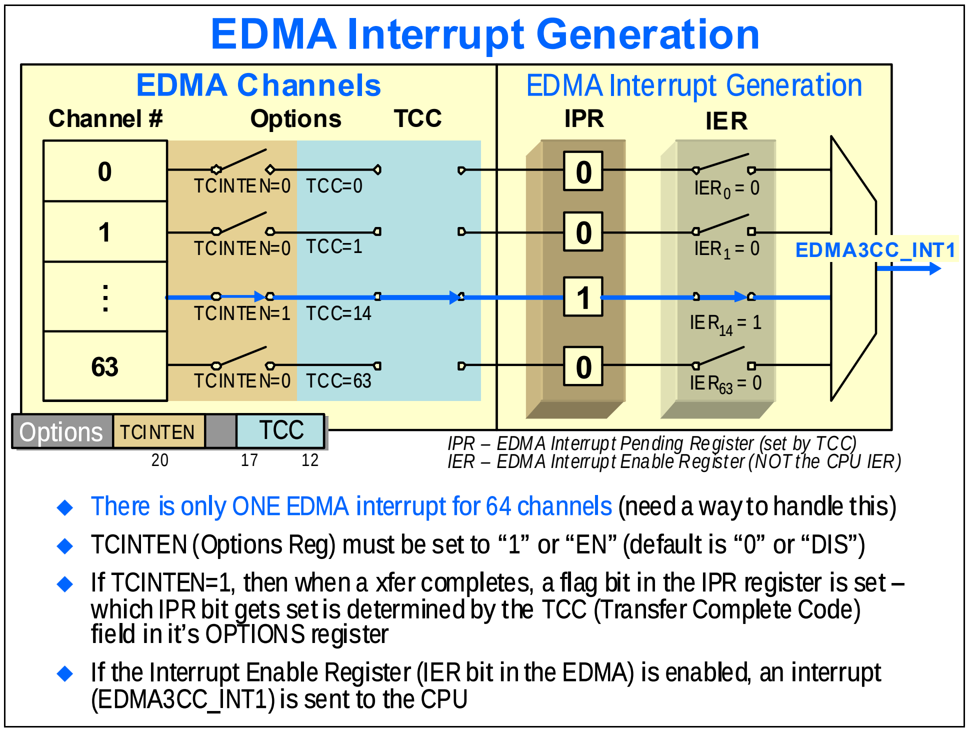
- Need: Generate CPU Interrupt when EDMA completes transfer
- Solution: Turn on EDMA3 interrupt generation
- Notes:
  - User has option to interrupt the CPU after the entire transfer is complete or after every "intermediate" transfer (ACNT for A-sync, A*BCNT for AB-sync).
  - User can respond to the interrupt in several ways:
    - by setting up a hardware interrupt (HWI_INTx) to call an ISR/handler
    - by polling the EDMA interrupt pending register (IPR) – best done using the _check() or _wait() LLD functions to (one-shot test or poll, resp.) the IPR bit

How is the EDMA interrupt generated?

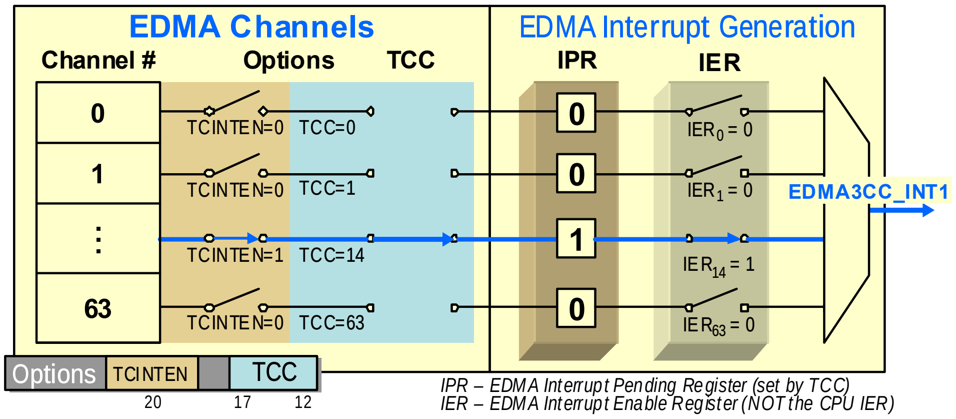
Our goal here is not to discuss everything you need to know about CPU interrupts – that's a topic that takes some time to understand. However, the plan is to show you the overall stream of events to that you understand the big picture. More study might be required to grasp all of the details necessary to correctly program interrupts on your device.

We are now moving on to the "A" or "Action" part of the above diagram. Do you want to interrupt the CPU when the transfer is complete? If so, the EDMA and the LLD APIs support this. You have a choice of either interrupting the CPU after every ACNT bytes is finished in an ACNT * BCNT (AB-sync) transfer (this is called intermediate interrupts) or you can wait until the entire transfer is complete to generate an interrupt.

In the code review, we will examine the APIs to set up interrupts in the EDMA.



EDMA Interrupt Generation



- There is only ONE EDMA interrupt for 64 channels (need a way to handle this)
- TCINTEN (Options Reg) must be set to "1" or "EN" (default is "0" or "DIS")
- If TCINTEN=1, then when a xfer completes, a flag bit in the IPR register is set – which IPR bit gets set is determined by the TCC (Transfer Complete Code) field in it's OPTIONS register
- If the Interrupt Enable Register (IER bit in the EDMA) is enabled, an interrupt (EDMA3CC_INT1) is sent to the CPU

This picture shows how EDMA interrupts are generated. Even though you have 64 or more channels that could cause an interrupt, there is only one EDMA interrupt for all channels. The EDMA interrupt dispatcher will poll the IPR bits to determine which channel fired and call the appropriate interrupt service routine.

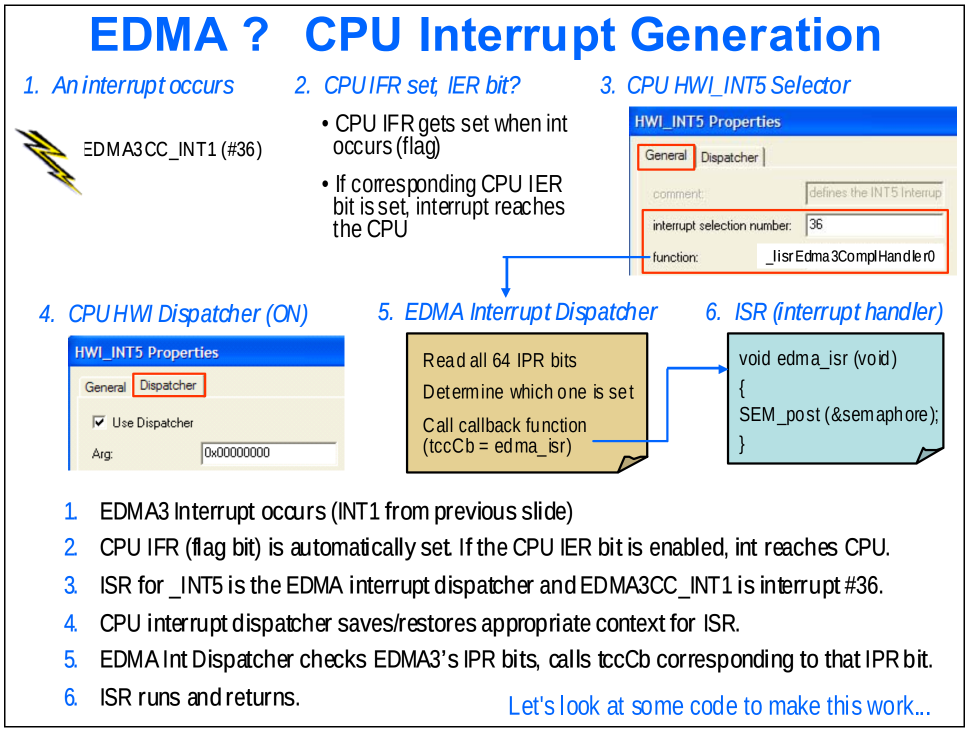
This is the first time we've seen the OPTions register. This register contains several bits that control the behavior of the transfer – including interrupts. The TCINTEN bit determines whether the IPR (interrupt pending register) will be set or not. The IPR is a flag that operates similar to the IFR bit on the CPU. Which IPR bit is set is determined by the TCC value for that transfer configuration. Most of the time, the channel # and TCC are the same, but that is not always the case.

So, when a channel completes (or if intermediate interrupts are selected), it will set the corresponding IPR bit (based on the TCC value in the configuration). If the EDMA IER bit is set as well, this generates an interrupt to the CPU.

The interrupt may not be taken due to other factors (it might be masked or all CPU interrupts are turned off). However, assuming the CPU interrupts are set up properly, this signal will interrupt the CPU.

This slide discusses how an EDMA interrupt is generated. Next, we'll provide a brief overview of what happens after this interrupt fires and how the CPU responds to it….

EDMA3 → CPU Interrupt Generation



EDMA ? CPU Interrupt Generation

1. An interrupt occurs
2. CPU IFR set, IER bit?
3. CPU HWI_INT5 Selector

EDMA3CC_INT1 (#36)

- CPU IFR gets set when int occurs (flag)
- If corresponding CPU IER bit is set, interrupt reaches the CPU

HWI_INT5 Properties — General | Dispatcher — comment: defines the INT5 Interrup — interrupt selection number: 36 — function: _lisrEdma3ComplHandler0

4. CPU HWI Dispatcher (ON)
5. EDMA Interrupt Dispatcher
6. ISR (interrupt handler)

HWI_INT5 Properties — General | Dispatcher — ☑ Use Dispatcher — Arg: 0x00000000

Read all 64 IPR bits
Determine which one is set
Call callback function (tccCb = edma_isr)

```
void edma_isr (void)
{
SEM_post (&semaphore);
}
```

1. EDMA3 Interrupt occurs (INT1 from previous slide)
2. CPU IFR (flag bit) is automatically set. If the CPU IER bit is enabled, int reaches CPU.
3. ISR for _INT5 is the EDMA interrupt dispatcher and EDMA3CC_INT1 is interrupt #36.
4. CPU interrupt dispatcher saves/restores appropriate context for ISR.
5. EDMA Int Dispatcher checks EDMA3's IPR bits, calls tccCb corresponding to that IPR bit.
6. ISR runs and returns.

Let's look at some code to make this work...

This slide shows the events that take place when an interrupt occurs and how the CPU processes that interrupt.

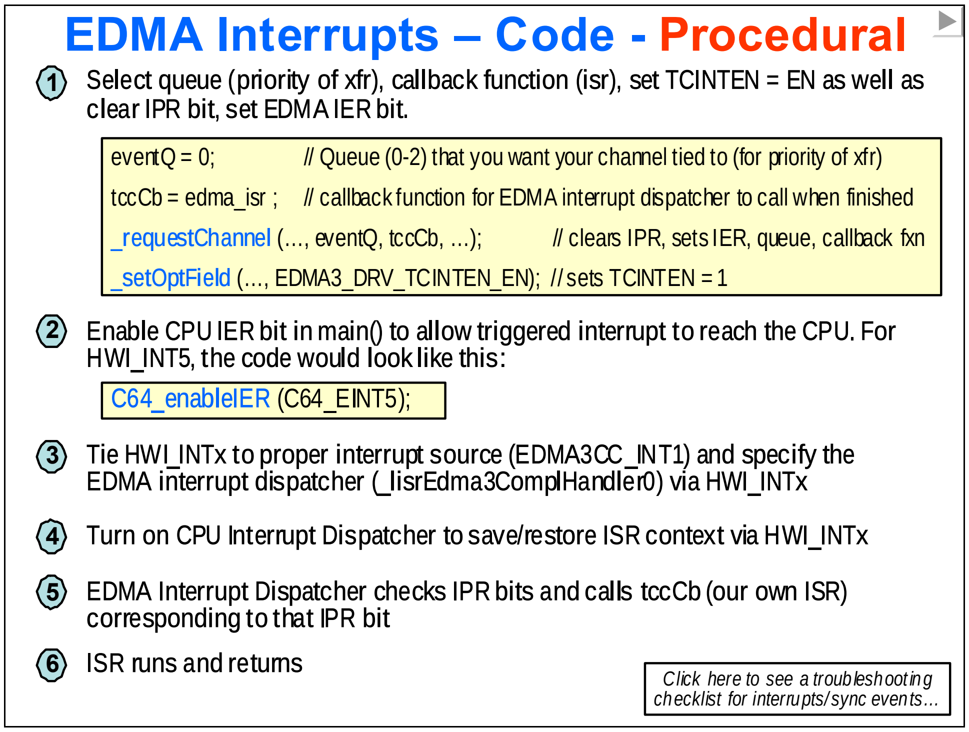
First, the interrupt occurs (i.e. the EDMA generates an interrupt). When this occurs, the CPU automatically sets a flag bit to indicate an interrupt fired. If the corresponding CPU IER bit is set, the ISR is called.

With most interrupts, the ISR is called, context is saved and the interrupt runs. However, using the EDMA, we must determine WHICH channel triggered the interrupt (it could be one of 64 channels). So, in step 3 above, you can see that we are calling an interrupt handler to read the IPR bits to determine which bit was set and then call the appropriate ISR for that channel.

In the LLD, you need to set up a "callback function" for the channel you are using. The LLD will set up a table of functions that correspond to specific IPR bits and match the IPR bit found with the function you desire. From a programmer's perspective, all you need to do is specify the callback function and the LLD does the rest. The only other piece you have to program is the name of the handler shown in step 3 above. This is the LLD interrupt handler function name and is required for processing any EDMA interrupts.

Next, we'll examine the LLD code to generate an EDMA interrupt…

EDMA Interrupts – Code Details



EDMA Interrupts – Code - Procedural

1. Select queue (priority of xfr), callback function (isr), set TCINTEN = EN as well as clear IPR bit, set EDMA IER bit.

```
eventQ = 0;            // Queue (0-2) that you want your channel tied to (for priority of xfr)
tccCb = edma_isr ;     // callback function for EDMA interrupt dispatcher to call when finished
_requestChannel (..., eventQ, tccCb, ...);        // clears IPR, sets IER, queue, callback fxn
_setOptField (..., EDMA3_DRV_TCINTEN_EN);  // sets TCINTEN = 1
```

2. Enable CPU IER bit in main() to allow triggered interrupt to reach the CPU. For HWI_INT5, the code would look like this:

```
C64_enableIER (C64_EINT5);
```

3. Tie HWI_INTx to proper interrupt source (EDMA3CC_INT1) and specify the EDMA interrupt dispatcher (_lisrEdma3ComplHandler0) via HWI_INTx
4. Turn on CPU Interrupt Dispatcher to save/restore ISR context via HWI_INTx
5. EDMA Interrupt Dispatcher checks IPR bits and calls tccCb (our own ISR) corresponding to that IPR bit
6. ISR runs and returns

Click here to see a troubleshooting checklist for interrupts/sync events...

The above shows the procedure for generating interrupts with the EDMA including setting up the CPU interrupts.

In the _requestChannel function, two of the parameters are the queue (there are 2-4 of them on each EDMA) and the callback function (tccCb). As stated before, the callback function is the ISR you want to run when this channel completes its transfer. The interrupt handler will initialize a function table with the specified callback function associated with the TCC specified (which defines the specific IPR bit to be set).

The programmer must also turn on the TCINTEN bit via the _setOptField API as shown. This allows this specific channel to set the corresponding IPR bit properly.

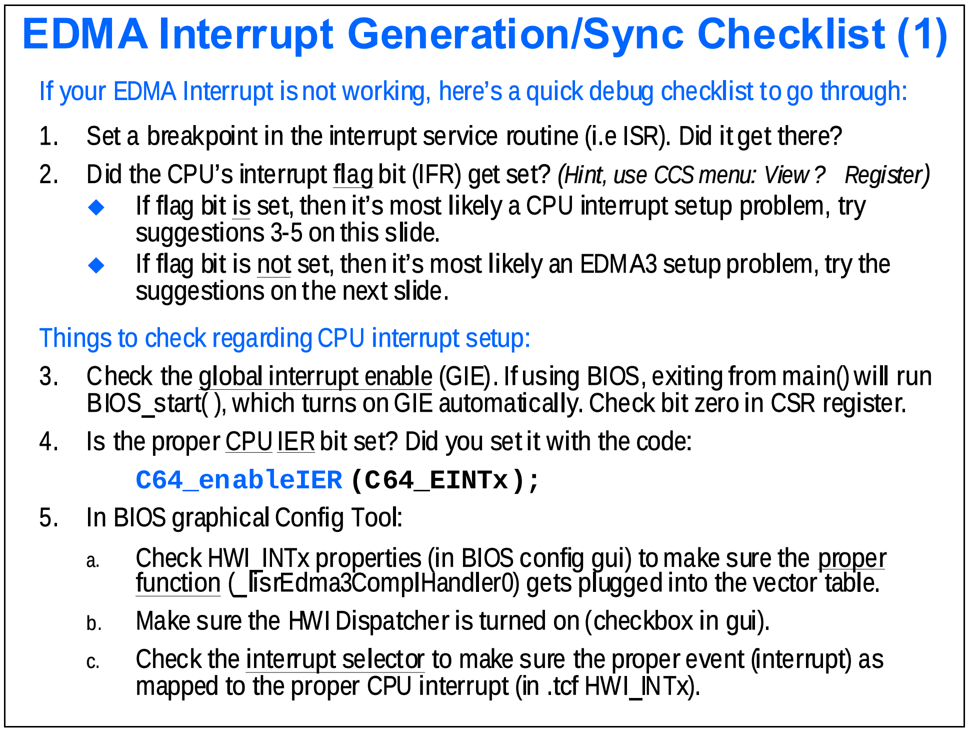
The chosen CPU interrupt (INT5 in the example above) must be enabled as well. Here, we are enabling INT5 (the range is INT4-15 for user interrupts).

In the HWI (.tcf file), we have chosen to tie the interrupt source (EDMA3CC_INT1, region 1 EDMA interrupt) to INT5 of the CPU and set the function to the EDMA interrupt handler function. In this dialogue box, we’ve also turned on the interrupt dispatcher to save/restore context.

The interrupt handler then checks the IPR bits and calls the specified callback function for that IPR bit. The ISR runs and then returns.

EDMA3 → CPU Interrupt Generation

If your interrupt is not working properly, try steps 1-2 below to help indicate where the problem lies. These two slides contain very handy info to help you debug your interrupt problem.



EDMA Interrupt Generation/Sync Checklist (1)

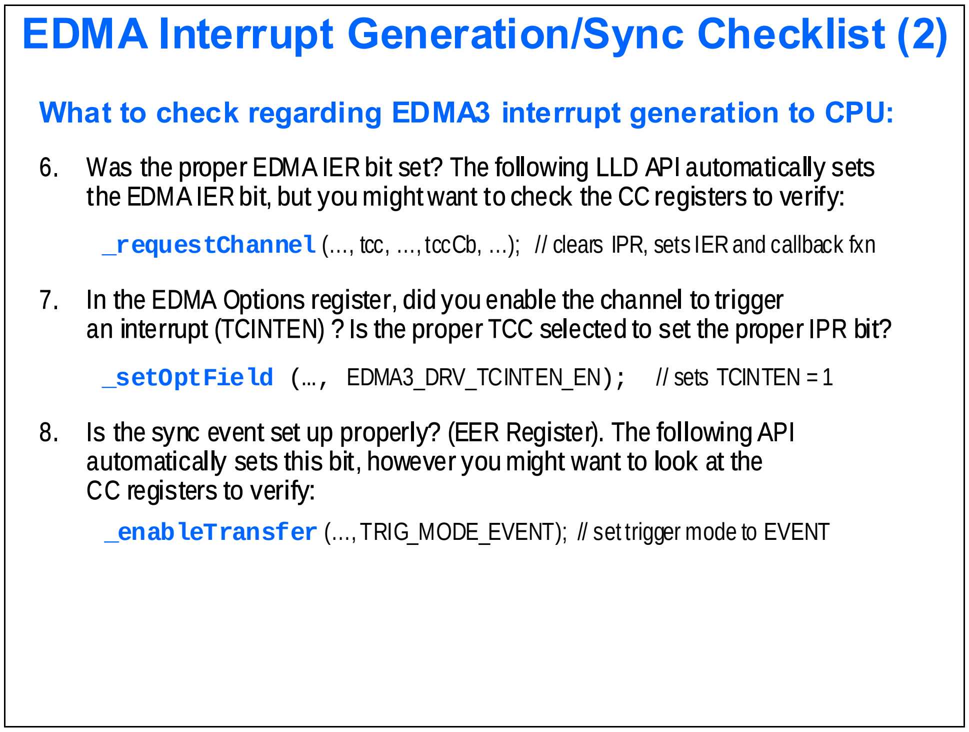
If your EDMA Interrupt is not working, here's a quick debug checklist to go through:

1. Set a breakpoint in the interrupt service routine (i.e ISR). Did it get there?
2. Did the CPU's interrupt flag bit (IFR) get set? *(Hint, use CCS menu: View ? Register)*
   - ◆ If flag bit is set, then it's most likely a CPU interrupt setup problem, try suggestions 3-5 on this slide.
   - ◆ If flag bit is not set, then it's most likely an EDMA3 setup problem, try the suggestions on the next slide.

Things to check regarding CPU interrupt setup:

3. Check the global interrupt enable (GIE). If using BIOS, exiting from main() will run BIOS_start( ), which turns on GIE automatically. Check bit zero in CSR register.
4. Is the proper CPU IER bit set? Did you set it with the code:

   `C64_enableIER (C64_EINTx);`
5. In BIOS graphical Config Tool:
   - a. Check HWI_INTx properties (in BIOS config gui) to make sure the proper function (_lisrEdma3ComplHandler0) gets plugged into the vector table.
   - b. Make sure the HWI Dispatcher is turned on (checkbox in gui).
   - c. Check the interrupt selector to make sure the proper event (interrupt) as mapped to the proper CPU interrupt (in .tcf HWI_INTx).



EDMA Interrupt Generation/Sync Checklist (2)

What to check regarding EDMA3 interrupt generation to CPU:

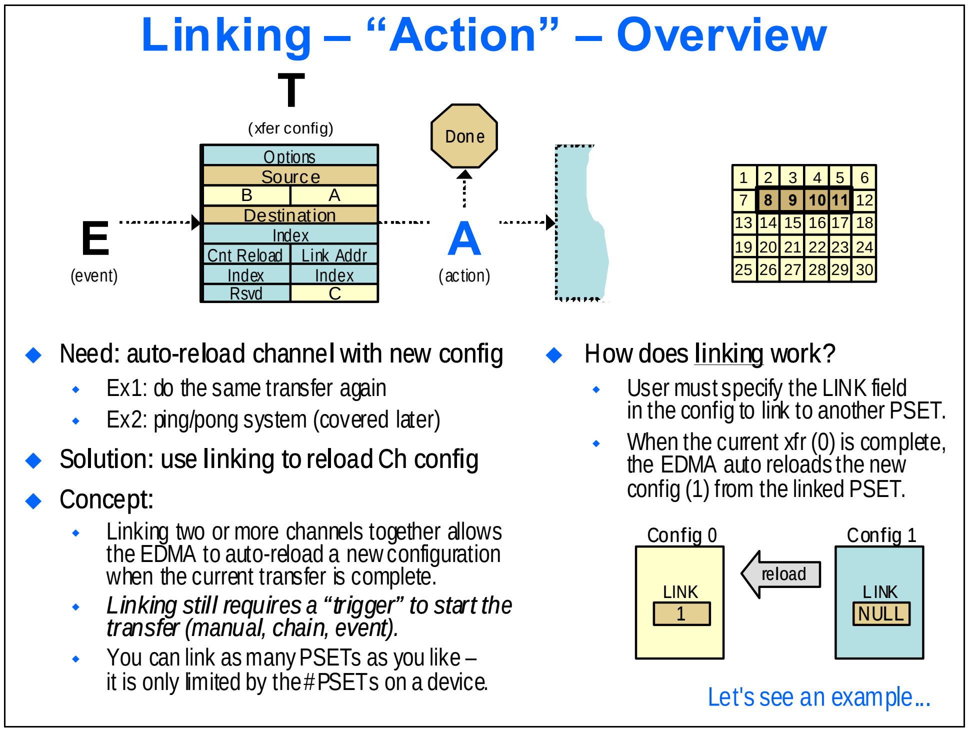
6. Was the proper EDMA IER bit set? The following LLD API automatically sets the EDMA IER bit, but you might want to check the CC registers to verify:

   `_requestChannel (…, tcc, …, tccCb, …);  // clears IPR, sets IER and callback fxn`
7. In the EDMA Options register, did you enable the channel to trigger an interrupt (TCINTEN) ? Is the proper TCC selected to set the proper IPR bit?

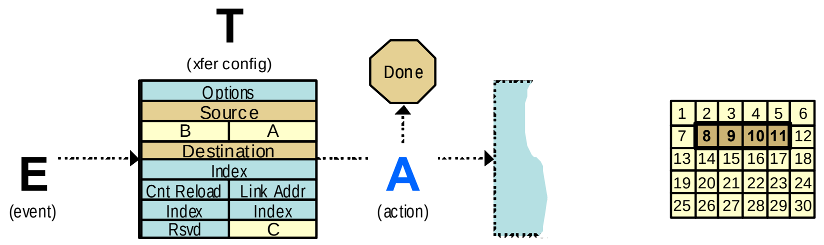
   `_setOptField (…, EDMA3_DRV_TCINTEN_EN);  // sets TCINTEN = 1`
8. Is the sync event set up properly? (EER Register). The following API automatically sets this bit, however you might want to look at the CC registers to verify:

   `_enableTransfer (…, TRIG_MODE_EVENT);  // set trigger mode to EVENT`

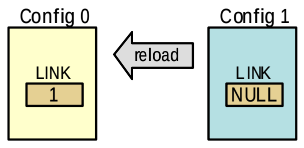
Action – Linking



Linking – “Action” – Overview



- Need: auto-reload channel with new config
  - Ex1: do the same transfer again
  - Ex2: ping/pong system (covered later)
- Solution: use linking to reload Ch config
- Concept:
  - Linking two or more channels together allows the EDMA to auto-reload a new configuration when the current transfer is complete.
  - *Linking still requires a “trigger” to start the transfer (manual, chain, event).*
  - You can link as many PSETs as you like – it is only limited by the # PSETs on a device.
- How does linking work?
  - User must specify the LINK field in the config to link to another PSET.
  - When the current xfr (0) is complete, the EDMA auto reloads the new config (1) from the linked PSET.



Let's see an example...

Your device has any where from 128-256 PSETs or sets of configuration parameters (8 32-bit registers). 64 channels (on most processors) can be active at any one time. So, this leaves at least another 64 “reload sets” or other configurations you can use to initialize or reload active channels. The act of reloading a channel from a PSET is called “linking”. Each configuration has a “link” address field (16 bits) that can point to another channel configuration.

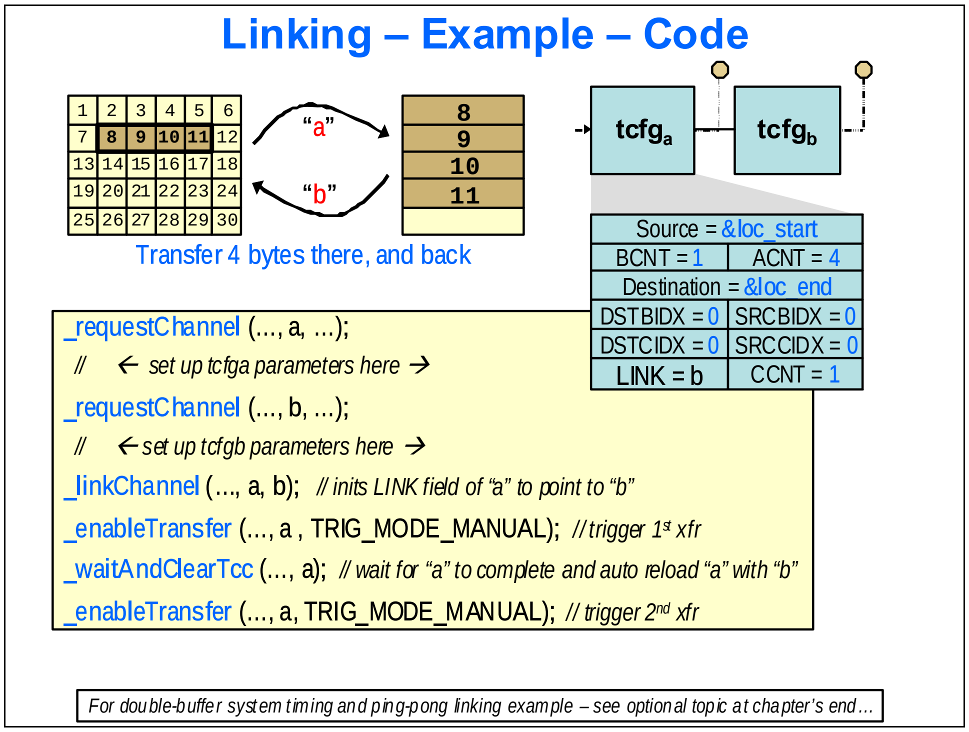
Linking can be performed in order to reload the same transfer or a completely different transfer. Often, systems employ a ping-pong buffering system. In this case, when the ping transfer is done, the EDMA can auto reload the pong transfer and on and on it goes.

Don’t confuse linking and chaining (covered later) – they are completely different. Linking reloads an active channel with a new configuration. Linking is NOT an event that will cause a transfer to start. You still must provide an event – either a manual start or via a synchronization event.

In the example on the slide above, you can see that channel “0” has its link field set to “1”. When 0’s transfer is done, the EDMA will copy the configuration from “1” into “0” and then wait for a trigger to start the next transfer.

Next, we’ll look at the LLD code to configure linking on the EDMA…

Linking – Code Details



Linking – Example – Code

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

“a”
“b”
8
9
10
11
tcfg$_a$
tcfg$_b$

Transfer 4 bytes there, and back

Source = &loc_start	
BCNT = 1	ACNT = 4
Destination = &loc_end	
DSTBIDX = 0	SRCBIDX = 0
DSTCIDX = 0	SRCCIDX = 0
LINK = b	CCNT = 1

```
_requestChannel (..., a, ...);
  //    ← set up tcfga parameters here →
_requestChannel (..., b, ...);
  //    ← set up tcfgb parameters here →
_linkChannel (..., a, b);  // inits LINK field of “a” to point to “b”
_enableTransfer (..., a , TRIG_MODE_MANUAL);  // trigger 1st xfr
_waitAndClearTcc (..., a);  // wait for “a” to complete and auto reload “a” with “b”
_enableTransfer (..., a, TRIG_MODE_MANUAL);  // trigger 2nd xfr
```

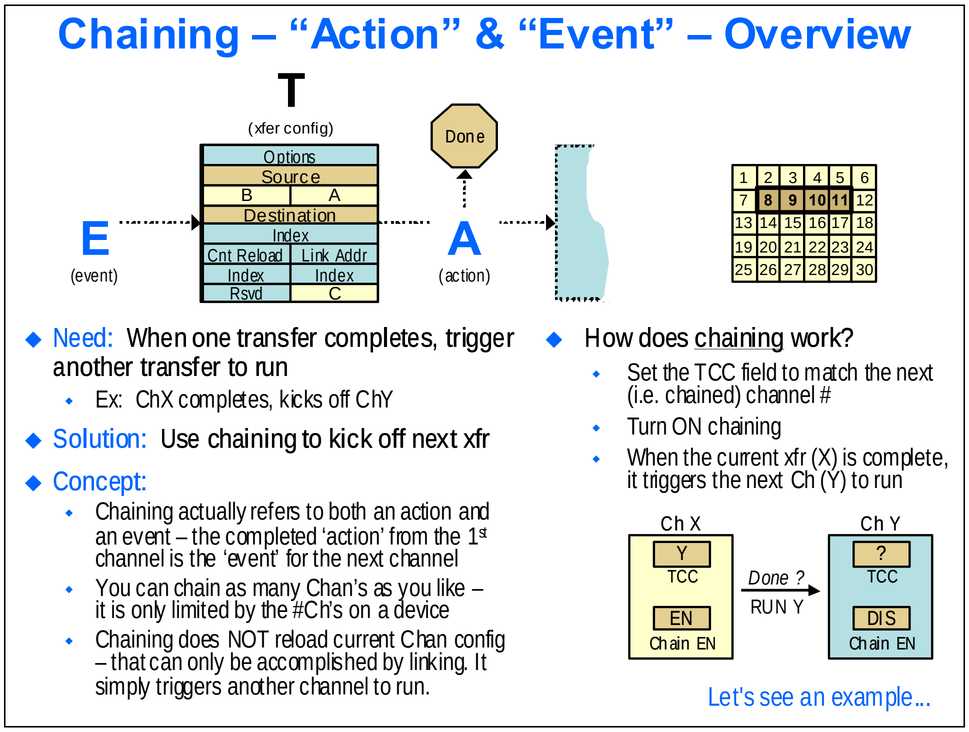
For double-buffer system timing and ping-pong linking example – see optional topic at chapter’s end…

The example above shows a horizontal line being transferred to memory and then back again. So, we have channel “a”, which is an active channel PARAM set with a configuration that will transfer the line from the 2D array to memory. Configuration “b” is simply a reload PARAM set (non active) which holds the configuration parameters for the transfer from memory back to the 2D array. The best way to view linking is that you have an “active” channel (a) and a “reload” channel (b). The reload channel (b) is a configuration that is copied to the active channel and then that channel is triggered again. So, we don’t actually trigger “b” to run as an active channel. It is only used as a reload of “a”. If you review the LLD example on linking, you’ll notice that the type of channel requested for “a” is a different type than channel “b”. (More details on this if you open up the code examples).

When channel “a” completes, it reloads its configuration registers with the “b” configuration and then waits for a trigger to start the transfer. In LLD code, we first perform a _requestChannel( ) for the active channel “a” and set up its parameters. Then, we do the same for the reload PARAM set “b”. _linkChannel( ) actually pokes the address of the “b” into “a”s link field. The first _enableTransfer kicks off the first “a” transfer. _waitAndClearTcc waits for “a” to complete. When “a” does complete, it auto reloads or “links” to “b”. Actually, the contents of the “b” configuration are copied into “a”.

The second _enableTransfer( ) triggers “a” to run again, but now it is configured with “b”s parameters instead – thus performing the transfer from memory back to the 2D array. Notice that the “mode” of the transfer is “TRIG_MODE_MANUAL” which is a manual start vs. any type of event synchronization.

Action/Event - Chaining



Chaining – "Action" & "Event" – Overview

- **Need:** When one transfer completes, trigger another transfer to run
  - Ex: ChX completes, kicks off ChY
- **Solution:** Use chaining to kick off next xfr
- **Concept:**
  - Chaining actually refers to both an action and an event – the completed 'action' from the 1st channel is the 'event' for the next channel
  - You can chain as many Chan's as you like – it is only limited by the #Ch's on a device
  - Chaining does NOT reload current Chan config – that can only be accomplished by linking. It simply triggers another channel to run.
- How does chaining work?
  - Set the TCC field to match the next (i.e. chained) channel #
  - Turn ON chaining
  - When the current xfr (X) is complete, it triggers the next Ch (Y) to run

Let's see an example...

When you need the completion of one transfer to trigger another channel (or itself) to execute a transfer, use chaining. It is not the most popular feature of the EDMA, however some systems cannot live without it.

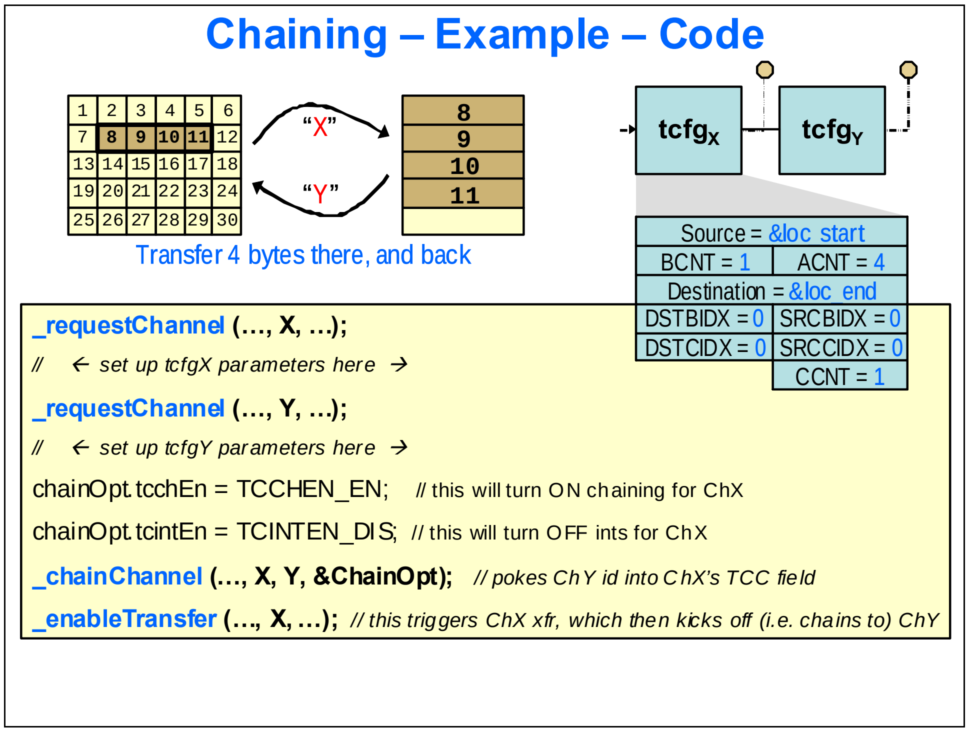
If chaining is enabled for channel X (as shown in the diagram) and channel X's TCC field is set to Y, when X completes, it will trigger Y to run.

Chaining refers to both an "Action" and an "Event". The action that occurs when Ch X's transfer completes (when chaining is enabled) is that it triggers another transfer (Y) to run. This is also an event synchronization for Ch Y.

Keep in mind that chaining does NOT reload or configure any channel parameters like linking does. Again, these two features are completely separate. However, they can be used in combination with each other (i.e. Ch X could trigger Y to run – chaining – and Ch X could link to Ch Z and reload its registers from Ch Z when the transfer completes).

To enable chaining, you need to do three things: (1) enable chaining in the channel that will CHAIN to another channel; (2) set the TCC value of the first channel to match the second channel; (3) initialize (configure) both channel configurations before the first one is triggered.

Chaining – Code Details



Chaining – Example – Code

Transfer 4 bytes there, and back

Source = &loc_start	
BCNT = 1	ACNT = 4
Destination = &loc_end	
DSTBIDX = 0	SRCBIDX = 0
DSTCIDX = 0	SRCCIDX = 0
	CCNT = 1

```
_requestChannel (…, X, …);
//    ← set up tcfgX parameters here →
_requestChannel (…, Y, …);
//    ← set up tcfgY parameters here →
chainOpt.tcchEn = TCCHEN_EN;     // this will turn ON chaining for ChX
chainOpt.tcintEn = TCINTEN_DIS;  // this will turn OFF ints for ChX
_chainChannel (…, X, Y, &ChainOpt);    // pokes ChY id into ChX's TCC field
_enableTransfer (…, X, …);  // this triggers ChX xfr, which then kicks off (i.e. chains to) ChY
```

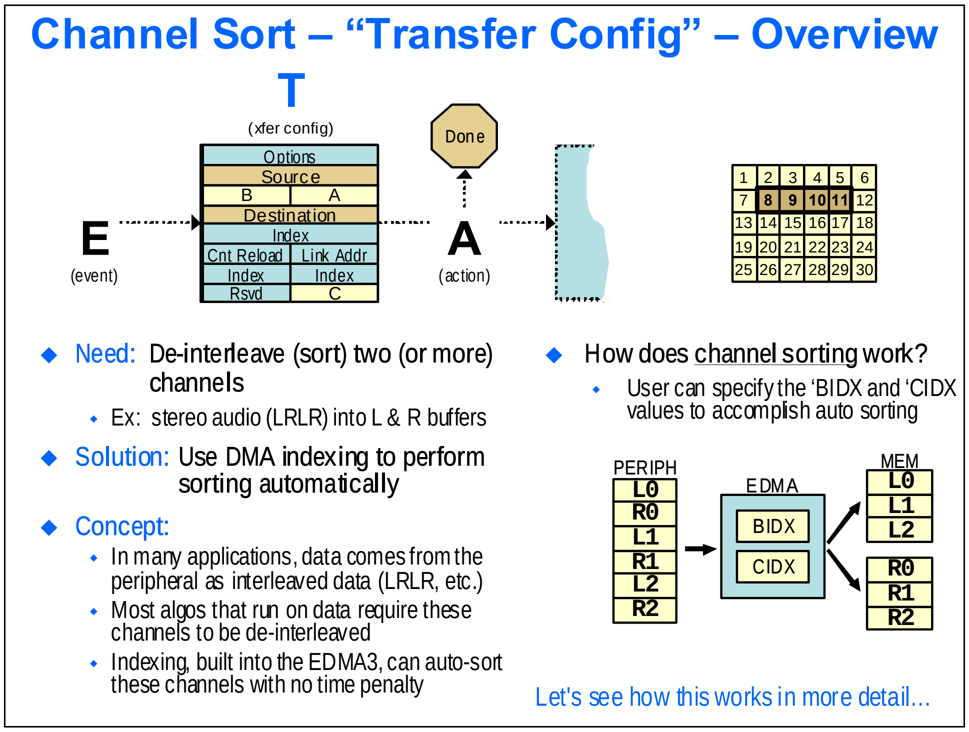
In this example, when Ch X finishes its transfer, it will trigger Ch Y to run. As before, you must request two PARAM sets – one for X and one for Y. In this case, both channels will be “active” channels.

In order for chaining to work, you must enable chaining in the configuration of the channel that “chains” to another channel (so in this example, we must turn on chaining for Ch X). You can also modify other OPTions register bits (e.g. interrupts) at the same time as shown in the example.

Using the _chainChannel( ) API, the channel id for the chained channel (Y in this case) is poked into Ch X’s TCC field.

Then, channel X is triggered with _enableTransfer. When Ch X finishes, it triggers Ch Y to run.

Transfer Config – Channel Sorting


T
(xfer config)
Options
Source
B
A
Destination
Index
Cnt Reload
Link Addr
Index
Index
Rsvd
C
E
(event)
Done
A
(action)
PERIPH
L0
R0
L1
R1
L2
R2
EDMA
BIDX
CIDX
MEM
L0
L1
L2
R0
R1
R2

Channel Sort – "Transfer Config" – Overview

- Need: De-interleave (sort) two (or more) channels
  - Ex: stereo audio (LRLR) into L & R buffers
- Solution: Use DMA indexing to perform sorting automatically
- Concept:
  - In many applications, data comes from the peripheral as interleaved data (LRLR, etc.)
  - Most algos that run on data require these channels to be de-interleaved
  - Indexing, built into the EDMA3, can auto-sort these channels with no time penalty
- How does channel sorting work?
  - User can specify the 'BIDX and 'CIDX values to accomplish auto sorting

Let's see how this works in more detail…

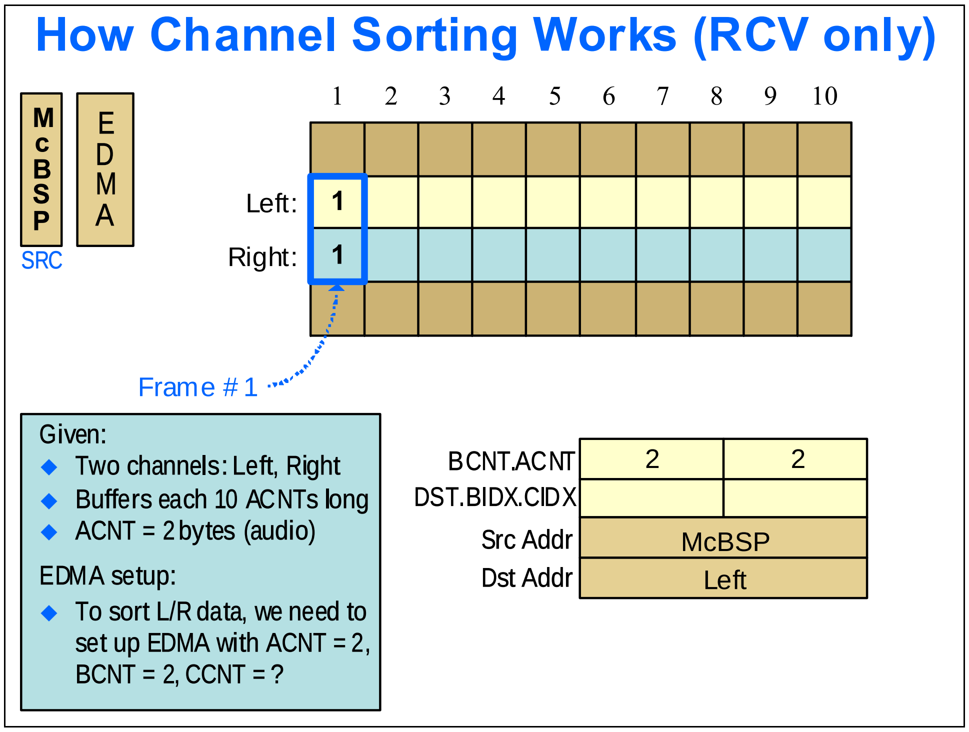
Channel sorting is the process of grouping together like data – for example, the Left and Right channels of stereo audio. The audio comes through the serial port L0, R0, L1, R1…and what you want to do is "sort" this data into separate "channels" such as L0, L1, L2, etc. and R0/R1/R2…and so forth. The indexing capability of the EDMA allows you to easily channel sort with NO overhead. The picture near the bottom right of this slide tells the story well.

Through a combination of 'BIDX and 'CIDX, you can easily sort data into 2 or more channels of information.

The following slides take you through the math and concepts involved with channel sorting.

The LLD code to perform channel sorting is shown in one of our examples. It is basically a wise use of the indexing capabilities of the EDMA.

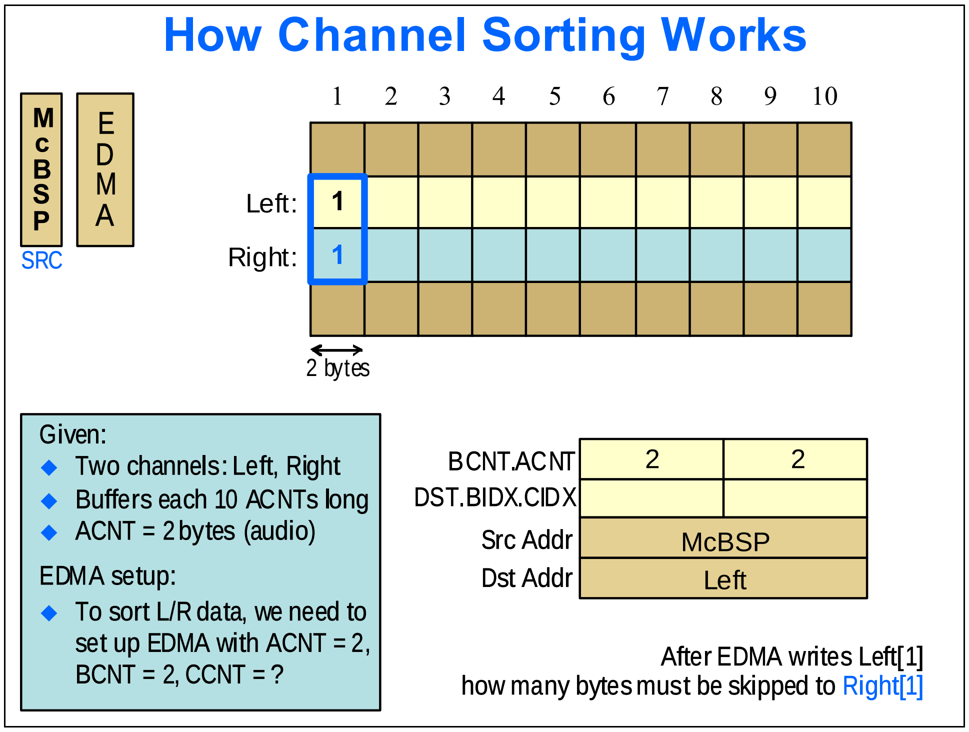
How Channel Sorting Works


How Channel Sorting Works (RCV only)
McBSP
SRC
EDMA
1 2 3 4 5 6 7 8 9 10
Left: 1
Right: 1
Frame # 1
Given:
Two channels: Left, Right
Buffers each 10 ACNTs long
ACNT = 2 bytes (audio)
EDMA setup:
To sort L/R data, we need to set up EDMA with ACNT = 2, BCNT = 2, CCNT = ?
BCNT.ACNT 2 2
DST.BIDX.CIDX
Src Addr McBSP
Dst Addr Left

Assuming that the serial port provides you with data in this format: LRLRLR, the first sample will be L0 and is placed at the top (or left in the diagram above) of "Left". The next value needs to jump the buffer size (in this case 10 16-bit locations) to get to the top of the "Right" channel. Because this is audio, ACNT is 16 bits or 2 bytes. Because we have stereo audio (2 channels), BCNT is also 2 (BCNT is almost always the number of channels you are sorting to).

So, the first L sample is transferred to its spot. Which index is employed to get from the top of the left buffer to the top of the right buffer? When BCNT decrements, 'BIDX is involved. So, after transferring the first L value, we have to jump 10 spots (or 20 bytes) to store the first R value.

What would CCNT be in this application? We have 10 pairs of L/R data, so CCNT = 10. Quickly do the math: ACNT * BCNT * CCNT = 2 * 2 * 10 = 40 bytes. Is that right? 20 samples of 16-bit L and R each – ok, that's 40 bytes. Maybe we're right for once.



How Channel Sorting Works

Given:
- Two channels: Left, Right
- Buffers each 10 ACNTs long
- ACNT = 2 bytes (audio)

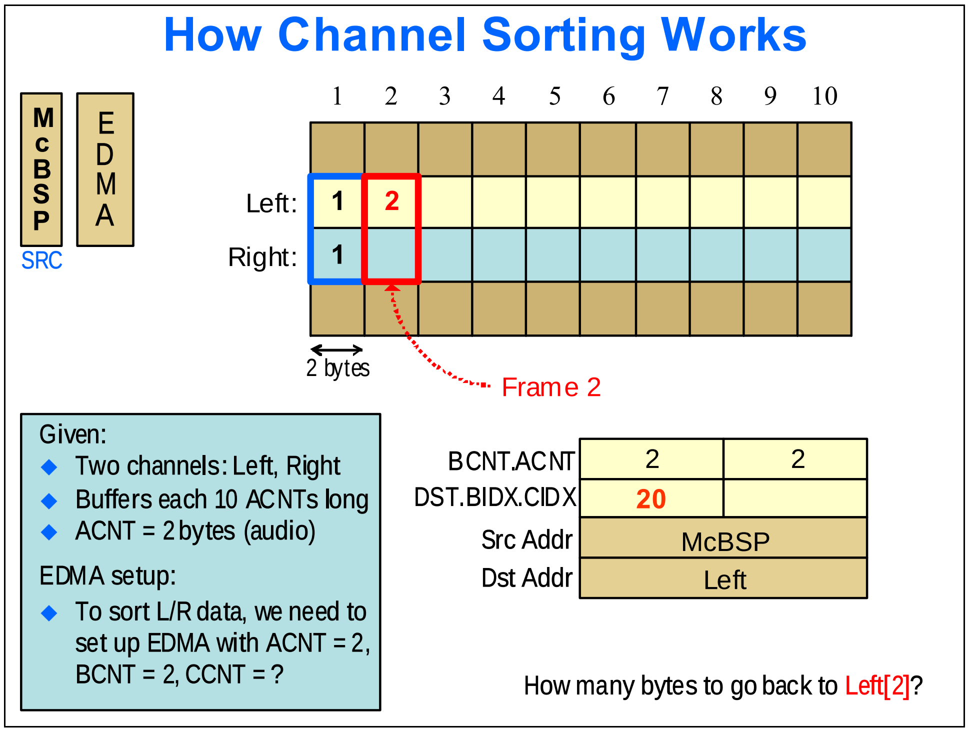
EDMA setup:
- To sort L/R data, we need to set up EDMA with ACNT = 2, BCNT = 2, CCNT = ?

BCNT.ACNT	2	2
DST.BIDX.CIDX		
Src Addr	McBSP	
Dst Addr	Left	

After EDMA writes Left[1]
how many bytes must be skipped to Right[1]

Ok, we’ve already covered the news on this slide. We know we need to skip 20 bytes from the first L to the first R. Geez these people are slow… ☺

Good question below…how do we get BACK to the left buffer for the L2 transfer?



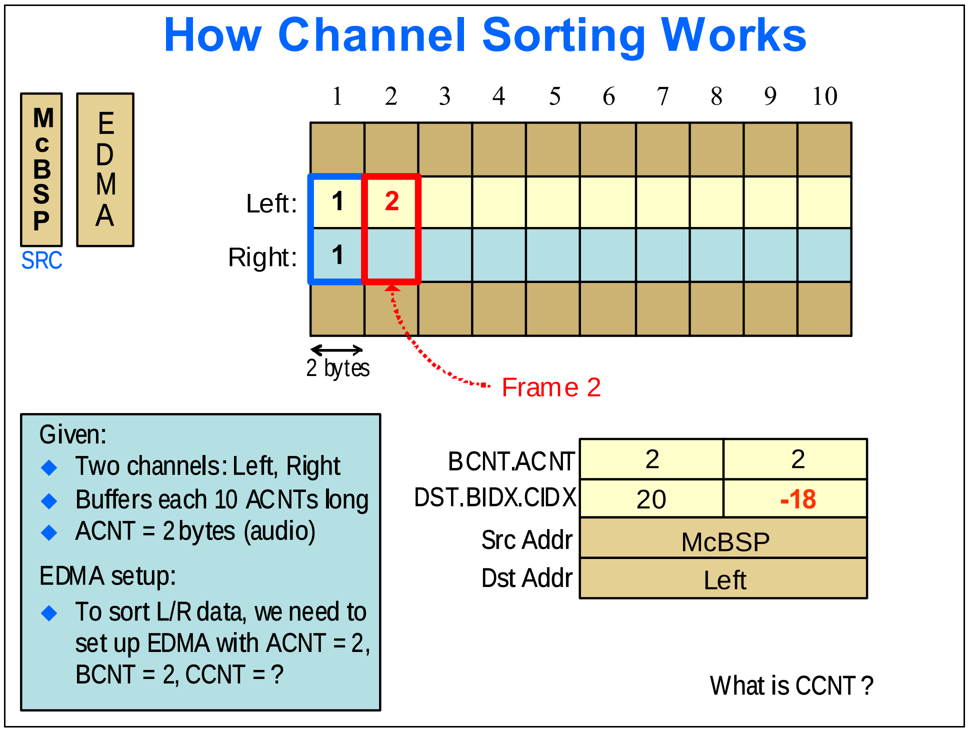
How Channel Sorting Works

Given:
- Two channels: Left, Right
- Buffers each 10 ACNTs long
- ACNT = 2 bytes (audio)

EDMA setup:
- To sort L/R data, we need to set up EDMA with ACNT = 2, BCNT = 2, CCNT = ?

BCNT.ACNT	2	2
DST.BIDX.CIDX	20	
Src Addr	McBSP	
Dst Addr	Left	

How many bytes to go back to Left[2]?



How Channel Sorting Works

Given:

- Two channels: Left, Right
- Buffers each 10 ACNTs long
- ACNT = 2 bytes (audio)

EDMA setup:

- To sort L/R data, we need to set up EDMA with ACNT = 2, BCNT = 2, CCNT = ?

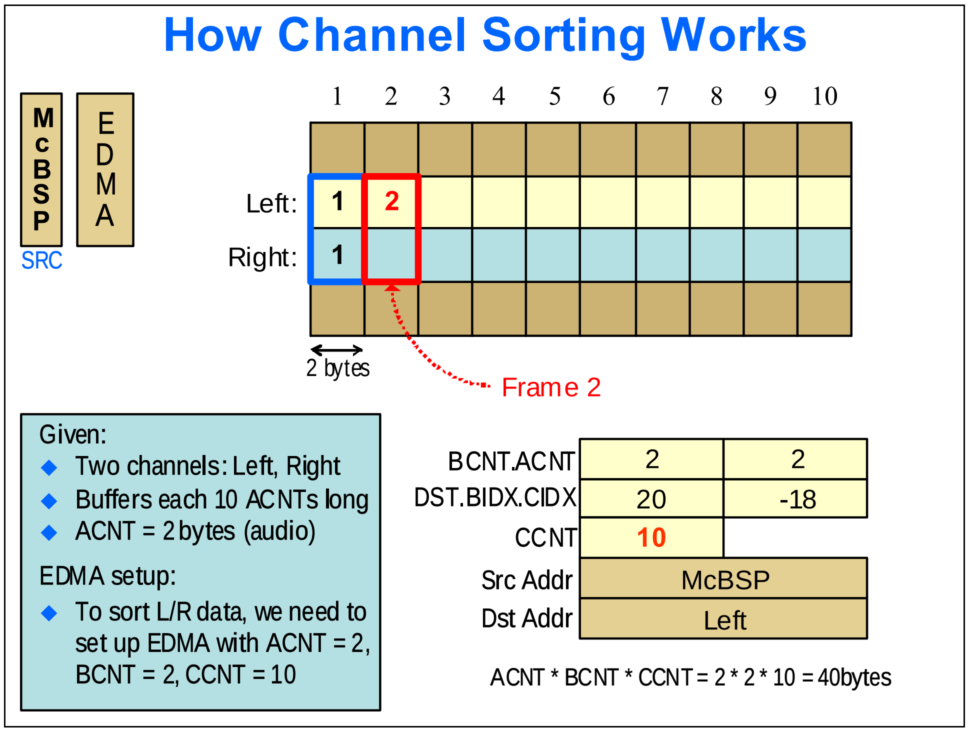
BCNT.ACNT	2	2
DST.BIDX.CIDX	20	-18
Src Addr	McBSP	
Dst Addr	Left	

What is CCNT ?

Ok, sneaky thought here. We have two indexes: ‘BIDX and ‘CIDX. When BCNT decrements to zero, CCNT gets decremented by one (this is at the point where the R1 value has been transferred in the diagram below. Now, how do we get BACK to L2 placement? When CCNT decrements, ‘CIDX is employed.

If you go back an entire buffer minus one data item (let’s see, buffer is 20 bytes minus one 16-bit value is, uh….., 18 bytes) – going back 18 bytes is a -18 for ‘CIDX. Beautiful. We do that 10 times and our channel sorting (not snorting) is complete.

What is CCNT? We already figured that out – it’s 10. 10 I tell you. Again, these people are S—L—O—W .



How Channel Sorting Works

Given:

- Two channels: Left, Right
- Buffers each 10 ACNTs long
- ACNT = 2 bytes (audio)

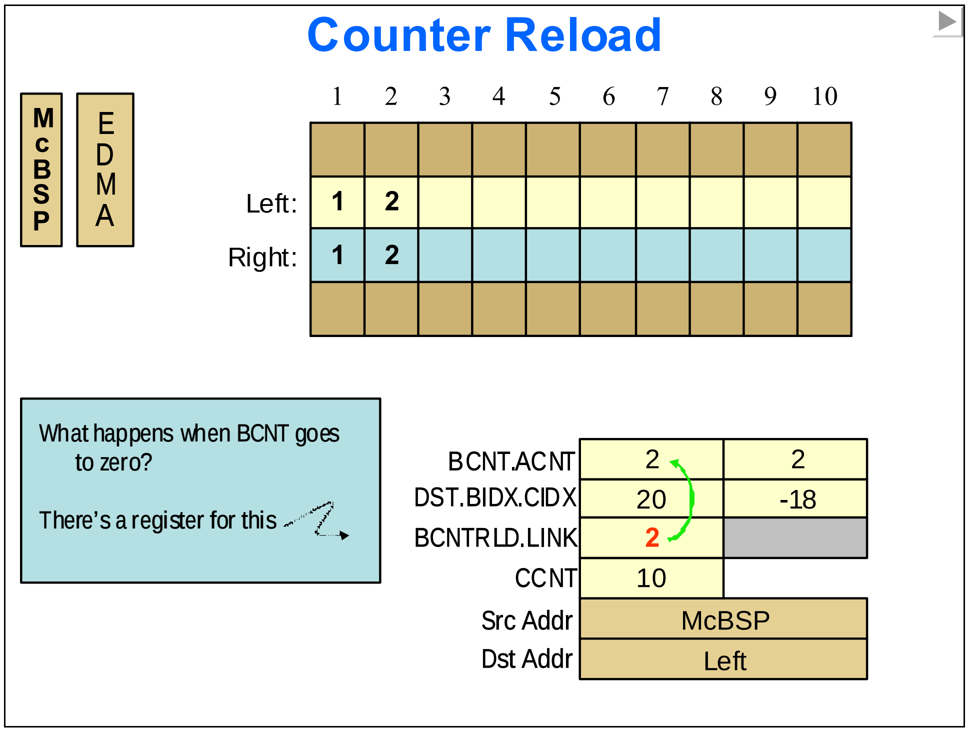
EDMA setup:

- To sort L/R data, we need to set up EDMA with ACNT = 2, BCNT = 2, CCNT = 10

BCNT.ACNT	2	2
DST.BIDX.CIDX	20	-18
CCNT	10	
Src Addr	McBSP	
Dst Addr	Left	

ACNT * BCNT * CCNT = 2 * 2 * 10 = 40bytes

Yep – and we already did that math 4 weeks ago. Then it goes on an on until the xfr is done. Got it. BCNTRLD is also necessary because the transfer controller knows ACNT implicitly, but not BCNT. If you have an A*B transfer and BCNT is greater than one, you have to specify the BCNT RELOAD value (which is just BCNT again).

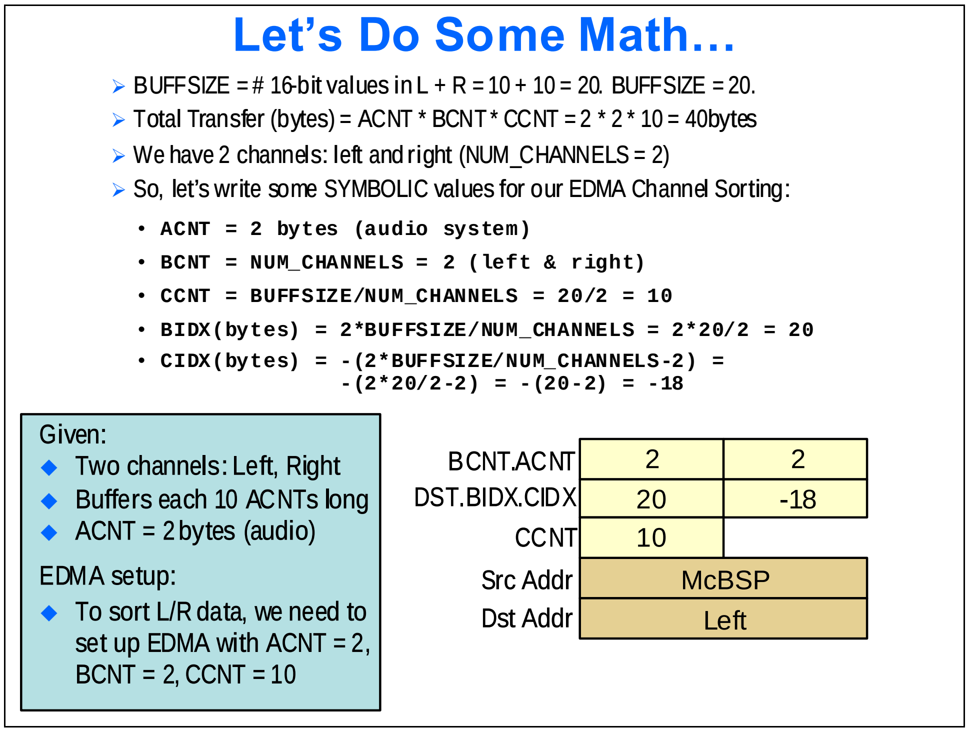


Counter Reload

What happens when BCNT goes to zero?

There's a register for this

BCNT.ACNT	2	2
DST.BIDX.CIDX	20	-18
BCNTRLD.LINK	2	
CCNT	10	
Src Addr	McBSP	
Dst Addr	Left	



Let's Do Some Math…

- BUFFSIZE = # 16-bit values in L + R = 10 + 10 = 20. BUFFSIZE = 20.
- Total Transfer (bytes) = ACNT * BCNT * CCNT = 2 * 2 * 10 = 40bytes
- We have 2 channels: left and right (NUM_CHANNELS = 2)
- So, let's write some SYMBOLIC values for our EDMA Channel Sorting:

```
• ACNT = 2 bytes (audio system)
• BCNT = NUM_CHANNELS = 2 (left & right)
• CCNT = BUFFSIZE/NUM_CHANNELS = 20/2 = 10
• BIDX(bytes) = 2*BUFFSIZE/NUM_CHANNELS = 2*20/2 = 20
• CIDX(bytes) = -(2*BUFFSIZE/NUM_CHANNELS-2) =
                -(2*20/2-2) = -(20-2) = -18
```

Given:
- Two channels: Left, Right
- Buffers each 10 ACNTs long
- ACNT = 2 bytes (audio)

EDMA setup:
- To sort L/R data, we need to set up EDMA with ACNT = 2, BCNT = 2, CCNT = 10

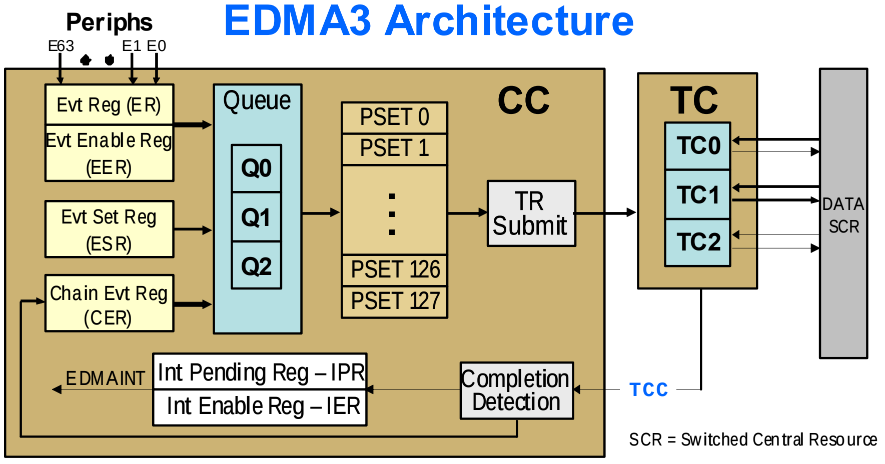
BCNT.ACNT	2	2
DST.BIDX.CIDX	20	-18
CCNT	10	
Src Addr	McBSP	
Dst Addr	Left	

If you're in the mood to do some fancy math with symbols (which is handy in code because it can flex with different needs – i.e. more channels, fewer channels, bigger buffers, etc.), then the slide above is for YOU. Most of the time, it is best to architect your code so that it can move and weave with the wind – so you don't keep changing 15 values in 974 spots in your code every 10 minutes.

This is just an example of how you can make the values symbolic in an N-channel system.

EDMA3 Architecture & Performance Tips

EDMA3 Architecture

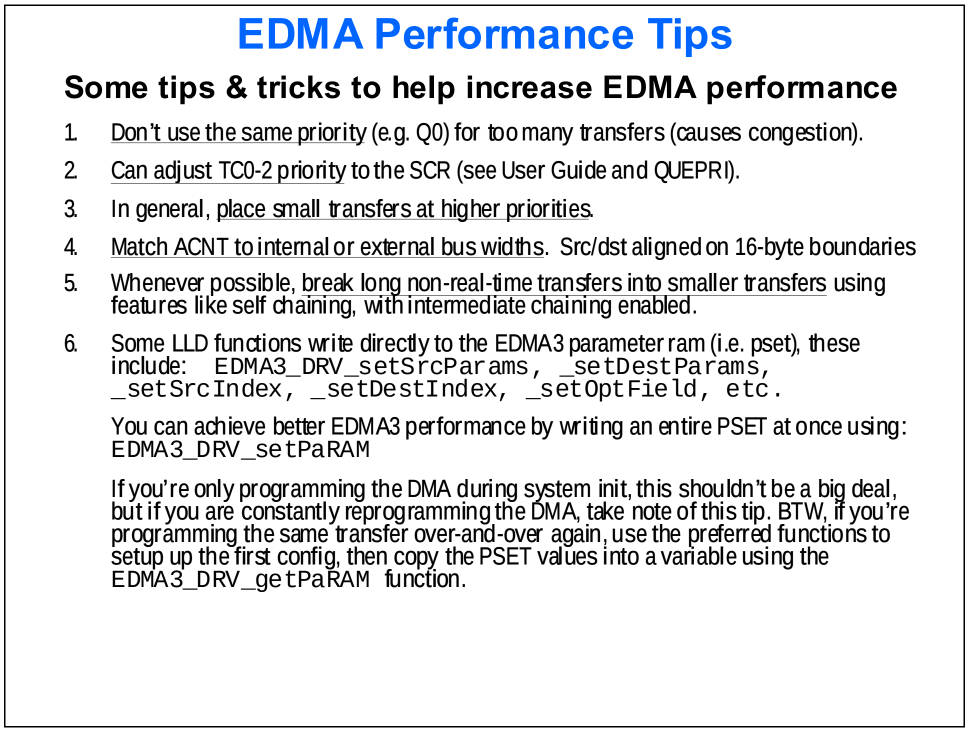


- EDMA consists of two parts: Channel Controller (CC) and Transfer Controller (TC)
- An event (from periph-ER/EER, manual-ESR or via chaining-CER) sends the transfer to 1 of 3 queues (Q0 is mapped to TC0, Q1-TC1, etc.)
- Xfr mapped to 1 of 128 PSETs and submitted to the TC (1 TR – transmit request – per ACNT bytes or “A*B” CNT bytes – based on sync).
- TC performs xfr (read/write) and then sends back a transfer completion code (TCC)
- The EDMA can then interrupt the CPU and/or trigger another transfer

Wow – there is a BUNCH of stuff here. This is the 200 page EDMA3 User Guide on one slide. Let’s start with the two pieces of the EDMA – the CC (Channel Controller) and TC (Transfer Controller). The CC is the “brain” which sets up the transfers, handles priorities and the queues and event triggering. The TC is the “dumb” transfer engine that just does what the CC puts on its plate. It deals with the SCR (Switched Central Resource) bus to contend with other masters for resources.

An event (from the ER register – which flags any events occurring) could cause a transfer configuration to be placed in one of 3 queues (DM6437 has 3 queues, other devices have 2 or 4). The Event Set Register gets set when you do a manual start of a transfer (it ignores everything else). The Chain Evt Register gets set if one channel chains to another. Ok, so one of these caused a transfer to enter one of the queues. That transfer is then mapped to a PSET as shown and part of that PSET is sent to the TC via TR Submit (this is either ACNT bytes or ACNT * BCNT bytes depending on the sync model you chose). Whatever is in Q0 gets placed in TC0, Q1 to TC1, etc.

The TC then reads/writes (copies) values from src to dst and deals with bus issues on the SCR. When the transfer is complete, the TCC value comes back to the CC and sets the proper IPR bit. And we know that we could either poll this bit or set it up to interrupt the CPU automagically.



EDMA Performance Tips

Some tips & tricks to help increase EDMA performance

1. Don't use the same priority (e.g. Q0) for too many transfers (causes congestion).
2. Can adjust TC0-2 priority to the SCR (see User Guide and QUEPRI).
3. In general, place small transfers at higher priorities.
4. Match ACNT to internal or external bus widths. Src/dst aligned on 16-byte boundaries
5. Whenever possible, break long non-real-time transfers into smaller transfers using features like self chaining, with intermediate chaining enabled.
6. Some LLD functions write directly to the EDMA3 parameter ram (i.e. pset), these include: `EDMA3_DRV_setSrcParams, _setDestParams, _setSrcIndex, _setDestIndex, _setOptField, etc.`

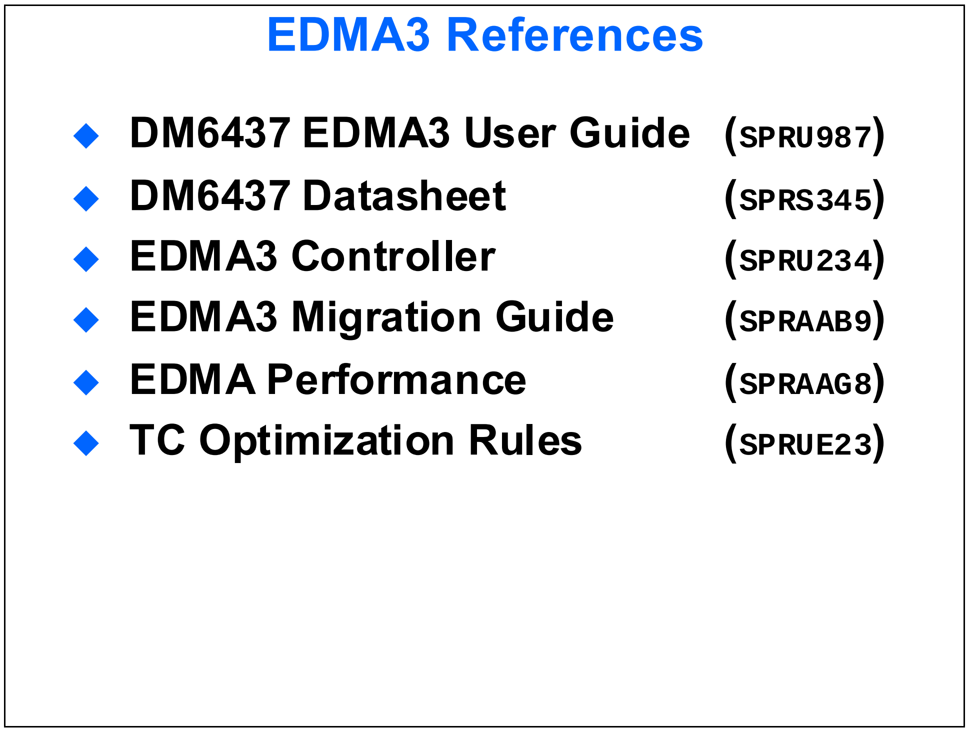
   You can achieve better EDMA3 performance by writing an entire PSET at once using: `EDMA3_DRV_setPaRAM`

   If you're only programming the DMA during system init, this shouldn't be a big deal, but if you are constantly reprogramming the DMA, take note of this tip. BTW, if you're programming the same transfer over-and-over again, use the preferred functions to setup up the first config, then copy the PSET values into a variable using the `EDMA3_DRV_getPaRAM` function.

Most of these tips are self-explanatory. Q0 is actually the highest priority queue by default. So it is best to put small, very important transfers into this queue. The other queues can be used for lower priority larger transfers. But be careful – you don't want to hog a resource and keep the CPU from getting access to that same resource (like L2 or DDR2 memory spaces). It is usually best to break large transfers into smaller chunks.

One neat trick we've learned is self-chaining. Let's say you have 16K of memory to move. Instead of moving the WHOLE darn thing with an AB transfer (ACNT = 1K, BCNT = 16, with AB-sync), why not use A-sync and self chain (chain to yourself) to move 16 1K blocks – giving the SCR some breathing room.

References



EDMA3 References

◆ DM6437 EDMA3 User Guide	(SPRU987)
◆ DM6437 Datasheet	(SPRS345)
◆ EDMA3 Controller	(SPRU234)
◆ EDMA3 Migration Guide	(SPRAAB9)
◆ EDMA Performance	(SPRAAG8)
◆ TC Optimization Rules	(SPRUE23)

For more information on these topics, refer to these User Guides.