

# **TMS320C621x/TMS320C671x EDMA Architecture**

---

*Jamon Bowen  
Jeffery Ward*

*TMS320C6000 Architecture*

## **ABSTRACT**

The enhanced DMA (EDMA) controller of the TMS320C621x™/TMS320C671x™ device is a highly efficient data transfer engine. To maximize bandwidth, minimize transfer interference, and fully utilize the resources of the EDMA, it is crucial to understand the architecture of the engine. Transfer requests (TRs) originate from many requestors, including sixteen programmable EDMA channels, the level 2 (L2) memory controller, and master peripherals. The EDMA controls access to resources and arbitrates between concurrent transfers. Understanding the interaction points for transfer requests in the EDMA architecture is crucial to creating a system that takes full advantage of the EDMA's capabilities.

---

## **Contents**

<b>1</b>	<b>Introduction</b> .....	<b>2</b>
<b>2</b>	<b>EDMA Architecture</b> .....	<b>2</b>
	2.1 Data Transfer Overview .....	3
	2.2 Transfer Requestors .....	3
	2.2.1 Level-two Memory Controller .....	4
	2.2.2 EDMA Channel Controller .....	5
	2.2.3 Master Peripherals .....	6
	2.2.4 Transfer Request Bus .....	6
	2.2.5 Transfer Controller .....	6
	2.2.6 Peripheral Ports .....	7
	2.2.7 Transfer Controller Commands .....	8
<b>3</b>	<b>Transfer Request Submission</b> .....	<b>10</b>
	3.1 L2 Transfer Requests .....	10
	3.1.1 CPU and Cache Transfer Requests .....	10
	3.1.2 QDMA Transfer Requests .....	12
	3.2 EDMA Channel Transfer Requests .....	12
	3.3 HPI Transfer Requests .....	12
<b>4</b>	<b>Priority Queue Allocation</b> .....	<b>13</b>
	4.1 Transfer Requestor Stalls .....	13
<b>5</b>	<b>Transfer Interaction and Arbitration</b> .....	<b>14</b>
<b>6</b>	<b>Priority Inversion</b> .....	<b>16</b>
	6.1 Priority Inversion Due to Port Blocking .....	16
	6.2 Priority Inversion Due to Multiple High-Priority Transfers .....	18
	6.3 Priority Inversion Due to TR Stalls (TR Blocking) .....	18
	6.4 Priority Inversion Due to Read/Write Parallelism .....	19

Trademarks are the property of their respective owners.

<b>7</b>	<b>Resolving Priority Inversion, TR Blocking/Stalls, and Port Blocking</b>	<b>20</b>
<b>8</b>	<b>Conclusion</b>	<b>20</b>
<b>9</b>	<b>References</b>	<b>21</b>

### List of Figures

Figure 1.	C621x/C671x EDMA Architecture Overview	3
Figure 2.	L2 Controller Functionality	4
Figure 3.	Peripheral Port Diagram	7
Figure 4.	Command and Data Busses	9
Figure 5.	L2 Services Multiple Transfers	14
Figure 6.	Port Activity is Determined by Command Buffers	15
Figure 7.	Port Blocking Scenario	17
Figure 8.	Multiple High-Priority Scenario	18
Figure 9.	TR Stall Scenario	19
Figure 10.	Parallel Read/Write Scenario	20

### List of Tables

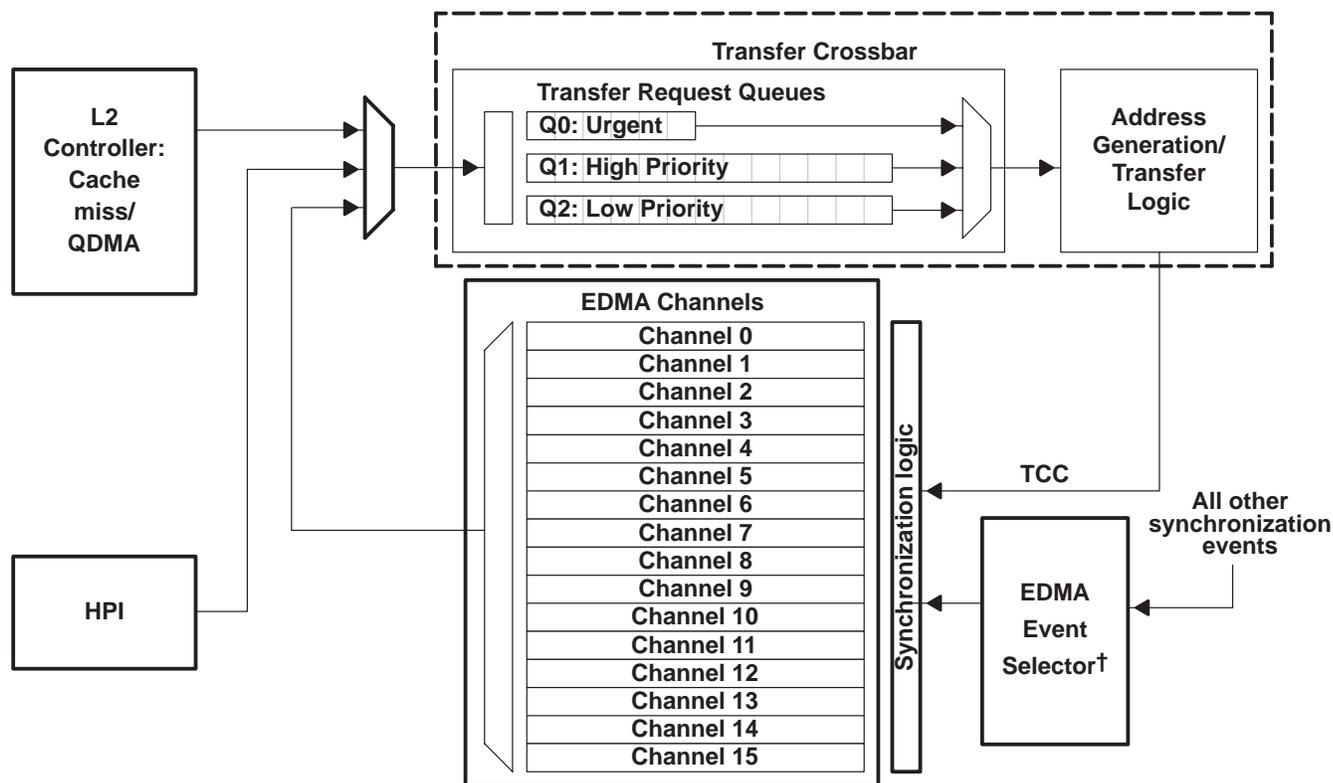
Table 1.	Command Buffers and Burst Sizes	8
Table 2.	EDMA/Cache Activity due to CPU Accesses	11
Table 3.	Data Transferred by an EDMA Channel Transfer Request	12
Table 4.	Data Transferred by an HPI Transfer Request	12
Table 5.	Priority Queue Lengths	13

## 1 Introduction

The enhanced DMA (EDMA) controller of the TMS320C621x/TMS320C671x DSP is a highly efficient data transfer engine, capable of handling up to 8 bytes per EDMA cycle, resulting in 1800 megabytes per second peak throughput at a CPU rate of 225 MHz. The EDMA performs all data movement between the on-chip level-two (L2) memory, external memory that is connected to the device through an external memory interface (EMIF), and the device peripherals. These data transfers include CPU-initiated and event-triggered transfers, master peripheral accesses, cache servicing, and non-cacheable memory accesses. The EDMA architecture has many features designed to service multiple high-speed data transfers simultaneously. With a working knowledge of this architecture and the ways in which data transfers interact, it is possible to create an efficient system and maximize the bandwidth utilization of the EDMA.

## 2 EDMA Architecture

The most important thing to understand, prior to setting up the data movement in a system, is the architecture of the transfer engine. By understanding this architecture, it is possible to then understand the stages through which a transfer is accomplished (see Figure 1). The architecture is the key to knowing how multiple transfers (from multiple transfer requestors) interact with one another, and ultimately impact the system performance.



† C6713, C6712C, C6711C only

Figure 1. C621x/C671x EDMA Architecture Overview

## 2.1 Data Transfer Overview

Every data transfer is initiated by a transfer request (TR), which contains all the information required to perform the transfer including: source address, destination address, transfer priority, element count, etc. TRs are sorted into queues (simple FIFO style buffers) based on priority. Once at the head of the queue, the TR is moved into the EDMA transfer controller's queue registers, which perform the actual data movement defined by the TR.

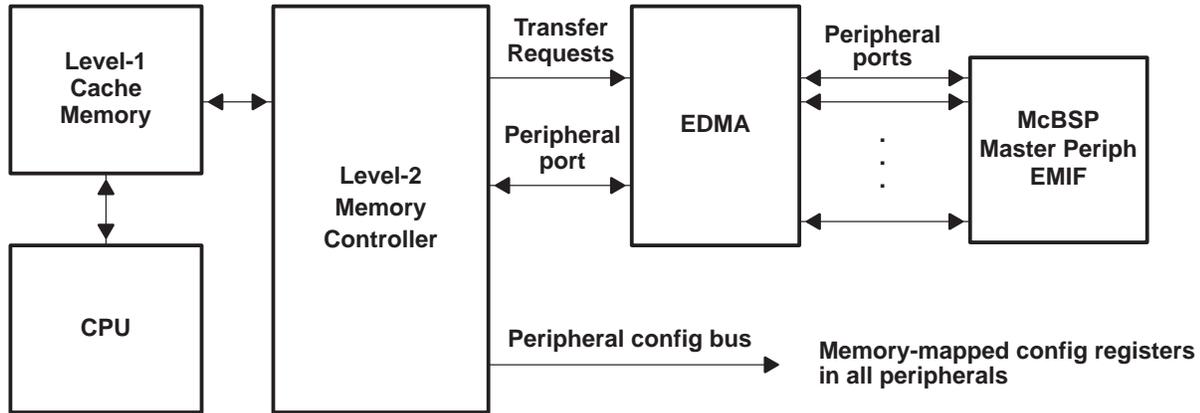
The entire process of TR submission, priority queuing, and arbitration, occurs at the speed of the EDMA, which is equal to CPU frequency. Data movement at the peripheral occurs at the speed of the peripheral. The peripheral ports buffer data to isolate the high-speed EDMA from the peripherals. This is a very efficient architecture, allowing the EDMA to service multiple simultaneous data transfers.

## 2.2 Transfer Requestors

Up to three requestors of data transfers reside inside the DSP: the L2 cache/memory controller, the EDMA channels, and the master peripheral. The transfers requested are different due to the different tasks that each performs. However, the way each transfer request is handled by the EDMA transfer controller is the same, regardless of its requestor.

### 2.2.1 Level-two Memory Controller

The L2 cache/memory controller performs many functions. It services CPU data accesses, submits quick DMA (QDMA) transfer requests, and maintains coherency of the level-1 cache, and of the level-2 cache (if enabled). All communication between the CPU block and the rest of the device must pass through the L2 controller as depicted in Figure 2.



**Figure 2. L2 Controller Functionality**

The L2 controller directs QDMA requests and external memory accesses to the EDMA, L2 cache/memory accesses to the L2 memory, and memory-mapped control register accesses to the peripheral configuration (config) bus. In addition, if any L2 memory is setup as cache, it maintains the coherency of the data between the cache and the cacheable memory space(s). The L2 controller receives L2 memory accesses from the CPU side and from the EDMA side. Constant CPU access to L2 SRAM (long string of store words indexed 8 words apart) can block EDMA L2 transfers and should be avoided (see *TMS320C621x/TMS320C671x EDMA Performance* (SPRAA00) for more details). Some C671x devices have a priority bit in the CCFG that can prevent this condition; refer to the device data sheet for details.

Some accesses to the L2 controller result in an EDMA transfer request (TR); others do not. The L2 controller generates a TR for the following conditions:

- The CPU issues a QDMA transfer.
- The CPU accesses a non-cacheable external memory space.
- The L2 controller performs a cache allocation from external memory – the result of a CPU access to a cacheable memory space.
- The L2 controller performs a cache writeback to external memory
- User initiated cache operations (flush, clean, etc.)

The L2 controller submits all cache servicing transfer requests to the urgent priority queue, which is reserved for cache requests. QDMA transfers (requested by the L2 memory controller) can be set to either low or high priority on a per-transfer basis via the priority bits of the QDMA options register.

Note that some accesses and data paths do not pass through the EDMA. The L2 controller does not generate an EDMA transfer request for the following conditions:

- The CPU accesses memory in the L2 SRAM space. This access goes directly to L2 within the cache memory system; refer to the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (SPRU609).

- The CPU accesses a memory-mapped config register. This access passes through the config bus.
- The CPU accesses a cacheable external memory element that is allocated in L2 or L1 cache. This access goes directly to L2 within the cache memory system; refer to the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (SPRU609).

Details on programming the QDMA can be found in the *TMS320C6000 EDMA Controller Reference Guide* (SPRU234). Information about configuring the L2 cache, defining cacheable and non-cacheable external memory spaces, and programming the cache configuration registers can be found in the *TMS320C621x/TMS320C671x DSP Two Level Internal Memory Reference Guide* (SPRU609).

## 2.2.2 EDMA Channel Controller

There are sixteen EDMA channels that can be configured in a special on-chip parameter RAM (PaRAM), with each channel corresponding to either a specific synchronization event, or a synchronization event programmed in the EDMA event selector (EDMA event selector availability varies by device). The RAM-based structure of the EDMA allows for a great deal of flexibility. Each channel has a complete parameter set accessible via the peripheral config bus, which makes each channels transfer parameters independent of one another. To allow for some interaction between transfers a linking mechanism is available to EDMA channels. Once fully exhausted, new channel parameters may be automatically loaded with a new set that is stored in the PaRAM via the linking mechanism.

One EDMA TR is issued per synchronization event received. The transfers requested by the EDMA channels are completely dependent on the configuration programmed by the user. Details on programming EDMA channels are not included in this document. For transfer examples see the examples section of the *TMS320C6000 EDMA Controller Reference Guide* (SPRU234).

### 2.2.2.1 EDMA Event Selector (C6713/C6712C/C6711C Only)

Some of the TMS320C671x DSPs have more events defined for the EDMA than EDMA channels available. To accommodate this, an EDMA event selector (similar to an interrupt selector) is available to select the event to associate with each EDMA channel. This event selector allows each channel (excluding the channels reserved for chaining) to be synchronized with any EDMA event via the event selection registers (ESEL0–ESEL3). For more information on the EDMA event selection register consult the *TMS320C6000 EDMA Controller Reference Guide* (SPRU234).

## 2.2.3 Master Peripherals

As of the writing of this document, the only master peripheral available to TMS320C621x/TMS320C671x DSPs is the Host Port Interface (HPI). Future DSPs in this line may contain additional master peripherals. Master peripheral servicing is performed without any user intervention, via a direct connection to the EDMA. This direct connection allows the master peripheral to submit transfer requests to the EDMA transfer controller, just as the L2 controller and the EDMA channels do.

The requests made to the EDMA are dependent on the master activity. Master activity consists of transfers between the master peripheral to or from any location in the DSP's memory map. The priority level of HPI transfers is fixed to high priority on current C621x/C671x devices, refer to the device data sheet for details.

For information on programming the master peripherals, see the appropriate chapter referenced in the *TMS320C6000 DSP Peripherals Overview Reference Guide* (SPRU190).

### **2.2.4 Transfer Request Bus**

The transfer requestors to the EDMA are connected to the transfer controller (TC) via the transfer request (TR) bus. If multiple TRs arrive at the TR bus simultaneously, they are submitted in the order of their priority. This has little impact on performance because these requests are arbitrated quickly (in about 2–4 EDMA cycles) compared to data transfer rates.

### **2.2.5 Transfer Controller**

Transfer requests are queued in the transfer controller based on their priority. The transfer controller is the portion of the EDMA that processes the TR and performs the actual data movement (see Figure 1).

Within the TC, the TR is shifted into one of the transfer request queues to await processing. The transfer priority level determines the queue to which it is submitted. There are three queues, corresponding to three priority levels: Q0 (Urgent, reserved for cache requests, 6 entries deep), Q1 (High, 13 entries deep), and Q2 (Low, 11 entries deep). TMS320C621x/ TMS320C671x EDMA and QDMA transfer requestors are programmable such that TRs can be submitted to either the high or the low priority queue; cache request TRs from the L2 are always submitted to the urgent queue.

Once the transfer request reaches the head of its queue, it is submitted to the queue registers to be processed. Only one TR from each priority queue can be serviced at a time by the address generation/transfer logic. However, the transfer logic can process transfers of different priorities concurrently, provided that they are to/from different source/destinations. To maximize the data transfer bandwidth in a system this architecture should be taken advantage of; transfers should be distributed among all priorities whenever possible. (This topic is discussed at length in the *TMS320C6000 EDMA IO Scheduling and Performance* (SPRAA00).

The TC contains three queue register sets, one for each priority queue, that monitor the progress of a transfer. Within the register set for a particular queue, the current source address, destination address, and count are maintained for a transfer. These registers are not present in the device's memory map and are unavailable to the CPU. The queue registers monitor the actual data transfer, which occurs between peripheral ports that are connected to the TC. The architecture of these peripheral ports is discussed in the next section.

### **2.2.6 Peripheral Ports**

Peripherals involved in high speed data traffic (McBSP/McASP, master peripheral, EMIF, and L2 controller) have ports that accept commands from the TC as shown in Figure 3. Each includes read and write FIFO command and data buffers between the high speed EDMA engine and the peripheral, which may operate at some lower frequency. The ports receive TC commands and access the peripherals directly, freeing the EDMA to service other transfers while waiting for a response from the peripheral. This design allows transfers to/from different peripherals on different priority levels to occur simultaneously.

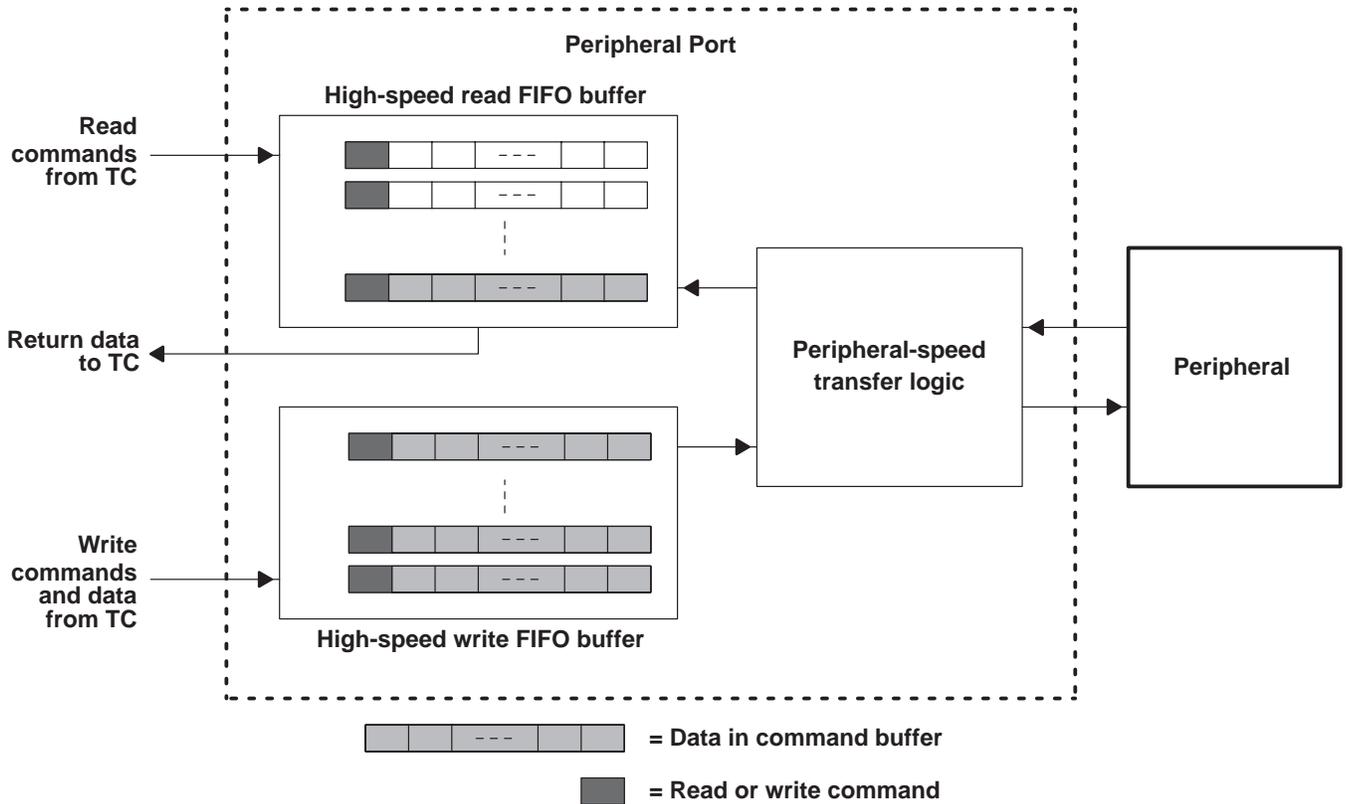


Figure 3. Peripheral Port Diagram

The number of command buffers in each peripheral port (as well as the default burst size of that port) is fixed in order to maximize efficiency. The buffer size doesn't limit the bursts between the TC and the port as long as some of the buffers are emptied as the burst continues (likely for L2).

Table 1. Command Buffers and Burst Sizes

Peripheral†	Reads		Writes	
	Command Buffers	Buffer Length (words)	Command Buffers	Buffer Length (words)
L2 Memory Controller	8	2	8	2
McBSP 0/1	2	1	2	1
McASP 0/1	2	4	2	4
HPI	4	8	2	8
EMIF	2	4	2	16

† Peripheral availability varies by specific device. Refer to device data sheet.

For the McBSP and McASP ports the EDMA speed relative to the external transfer rate generally allow the external transfer ample time to be serviced. The buffers for these ports only exist to optimize the internal TC traffic.

Peripheral and EMIF ports service all commands in the order of their arrival. For example, suppose four read commands arrive followed by four write commands. The four reads are serviced followed by the four writes.

In contrast, the L2 port services reads and writes alternately, in the order of their arrival. Again, suppose four read commands arrive followed by four write commands. The first read is serviced, then the first write, followed by the second read, then the second write, and so on.

### 2.2.7 **Transfer Controller Commands**

To perform a transfer, the TC sends commands to source and destination ports for data to be read/written. These commands are for small bursts of data, which are less than or equal to the total transfer size of the submitted transfer request. The default burst size and the number of command buffers per port is shown above in Table 1. The TC sends commands to the ports for data transfers, but the actual data movement doesn't occur until the port is ready. However, waiting for the port to become ready does not stall the TC. Therefore, if the different queues request transfers to/from different ports, the transfers can occur simultaneously. Transfer commands made to the same port(s) are arbitrated by the TC according to priority.

To initiate a data transfer, the TC submits a command to the source or destination pipeline. There are three commands: pre-write, read, and write. Commands can be submitted to both pipelines once per cycle by any of the queue register sets. The TC arbitrates every cycle (separately for each pipeline) to allow the highest priority command that is pending to be submitted to the appropriate port. The pre-write command is issued to notify the destination that it is going to receive data. Once the destination has available space to accommodate the incoming data, it sends an acknowledgement to the EDMA that it is ready.

After receiving the acknowledgment from the destination, a read command is issued to the source port. Data is read at the maximum frequency of the source into the command buffer, then passed to the EDMA routing unit to be sent to the destination. Once the routing unit receives the data, the data is sent along with a write command to its destination port.

Due to the EDMA's capability to wait for the destination's readiness to receive data, the source resource is free to be accessed for other transfers until the destination is ready. This provides an excellent utilization of resources and is referred to as write-driven processing. All write commands and data are sent from the EDMA to all resources on a single bus. The information is passed at the clock speed of the EDMA, and data from multiple transfers is interleaved based on priority when occurring simultaneously.

In this way, the EDMA transfer controller services commands and data from multiple transfers simultaneously. Also, ports can service more than one active transfer if they have the bandwidth to do so, always giving precedence to the highest priority transfer. This is especially useful for the L2 memory port which is the fastest port in the system, and can often service multiple active TRs.

The read data arrives on unique buses from each resource. This is to prevent contention and to ensure that data can be read at the maximum rate possible. Once the data arrives at the routing unit, the data that is available for the highest priority transfer is moved from its read bus to the write bus and sent to the destination port.

The queue register sets, command bus, and routing unit are depicted below in Figure 4.

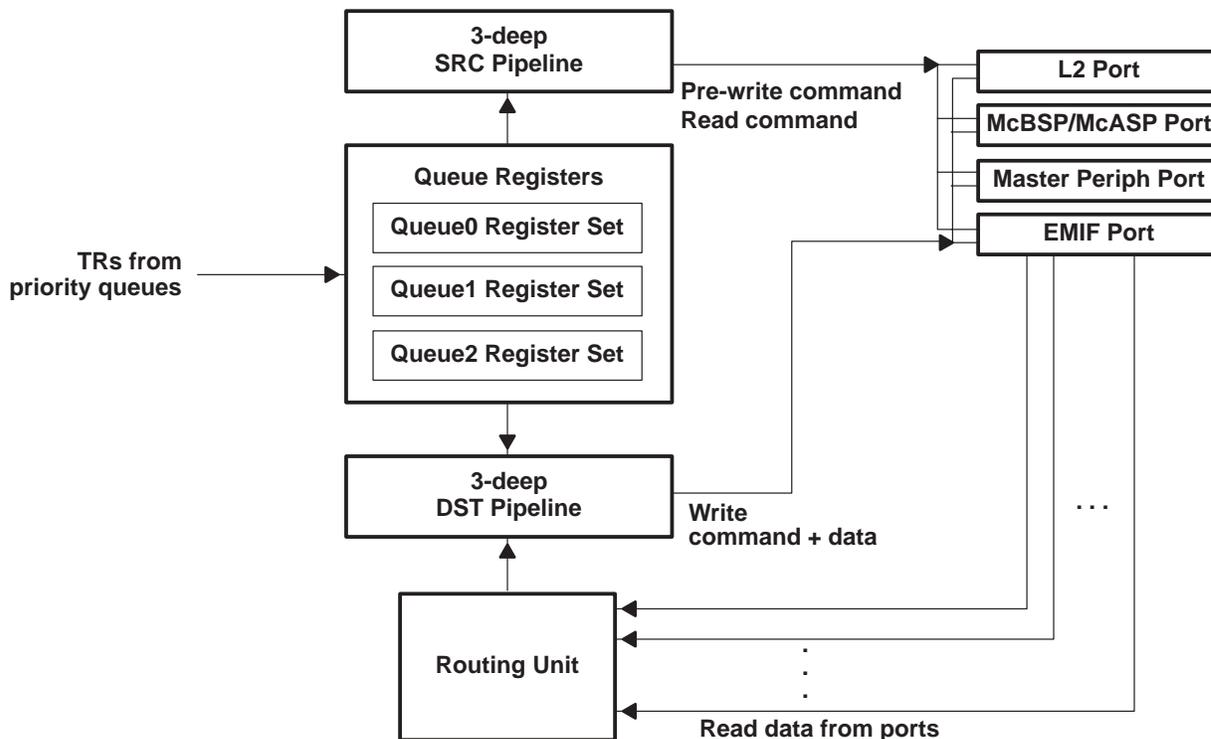


Figure 4. Command and Data Busses

### 3 Transfer Request Submission

Knowing how and when TRs are submitted is important to understand when scheduling data traffic in a system. The types of TRs submitted to the hardware differ slightly depending on the requestor, but all TRs contain the same essential information: source and destination addresses, element count, and the relationship between the elements within the source and destination regions (fixed, increment, decrement, or indexed).

#### 3.1 L2 Transfer Requests

The L2 transfer controller handles TR submission for CPU data accesses, L1/L2 cache allocations from EMIF, L1/L2 cache evictions to EMIF, and QDMA transfers.

##### 3.1.1 CPU and Cache Transfer Requests

The L2 controller services CPU requests and maintains L2 cache coherency. The L2 cache is of programmable size, and it resides between the CPU's level-1 cache (L1) and the rest of the DSP's memory mapped space.

All cacheable memory spaces will be serviced by the L1 cache and/or the L2 cache. If L2 cache size is zero (L2 cache is disabled), then cacheable memory will be serviced by the L1 cache only. If L2 cache size is not zero (L2 cache is enabled), then cacheable memory will be serviced by both L1 and L2 cache.

When determining system traffic, it is important to know when the L2 controller will generate transfer requests. There are five basic CPU/L2 actions which trigger TR submission, listed above in section 2.2.1. However, one or more of these actions can be triggered based on CPU activity. To determine exactly what circumstances generate which TRs, refer to Table 2.

**Table 2. EDMA/Cache Activity due to CPU Accesses**

Read/Write Destination	L1 Controller Action	L2 Cache Enabled/ Disabled	L2 Controller Action	Number of TRs Submitted	Number of Elements per TR
Internal registers	None	Don't care	None	0	–
Memory mapped control registers	Forward request to L2 controller	Don't care	Read from config bus – no EDMA action	0	–
L2 SRAM	Hit returns data; read miss allocates one L1 cache line from L2 controller. Write miss is passed on to L2.	Don't care	Read SRAM – no EDMA action	0	–
Non-cacheable EMIF	Forward request to L2 controller	Don't care	Submit TR to EDMA	1	1
Write Access to Cacheable EMIF	Hit returns data; miss is passed on to L2.	Disabled	Submit TR to EDMA	1	1
Read Access to Cacheable EMIF	Hit returns data; read miss allocates one L1 cache line from L2 controller.	Disabled	Requests 1 L1 cache line from EDMA	1	L1 line size (32 bytes)
Cacheable EMIF	Hit returns data; read miss allocates one L1 cache line from L2 controller. Write miss is passed on to L2	Enabled	Hit returns L1 cache line; miss allocates one L2 cache line from EDMA	2	128 Bytes transferred by the 2 TRs

Note the two-level memory structure. A data request traverses the memory hierarchy until the data is found. The hierarchical data access sequence is the following:

1. The CPU requests data from the L1 controller.
2. The L1 controller checks L1 memory, and requests data from the L2 controller if the data is not in L1.
3. The L2 controller checks L2 memory, and if the data is not in L2, requests data from the peripheral config bus or the EDMA (depending on the address range of the access).
4. Data requests to the EDMA result in TRs.

Cache hits can reduce CPU wait states, and they have the added benefit of reducing EDMA traffic. For example, by using cache to access data in EMIF, the first request allocates one cache line from EMIF by submitting a TR. Subsequent hits to that cache line are returned quickly and no TR is issued.

Also note that cache writebacks to EMIF will generate a single TR to write out modified data. These occur anytime there is no space in cache memory for a pending allocation, and the least recently used cache line contains dirty data.

For additional details on the two-level cache architecture of the C621x/C671x devices, including how to define memory spaces as cacheable, see the *TMS320C621x/TMS320C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609).

### 3.1.2 QDMA Transfer Requests

The L2 controller also submits TRs for QDMA transfers. QDMA transfers are initiated by software. QDMA transfers are submitted by writing to the QDMA pseudo-registers. QDMA transfers are for simple block or frame transfers and can take as little as 1 cycle to submit. The resulting TR from a QDMA transfer is submitted to the transfer controller from the L2 on either the high or low priority queue. Thus, the L2 controller can submit TRs to any queue: Q0 – cache service requests, Q1 – QDMA, and Q2 – QDMA. For more information on submitting QDMA transfer requests see the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

### 3.2 EDMA Channel Transfer Requests

EDMA channels can be programmed to transfer data in a large variety of ways. Each channel is synchronized to a particular system event. One event corresponds to one TR submission, which transfers all or some of the data described by the parameter set. Due to the large number of configurations possible, the programming of an EDMA channel is not described in this document. For details see the *TMS320C6000 EDMA Controller Reference Guide* (SPRU234). The amount of data transferred by a single TR is shown in Table 3.

**Table 3. Data Transferred by an EDMA Channel Transfer Request**

Source Dimension	Destination Dimension	Synchronization	Data Transferred by TR
1-D	1-D	Read/Write (FS=0)	1 element
1-D	1-D	Frame (FS=1)	Element count (one frame)
	Other	Array (FS=0)	Element count (one array)
	Other	Block (FS=1)	(Array count + 1) x element count

### 3.3 HPI Transfer Requests

The HPI controller submits TRs onto the high priority queue based on programmable actions performed by the host. To maximize the bandwidth available to host data transfers there are read and write FIFOs implemented in the HPI FIFO, each of which can contain eight 32-bit words. When possible, the HPI bursts multiple words between the HPI FIFOs and the physical memory. Table 4 describes the burst size of the TR submitted, depending on the host activity involved.

**Table 4. Data Transferred by an HPI Transfer Request**

Host Access	Situation	Data Transferred by TR
Non-auto-increment read	HPI reads HPID register	1 word
Non-auto-increment write	HPI writes HPID register	1 word
Auto-increment read	HPI reads HPID register and FIFO is empty	8 words
Auto-increment read	HPI reads HPID register, FIFO is less than or equal to half full	4 words
Auto-increment write	HPI writes to HPID and data is the fourth element written since last TR issued	4 words

Data is transferred based on the host activity. If the host is performing individual accesses (accessing HPID in non-auto-increment mode), TRs are submitted for each individual element. If the host is performing burst transfers (accessing HPID in auto-increment mode) then TRs are submitted for multiple contiguous elements at a time. See the *TMS320C6000 DSP Host Port Interface Reference Guide* (SPRU578) for additional information on the HPI, including a block diagram, register descriptions, pin listing, and waveforms.

## 4 Priority Queue Allocation

To prevent any one transfer requestor from inundating the priority queues it is necessary to limit the number of outstanding TRs each requestor can submit. For this purpose, each requestor has a limit on the number of TRs it can submit on a priority level. The lengths allocated to the various requestors are listed in Table 5.

**NOTE:** An outstanding TR is one that has been submitted to the priority queues and is awaiting processing, once the TR reaches the priority queue registers it is no longer considered outstanding.

**Table 5. Priority Queue Lengths**

Queue	Priority	Total Queue Length	Requestor	Requestor Queue Limit
Q0	Urgent	6	L2 Controller	6
Q1	High	13	QDMA	3
			EDMA Channels	8
			Master peripherals (HPI)	2
Q2	Low	11	QDMA	3
			EDMA Channels	8

### 4.1 Transfer Requestor Stalls

If a requestor has submitted its maximum allotment of TRs for a given priority queue, its next TR of the same priority stalls the requestor. The stall is resolved when a TR from the stalled requestor on the priority level that caused the stall reaches the queue registers.

If the L2 controller requestor experiences a TR stall, subsequent requests by the CPU will be stalled as well. For example, if the CPU submits four consecutive QDMAs to the high priority queue, the L2 requestor is stalled until the first QDMA begins processing. While stalled, the L2 requestor cannot submit TRs on any priority for any action, including QDMA, CPU EMIF accesses, and cache accesses.

If the EDMA channel controller requestor is stalled, subsequent events will not generate TRs, regardless of their priority. For example, if nine low priority EDMA transfers are triggered consecutively, the EDMA channel requestor will stall. Subsequent events for any priority level cannot be submitted until the first low-priority transfer begins servicing.

Note that if the EDMA channel controller requestor is stalled, the EDMA channel controller continues to receive synchronization events, and those events will generate TRs once the controller is freed. Events are not lost during a stall unless the same particular event is received multiple times during the stall.

A TR stall scenario is depicted in the priority inversion section below. TR stalls severely inhibit efficient operation and must be avoided in a system.

## 5 Transfer Interaction and Arbitration

Knowing how multiple transfers interact once they are submitted is important for maximizing the performance obtained.

There are three places in the EDMA where transfers must be arbitrated: at the transfer requestor nodes, in the priority queues, and during active processing in the queue registers.

Arbitration at the requestor nodes is a simple matter. If a requestor is stalled, no TRs may be submitted. If requestors are not stalled, TRs are submitted in the order of their arrival. If TRs arrive simultaneously, they are submitted in a round-robin style. This has little significance as the delay is only 2–4 EDMA cycles per request.

In the priority queues, arbitration is also a fairly simple matter. TRs submitted to a single priority queue are processed serially in the order of their arrival. The priority queue is a simple FIFO in this respect. For this reason, long data transfers should not be placed on the same priority as short, time-critical transfers, because the short transfer could be queued behind the long transfer. If a large transfer is considered to be high priority, it is best to break the transfer up into multiple shorter bursts by using the linking or chaining capabilities of the EDMA (see the *TMS320C6000 EDMA Controller Reference Guide*, SPRU234). By submitting multiple small TRs for one large transfer, the time-critical TRs (McBSP/HPI/etc.) can get in between the small TRs and not be queued for the full duration of the long transfer.

The final place TRs are arbitrated is in the queue registers. Each priority level has a queue register set which, when ready to service a transfer, fetches a TR from the head of its priority queue and begins processing.

The design of the queue registers and priority scheme make the EDMA a very efficient transfer engine, capable of servicing up to three simultaneous transfers – one in each queue register set. Determining arbitration among the queue registers is generally a simple task. Commands are issued from the TRs to the peripheral ports. If two TRs attempt to utilize the same port, they are arbitrated by priority.

To make the EDMA more efficient, ports are only “busy” if they are actively transferring data. This is true for any port, but is mainly an advantage for the L2 port. Take the following example as a first approximation of how this occurs:

Assume that a TR is transferring data from the L2 memory to external memory on the EMIF. Since the L2 memory operates at a higher frequency than the EMIF, the L2 port will be “busy” only for short intervals during the transfer. This allows a TR that is servicing the McBSP to be serviced simultaneously, even though it utilizes the L2. Figure 5 illustrates this concept.

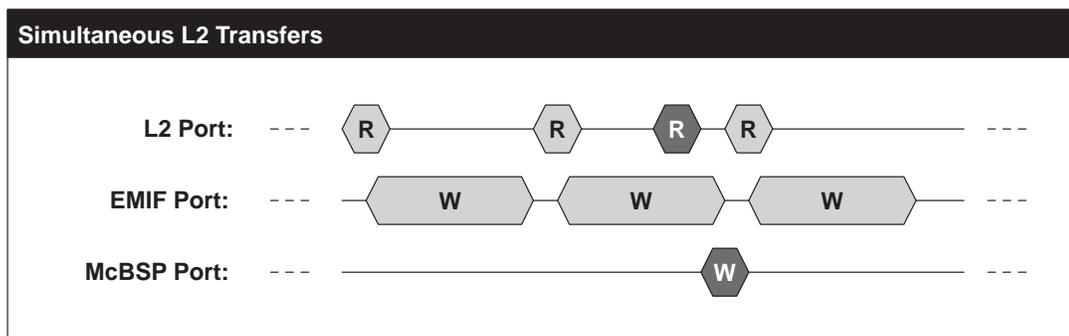
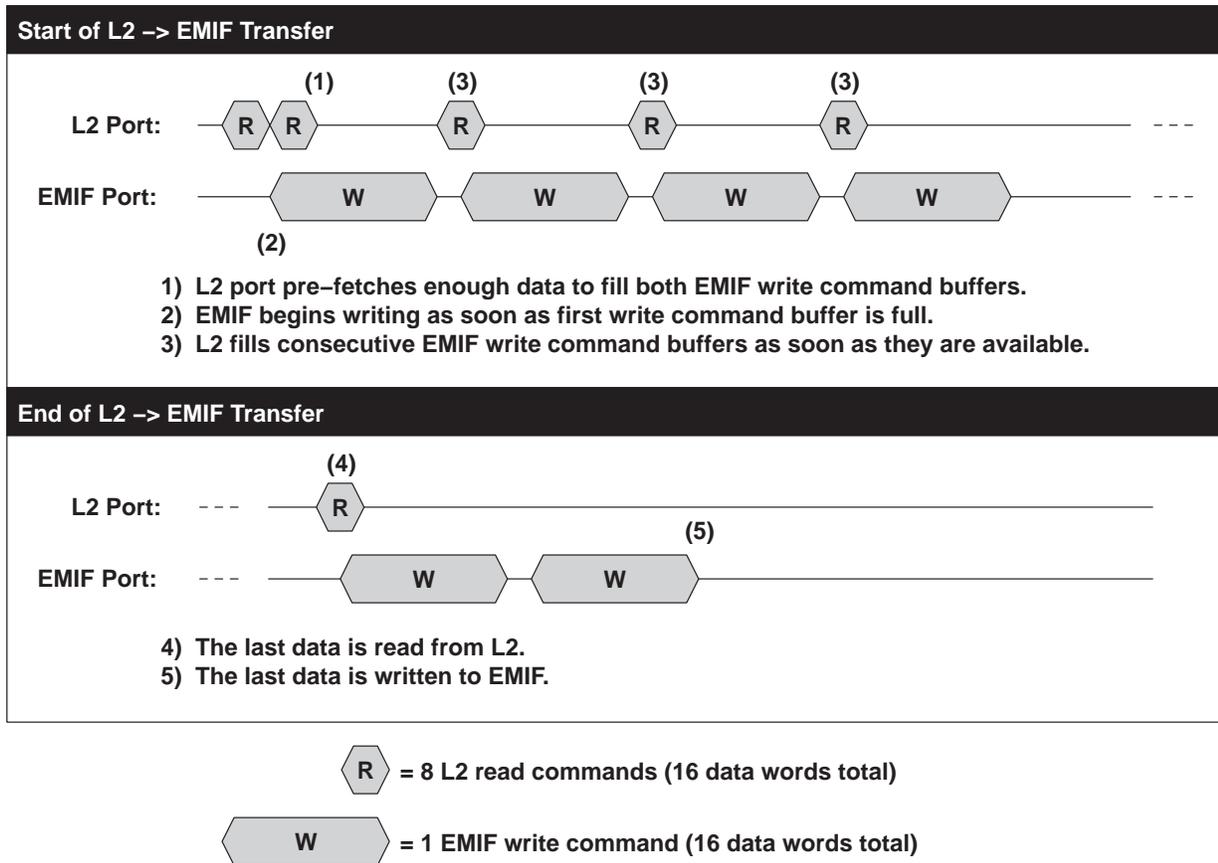


Figure 5. L2 Services Multiple Transfers

To expand on this example, remember that data is stored in the peripheral port's command buffers in between these bursts. There are multiple command buffers in each port, and buffer size depends on the peripheral. The EDMA will burst data as long as there is space in the command buffers. The sizes of these command buffers are shown in table 1.

Consider another transfer from the L2 port to EMIF. The L2 port has 8 read command buffers of 2 words each. The EMIF has 2 write command buffers of 16 words each. The activity at the ports will be as shown in Figure 6.



**Figure 6. Port Activity is Determined by Command Buffers**

The L2 port will be issued 8 read commands (of 2 words each). At this point, one full EMIF write command buffer will be filled (16 words), and the EMIF will begin writing this data. At the same time, the L2 will still be reading data to fill up the remaining EMIF write command buffer. This is the initial pre-fetch, during which time the L2 bandwidth is utilized fully.

After this pre-fetch, the L2 is utilized to keep the EMIF write buffers full. The L2 is only “busy” for a fraction of time for the remainder of the transfer. This fraction is roughly equal to the ratio of EMIF bandwidth to L2 bandwidth. Ideally, if the EMIF had a bandwidth of 400 MB/sec, and the L2 had a bandwidth of 1800 MB/sec, this transfer would only require about 22% of the L2’s bandwidth. This number is a very rough estimate, not accounting for data propagation delays and other factors, but it is useful for approximation.

At any time during this transfer, the L2 port may be diverted to service a higher priority transfer. However, because there is pre-fetched data stored in the EMIF command buffers, the L2 port may service the higher priority transfer and return to the EMIF transfer without interrupting EMIF data flow. By buffering data in the peripheral ports and utilizing write driven processing, the EDMA makes excellent utilization of resources and services multiple transfers efficiently.

## 6 Priority Inversion

Under certain circumstances, a low-priority transfer will appear to take precedence (as seen from the device pin perspective) over another transfer which the user intended to be of higher priority. This situation is known as priority inversion.

When defining types of priority inversion, the terms *low-priority transfer* and *high-priority transfer* are used to reference transfers of relative priority, rather than absolutely corresponding to transfers on priorities Q2 and Q1. For example, a transfer on priority Q0 (urgent priority) is a *high-priority transfer* when compared to the relatively *low-priority transfer* on Q1 (high priority).

There are four basic priority inversion scenarios:

- Priority inversion due to port blocking
- Priority inversion due to multiple high-priority transfers
- Priority inversion due to TR stalls
- Priority inversion due to read/write parallelism

### 6.1 Priority Inversion Due to Port Blocking

Priority inversion can occur when a port is in use for a low-priority transfer. When a high-priority transfer request reaches the queue registers, it immediately has priority over the low-priority transfer to submit commands to the ports. However, because the ports are simple FIFO buffers, any remaining commands from the low-priority transfer must complete before the high priority commands are serviced. Since commands are serviced in the order of their arrival, these commands can be from the read or write command buffers, or both.

From the pin perspective, it seems that the low-priority transfer is taking precedence, because the pins will still service the transfer in progress until the buffered read/write commands are flushed.

Below is an example port blocking scenario. At some time before 0, transfer request A is active in the low priority level, writing to EMIF (a 32-bit, 100 MHz SBSRAM). It fills the EMIF peripheral port with write commands and data. The size of the write command buffer is determined from Table 1.

Later, at time 0, transfer request B, a cache service request which writes to EMIF, arrives from priority queue 0. The destination pipe recognizes this and immediately gives commands from TR B priority. However, as illustrated at 160 nS and 320 nS, the EMIF will not begin servicing TR B's write commands until TR A's previously buffered commands are completed (flushed from the buffer).

The delay depends on the operating frequency and bandwidth of the destination and the size and number of commands to be flushed. In this case, EMIF flushes 2 commands of 16 words each, which it takes a total of 320 nS for a 100MHz, 32-bit external memory.

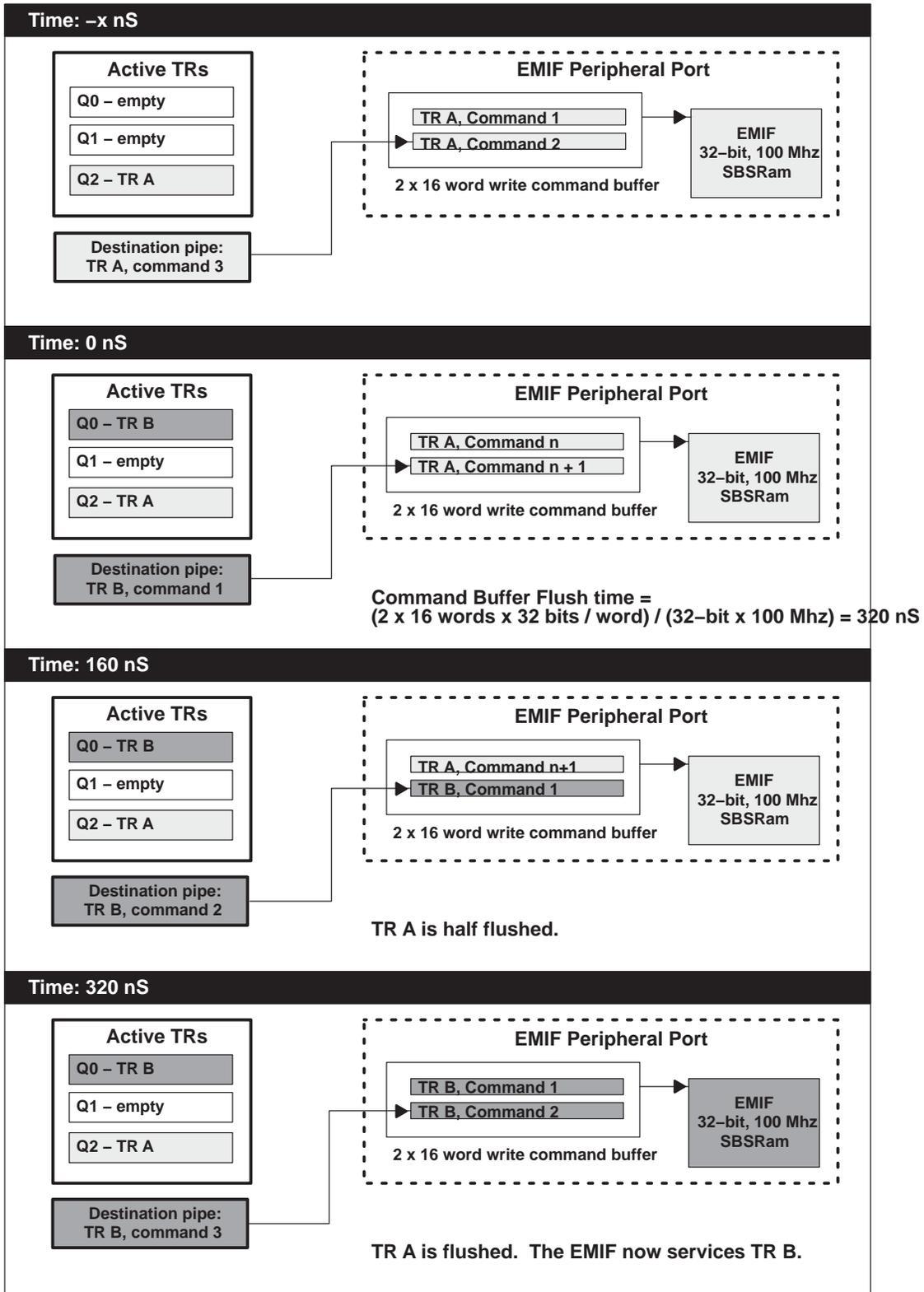


Figure 7. Port Blocking Scenario

This type of priority inversion is mainly a concern for the EMIF, which has the largest command buffers and a longer flush time. The effects of this type of priority inversion can be minimized by proper system traffic scheduling.

## 6.2 Priority Inversion Due to Multiple High-Priority Transfers

Priority inversion can occur if a high-priority transfer is waiting for another transfer on the same priority level to complete. This is because there is only one set of queue registers per priority queue. While the high-priority transfer is stalled in the queue, a low-priority transfer in progress could utilize resources that the pending high-priority transfer is waiting to use.

From the pin perspective, it seems that the low-priority transfer is taking precedence. The status of the high-priority transfer, stuck in the priority queue behind another high-priority transfer in progress, is externally invisible.

Figure 8 is an example of priority inversion due to multiple high-priority transfers. Assume that transfer request A is a relatively large, ongoing transfer in the transfer controller. At some later time, TRs B and C enter the queues, both intending to read from the same source. Even though TR C is of higher priority, TR B will begin servicing first, because TR C must wait for TR A to finish processing in the Q1 queue registers.

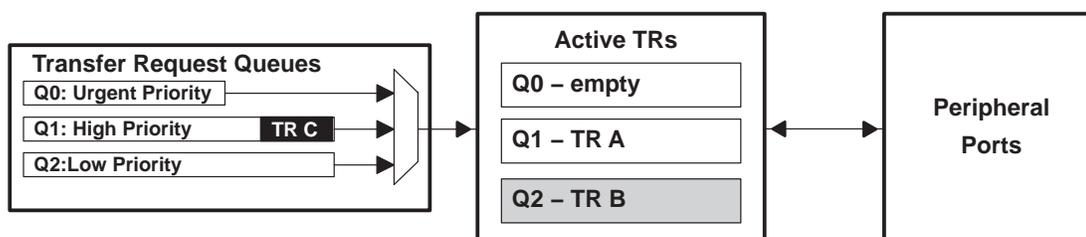


Figure 8. Multiple High-Priority Scenario

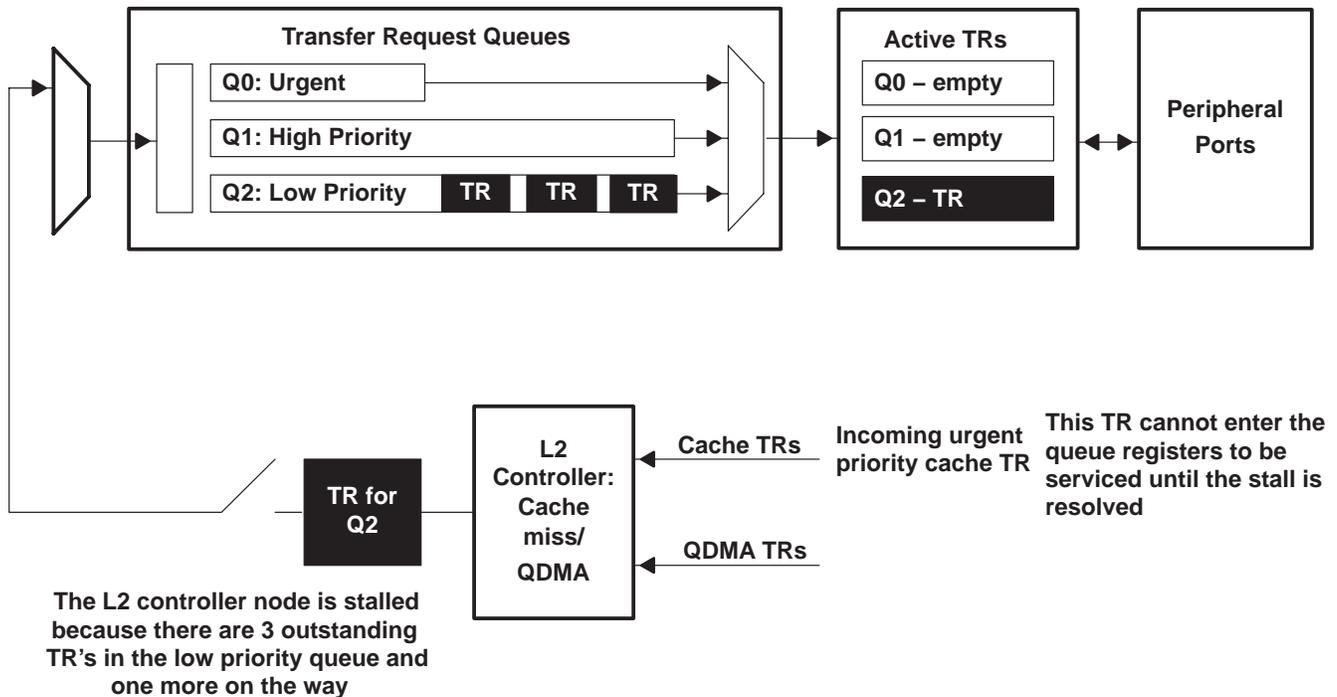
This type of priority inversion is alleviated by properly scheduling transfers on all available priority levels, and by breaking up large, high-priority transfers.

## 6.3 Priority Inversion Due to TR Stalls (TR Blocking)

Priority inversion can occur if the high priority requestor is stalled and therefore cannot submit a high priority TR. This case is especially applicable to the EDMA and L2 requestors. If the EDMA requestor is stalled (by submitting more than its allotment of TRs to a priority level), subsequent events on any priority are not serviced until the stall is resolved. Similarly, if the L2 requestor (Cache servicing and QDMA) is stalled (by submitting more than its allotment of TRs to a priority level) subsequent L2 submissions are delayed.

From the pin perspective, it could seem that a low-priority transfer is taking precedence. The status of the stalled requestor is externally invisible (another reason stalls should be avoided).

An example of priority inversion due to a transfer requestor stall is shown in Figure 9. The QDMA is limited to 4 low-priority transfer requests: 1 in the queue registers, and 3 TRs that can be submitted to the low priority queue. As shown, the QDMA channel controller has attempted to submit a fifth TR to Q2, resulting in a stall. The urgent priority event from a cache service request will not be serviced until the L2 controller stalled is resolved (when a Q2 TR enters the TC for processing). The stall only affects the L2 node, so the EDMA channels or a master peripheral can continue to submit transfer requests.



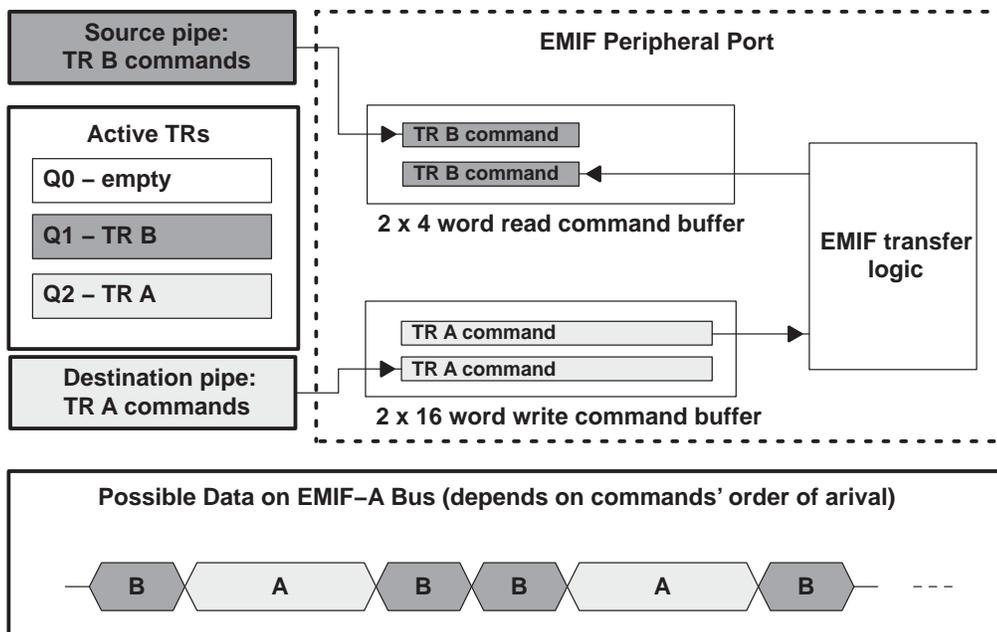
**Figure 9. TR Stall Scenario**

This type of priority inversion should be avoided by properly scheduling system traffic, thereby eliminating TR stalls.

#### 6.4 Priority Inversion Due to Read/Write Parallelism

The read and write command pipelines in the EDMA transfer controller operate in parallel to maximize bandwidth and minimize transfer stalls. Unfortunately, this has a negative effect on the priority scheme in that, if a high-priority transfer reads from a port and a low-priority transfer writes to a port (or vice versa), the port will receive both sets of commands. This is an issue because most ports have no priority decode mechanism so they simply service commands in the order of their arrival.

An example of priority inversion due to read/write parallelism is shown in Figure 10. Assume TR B is reading from EMIF on high priority, and TR A is writing to EMIF on low priority. The source and destination pipelines submit commands separately, and the peripheral port simply services commands in the order of their arrival.



**Figure 10. Parallel Read/Write Scenario**

This type of priority inversion mainly affects memory ports which often service multiple medium to large transfers. However, the L2 port is generally much faster than the other ports involved and there is no noticeably degradation. This priority inversion can be avoided by properly scheduling transfers or adjusting priority levels.

## 7 Resolving Priority Inversion, TR Blocking/Stalls, and Port Blocking

Priority inversion, TR stalls/TR blocking, and port blocking scenarios can be minimized or even avoided completely by properly setting up data flow in a system. This is discussed in depth in *TMS320C6000 EDMA IO Scheduling and Performance* (SPRAA00).

## 8 Conclusion

The C621x/C671x EDMA is a highly efficient data transfer. To be able to schedule system traffic, maximize bandwidth utilization and minimize transfer conflicts, it is important to understand the architecture of this transfer engine. Transfers start when a requestor (L2 controller, EDMA channel, HPI) submits a transfer request for data to be transferred by the EDMA. These transfers can interact with one another at three different times: during submission by the requestor, within the transfer priority queues, and during active transferring. The first of the three has very little impact on performance, but the latter two can have an impact depending on the transfers' priority levels and properties. Transfer requests submitted to the same queue will be serviced in the order they are received. Active transfers (those in the queue registers) submit commands to the peripheral ports in order of priority. The ports have command buffers, and service commands in a FIFO manner. Because ports carry out transfers, the EDMA can service multiple requests simultaneously. By understanding the EDMA architecture, it is possible to maximize the data throughput and minimize the blocking of time-critical data transfers.

## 9 References

1. *TMS320C621x/TMS320C671x EDMA Queue Management Guidelines* (SPRA720)
2. *TMS320C6000 EDMA IO Scheduling and Performance* (SPRAA00)
3. *TMS320C621x/TMS320C671x EDMA Performance Data* (SPRAA03)
4. *TMS320C6000 DSP Peripherals Overview Reference Guide* (SPRU190)
5. *TMS320C6000 EDMA Controller Reference Guide* (SPRU234)
6. *TMS320C621x/ TMS320C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609)

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<b>Products</b>		<b>Applications</b>	
Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>	Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>	Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>	Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>	Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>	Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>	Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>	Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
		Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
		Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
		Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

Mailing Address: Texas Instruments  
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2004, Texas Instruments Incorporated