TMS320TCI6487/8
Digital Signal Processor
Silicon Revisions 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Silicon Errata

Literature Number: SPRZ248H
September 2007–Revised March 2011

List of Figures

List of Tables

TMS320TCI6487/8 DSP Silicon Revisions 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

1 Introduction

This document describes the silicon updates to the functional specifications for the TMS320TCI6487/8 digital signal processor; see the device-specific data manual, *TMS320TCI6487/8 Communications Infrastructure Digital Signal Processor* (literature number SPRS358).

1.1 Device and Development Support Tool Nomenclature

To designate the stages in the product development cycle, TI assigns prefixes to the part numbers of all DSP devices and support tools. Each DSP commercial family member has one of three prefixes: TMX, TMP, or TMS (e.g., TMS320TCI6487/8ZUN). Texas Instruments recommends two of three possible prefix designators for its support tools: TMDX and TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices/tools (TMS/TMDS).

Device development evolutionary flow:

TMX	Experimental device that is not necessarily representative of the final device's electrical specifications
TMP	Final silicon die that conforms to the device's electrical specifications but has not completed quality and reliability verification
TMS	Fully-qualified production device

Support tool development evolutionary flow:

TMDX	Development-support product that has not yet completed Texas Instruments internal qualification testing
TMDS	Fully-qualified development-support product

TMX and TMP devices and TMDX development-support tools are shipped against the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

TMS devices and TMDS development-support tools have been characterized fully, and the quality and reliability of the device have been demonstrated fully. TI's standard warranty applies.

Predictions show that prototype devices (TMX or TMP) have a greater failure rate than the standard production devices. Texas Instruments recommends that these devices not be used in any production system because their expected end-use failure rate still is undefined. Only qualified production devices are to be used.

1.2 Package Symbolization and Revision Identification

The device revision can be determined by the lot trace code marked on the top of the package. The location of the lot trace code for the ZUN package is shown in Figure 1. Figure 1 also shows an example of TCI6487/8 package symbolization.

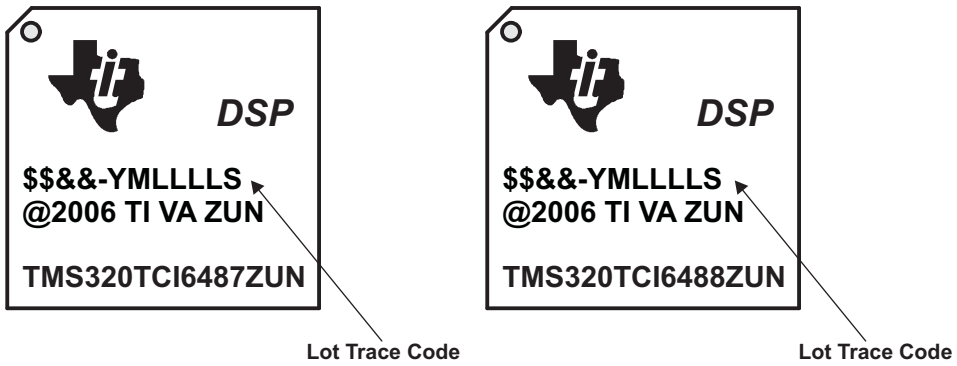


Figure 1. Lot Trace Code Examples for TMS320TCI6487/8 (ZUN Package)

Silicon revision correlates to the lot trace code marked on the package. This code is of the format $$&&-YMLLLLS . If && is "21", then the silicon is revision 2.1. Table 1 lists the silicon revisions associated with each lot trace code for the TCI6487/8 device.

Each silicon revision uses a specific revision of the CPU and the TMS320C64x+™ megamodule. The CPU revision ID identifies the silicon revision of the CPU. Table 2 lists the CPU and C64x+ megamodule revision associated with each silicon revision. The CPU revision can be read from the REVISION_ID field of the CPU control status register (CSR). The C64x+ megamodule revision can be read from the REVISION field of the megamodule revision ID register (MM_REVID) located at address 0181 2000h.

The VARIANT field of the JTAG ID register (located at 0288 0814h) changes between silicon revisions. Table 2 lists the contents of the JTAG ID register for each revision of the device. More details on the JTAG ID register can be found in the device-specific data manual, *TMS320TCI6487/8 Communications Infrastructure Digital Signal Processor* (literature number SPRS358).

Table 1. Lot Trace Codes

LOT TRACE CODE (&&)	SILICON REVISION	COMMENTS
21	2.1	Silicon revision 2.1
20	2.0	Silicon revision 2.0
13	1.3	Silicon revision 1.3
12	1.2	Silicon revision 1.2
11	1.1	Silicon revision 1.1
10	1.0	Initial silicon revision

Table 2. Silicon Revision Variables

SILICON REVISION	CPU REVISION	C64X+ MEGAMODULE REVISION	JTAG ID REGISTER VALUE
2.1	CPU_ID = 10h REVISION_ID = 05h	Rev. 3.2 MM_REVID[REVISION] = 2h	0x4009 202Fh VARIANT = 0100b
2.0	CPU_ID = 10h REVISION_ID = 05h	Rev. 3.1 MM_REVID[REVISION] = 1h	0x3009 202Fh VARIANT = 0011b
1.3	CPU_ID = 10h REVISION_ID = 00h	Rev. 3.0 MM_REVID[REVISION] = 0h	0x2009 202Fh VARIANT = 0010b
1.2	CPU_ID = 10h REVISION_ID = 00h	Rev. 3.0 MM_REVID[REVISION] = 0h	0x1009 202Fh VARIANT = 0001b
1.1	CPU_ID = 10h REVISION_ID = 00h	Rev. 3.0 MM_REVID[REVISION] = 0h	0x1009 202Fh VARIANT = 0001b
1.0	CPU_ID = 10h REVISION_ID = 00h	Rev. 3.0 MM_REVID[REVISION] = 0h	0x0009 202Fh VARIANT = 0000b

1.3 Silicon Updates

Table 3 lists the silicon updates applicable to each silicon revision. For details on each advisory, see Section 2.2, Section 3.2, Section 4.2, Section 5.2 , Section 6.2, and Section 7.2 or click on the link below.

Advisory numbers in Table 3 are shown for each silicon revision as they were assigned and added to this document. However, if the design exceptions are still applicable, the advisories have been moved up to the latest silicon revision section. Therefore, advisory numbering may not be sequential.

Table 3. Silicon Revisions 1.0, 1.1, 1.2, 1.3, 2.0, 2.1 Updates

SILICON UPDATE ADVISORY	SEE	APPLIES TO SILICON REVISON					
		1.0	1.1	1.2	1.3	2.0	2.1
SRIO: Packet Forwarding Cannot Be Used With NREAD Response Packets Greater Than 16 Bytes	Advisory 1.0.1	X	-	-	-	-	-
I2C ROM Boot Fails When in Big Endian Mode	Advisory 1.0.3	X	-	-	-	-	-
Potential Random E-fuse Blow	Advisory 1.2.8	X	X	X	-	-	-
EDMA3CC COMPACTV Issue	Advisory 1.2.10	X	X	X	-	-	-
DSP SDMA/IDMA: Unexpected Stalling of SDMA/IDMA Access to L2 SRAM	Advisory 1.3.1	X	X	X	X	-	-
Potential Data Corruption on SCR Bridge	Advisory 1.3.2	X	X	X	X	-	-
Potential Insertion or Deletion of 2 Bits in SerDes Data Stream	Advisory 1.3.3	X	X	X	X	-	-
MAC EOI Register Write Causes Potential CPU Lockup	Advisory 1.3.4	X	X	X	X	-	-
I2C: Slave Boot Aborts	Advisory 1.3.6	X	X	X	X	-	-
RAC: Potential Corruption of RAC Statistics	Advisory 1.3.7	X	X	X	X	-	-
EMAC Boot Issue	Advisory 1.3.8	X	X	X	X	-	-
IP Block Containing CIC, CFGC, DTF, and IPC Registers Does Not Return Write Request Correctly	Advisory 1.3.9	X	X	X	X	-	-
DMA Access to L2 SRAM May Stall When the DMA Has Lower Priority Than the CPU	Advisory 1.3.12	X	X	X	X	-	-
DMA Corruption of L2 RAM Data	Advisory 2.0.6	-	-	-	-	X	-
L2 Victim Traffic Due To L2 Block Writeback During Any Pending CPU Request	Advisory 2.0.7	X	X	X	X	X	-
Potential SerDes Clocking Issue	Advisory 2.1.1	X	X	X	X	X	X
SRIO OUTBOUND_ACKID Field Not Read Correctly	Advisory 2.1.2	X	X	X	X	X	X
SRIO Port 0 Reset Affects Other Ports	Advisory 2.1.3	X	X	X	X	X	X
DMA Access to L2 SRAM May Stall When the DMA and the CPU Command Priority is Equal	Advisory 2.1.4	X	X	X	X	X	X
DMA Corruption of External Data Buffer	Advisory 2.1.5	X	X	X	X	X	X
L1P$ Miss May Block SDMA Accesses (Asymmetric L2 Configuration Only)	Advisory 2.1.6	X	X	X	X	X	X
SPLOOP CPU Cross-Path Stall	Advisory 2.1.7	X	X	X	X	X	X
DMA Corruption of L1D$ Allocation	Advisory 2.1.8	-	-	-	-	X	X

Table 3. Silicon Revisions 1.0, 1.1, 1.2, 1.3, 2.0, 2.1 Updates (continued)

SILICON UPDATE ADVISORY	SEE	APPLIES TO SILICON REVISON					
		1.0	1.1	1.2	1.3	2.0	2.1
Error Detection and Correction Incorrectly Reporting Error	Advisory 2.1.9	X	X	X	X	X	X
SRIO May Fail to Send Interrupt for Completed TX or RX Message	Advisory 2.1.10	X	X	X	X	X	X
Serial RapidIO Internal Digital Loopback is Not Always Stable	Advisory 2.1.11	X	X	X	X	X	X

2 Silicon Revision 2.1 Usage Notes and Known Design Exceptions to Functional Specifications

2.1 Silicon Revision 2.1 Usage Notes

Usage Notes highlight and describe particular situations where the device's behavior may not match presumed or documented behavior. This may include behaviors that affect device performance or functional correctness. These notes will be incorporated into future documentation updates for the device (such as the device-specific data manual), and the behaviors they describe will not be altered in future silicon revisions.

2.1.1 EMAC: SERDES PLL Lock

A LOCK_CTL register is missing in the *EMAC Subsystem SGMII Registers* section of the *TMS320TCI6487/8 DSP Ethernet Media Access Controller (EMAC)/ Management Data Input/Output (MDIO) User's Guide* (literature number SPRUEF0). The register is present in the device, but it is not documented.

The register is at address offset 0x88 in the EMAC Control Module Registers memory map (base address 0x02C8 1000). The LOCK_CTL register is shown in Figure 2 and described in Table 4.

31	1	0
Reserved		LOCK_EN
R-0		R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure 2. LOCK_CTL Register

Table 4. LOCK_CTL Register Field Descriptions

Bit	Field	Value	Description
31	Reserved		Reserved. Read as zero.
0	LOCK_EN		Lock Enable. This bit controls enabling of the SERDES lock function.
		0	SERDES lock function is disabled. The lock function does not asynchronously reset the SGMII block.
		1	SERDES lock function is enabled. The lock function asynchronously resets SGMII block.
			The value of the LOCK bit of the SGMII status register is valid only if the lock function is enabled. If the lock function is disabled, then the value of the LOCK bit is always read as 1.

The SGMII initialization has the following steps:

```
Uint32 * lock_ctl;

/* Soft reset SGMII and wait till the Reset is complete */
SGMII_REGS->SOFT_RESET = 0x00000001;
while (SGMII_REGS->SOFT_RESET != 0x00000000);

SGMII_REGS->CONTROL = 0x00000001;
SGMII_REGS->MR_ADV_ABILITY = 0x00000001;
SGMII_REGS->TX_CFG = 0x00000e21;
SGMII_REGS->RX_CFG = 0x00081021;
SGMII_REGS->AUX_CFG = 0x0000000b;
```

The SGMII AUX_CFG register enables power for the SERDES PLL. After the SERDES PLL is powered up, it takes 1 µs (for SERDES PLL power regulator to stabilize) + 200 REFCLKP/N (1.6 µs for 125-MHz REFCLK) cycles to lock in the required frequency. So, software has to run a loop for checking the status of the PLL lock before proceeding. The SGMII status register provides the status of the SERDES PLL (the LOCK bit shows whether it is locked or not).

The LOCK signal is a field (single bit) of the SGMII status register. It is pulled high at reset and remains high until the LOCK_EN field in the LOCK_CTL register is disabled. Only after enabling the LOCK_EN bit in the LOCK_CTL register can the LOCK bit in the SGMII status register reflect the correct status of the SERDES PLL lock.

2.1.2 User-Initiated Cache Coherence Operations Using Global Address Causes an Access Across the SCR

It has been found that if a user issues a manual writeback or writeback and invalidate command using the global address of the Core, a single 128-bit word access is issued out of the MDMA port across the SCR to the same core's SDMA port. Note that this does not happen on manual invalidate commands. Also, it does not happen when the local address is used instead of the global address.

NOTE: This behavior violates Advisory 1.3.1, *DSP SDMA/IDMA: Unexpected Stalling of SDMA/IDMA Access to L2 SRAM*. If performing these operations, ensure that only local addresses are used. For further details, see the *Deadlock Avoidance* section in Workaround Method 3 of Advisory 1.3.1.

2.1.3 Bootloader: Multicore Reset Control Enhancement

For silicon revisions 2.x on the TCI6487/8 device, an enhancement has been added to enable the user to control whether to take core1 and core2 out of reset in the bootloader.

Address 0x108FFFF8 is used as a flag to signal the bootloader whether to take core1 and core2 out of reset at the end of the boot. The bootloader initializes this address to 0. If this address is initialized to a non-zero value by the application code or the secondary bootloader after the bootloader finishes downloading the boot image, the first-level bootloader does not take core1 and core2 out of reset; otherwise, the first-level bootloader takes core1 and core2 out of reset.

If core1 and core2 are kept in the reset state after the first-level boot, the application code or the secondary bootloader in core0 can choose when to take core1 and core2 out of reset by setting the EVTPULSE4 bit (bit 4) of the C64x+ Megamodule core0's EVTASRT register to 1. This process is valid only once: writing 1, then writing 1 again does not bring core1 and core2 out of reset again. Core1 and core2 begin execution from their L2 RAM base address after this operation.

2.2 Silicon Revision 2.1 Known Design Exceptions to Functional Specifications

Table 5 lists the silicon revision 2.1 known design exceptions to functional specifications.

Table 5. Silicon Revision 2.1 Advisory List

Advisory 2.1.1 *Potential SerDes Clocking Issue*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details: An issue has been found in the SerDes interfaces that causes a SerDes clocking problem in normal functional operation. This problem will not occur when external pull-down is applied on the TCK pin (JTAG controller clock). SerDes are used in the Ethernet interface (EMAC), Serial RapidIO® interface (SRIO) and the Antenna Interface (AIF).

The TCK pin (JTAG controller clock) is internally assigned to an internal signal that is used by the SerDes macro. For the SerDes macro to get proper clocking in the normal functional operation, it needs the internal signal to be held low. However, there is an internal pull-up on the TCK, creating problems for SerDes operation. This problem exists on all SerDes interfaces.

Workaround: The TCK pin should be externally pulled down with an 1-kΩ resistor.

Advisory 2.1.2 *SRIO OUTBOUND_ACKID Field Not Read Correctly*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details: The OUTBOUND_ACKID field of the RIO_SP(n)_ACKID_STAT register should be updated by hardware each time a packet is sent out. The value should reflect the ACKID value to be used on the next transmit packet. This field is being updated by the hardware as expected. The field can also be written by the software and these writes also succeed. However, a hardware error prevents this field from being read. The OUTBOUND_ACKID always reads as zero. This problem does not cause any impact to link operation.

Workaround: There is no workaround for this advisory.

Advisory 2.1.3 *SRIO Port 0 Reset Affects Other Ports*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details: The SerDes for SRIO should allow the reset of individual 1X ports without affecting the state of the other operational ports. There are dedicated MMR bits to reset 1X ports, which are the BLKn_EN (n=5..8) at offsets 0x60 and 0x68. However, the BLK5_EN that controls reset for port 0 also resets all other ports. Therefore, it is impossible to reset port 0 without affecting all other ports.

Workaround: There is no workaround for this advisory.

Advisory 2.1.4 *DMA Access to L2 SRAM May Stall When the DMA and the CPU Command Priority is Equal*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details: The L2 memory controller in the GEM has programmable bandwidth management features that are used to control bandwidth allocation for all requestors. There are two parameters to control this, command priority and arbitration counter MAXWAIT values. Each requestor has a command priority and the requestor with the higher priority wins. However, there are also counters associated with each requestor that track the number of cycles each requestor loses arbitration. When this counter reaches a threshold (MAXWAIT), which is programmed by the user (or default value), the losing requestor gets an arbitration slot and wins for that cycle. There are four such requestors: CPU, DMA (SDMA and IDMA), user cache coherency operation, and global cache coherence. Global-coherence operations are highest priority, while user-coherence operations are lowest priority. However, there is active arbitration done for the CPU and the DMA (SDMA/IDMA) commands. The priority for DMA commands comes from an external master as part of the SDMA command or a programmable register, IDMA1_COUNT, in the GEM for IDMA commands. The priority for CPU accesses to L2 is in a programmable register, CPUARBU, in the GEM. For the default priority values, see Table 6.

More details on the bandwidth management feature can be found in the *C64x+ DSP Megamodule Reference Guide* (SPRU871).

Table 6. TCI6487/8 Default Master Priorities

MASTER	DEFAULT MASTER PRIORITIES (0 = Highest priority, 7 = Lowest priority)	PRIORITY CONTROL
EDMA3TCx	0	QUEPRI.PRIQx (EDMA3 register)
SRIO (Data Access)	0	PER_SET_CNTL.CBA_TRANS_PRI (SRIO register)
SRIO (Descriptor Access)	1	PRI_ALLOC.SRIO_CPPI
EMAC	1	PRI_ALLOC.EMAC
RAC Back-End (TCI6488 only)	7	RAC register
C64x+ Megamodule (MDMA port)	7	MDMAARBE.PRI (C64x+ Megamodule register)
C64x+ Megamodule (CPU Arbitration control to L2)	1	CPUARBU (C64x+ Megamodule register)
C64x+ Megamodule (IDMA channel 1)	0	IDMA1_COUNT (C64x+ Megamodule register)

NOTE: When the SDMA has finished sending all of its commands to the L2 controller, the C64x+ Megamodule drops the transfer priority down to 7 if no further commands are in the pipeline. This condition happens when there is a single-word access, a burst of <32B with no other SDMA commands pending or for the last 64B only of a burst that is >64B with no other SDMA commands pending. This effective priority level is what the L2 controller uses to arbitrate these SDMA commands with the CPU, irrespective of the master peripheral's actual programmed priority value. Therefore, priority 7 is not a valid priority level for the CPU. If, for any reason, this "demoted" transfer is still pending upon initiation of another transfer, it automatically inherits the priority of that new transfer and is pushed through such that it does not stall the new transfer.

The L2 memory controller is supposed to give equal bandwidth to the DMA and the CPU, by alternating between the two for arbitration. Instead, the L2 memory controller gives larger bandwidth allocation to the CPU accesses when the DMA and the CPU priorities are same. The CPU commands keep winning arbitration over the DMA as long as there are no other internal conditions (stalls, etc.) that force the DMA to win arbitration. This typically happens when CPU accesses keep the L2 memory controller busy every cycle, hence, the DMAs stall until the stream of CPU accesses completes. For example, if a continuous stream of L1D write misses to L2 keep the L2 memory controller busy every cycle, the DMAs stall for the entire duration of the write miss stream.

Workaround: Ensure that the CPU is at a different priority than the DMA commands to L2. The priority for CPU accesses to L2 is in a programmable register, CPUARBU, in the GEM. However, lowering the CPU priority may impact the performance since, in case of contention, the CPU accesses to L2 can get stalled due to DMA accesses. The CPUARBU should not be set to 7 (see Note above).

The recommended workaround is to pick a single priority value for the CPU accesses to L2 only. That leaves the remaining 6 priority levels free for other accesses to L2 (see Table 7).

Table 7. TCI6487/8 Valid Priority Settings

CPU PRIORITY	ALLOWED SDMA PRIORITIES
0	1-7
1	0, 2-7
2	0-1, 3-7
3	0-2, 4-7
4	0-3, 5-7
5	0-4, 6-7
6	0-5, 7
7(1)	-

(1) Do not set CPU priority to 7.

Advisory 2.1.5 *DMA Corruption of External Data Buffer*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details:

Under a specific set of circumstances, an L1D snoop-write updates an unintended L1D cache line. This leads to a corrupted line in L1D and can lead directly to program misbehavior. If the corrupted line is then modified by a CPU write access, a subsequent victim writeback from L1D could commit the corrupted line to lower levels of memory. Two key requirements for this issue are:

- The DMA writes to buffers in UMAP1 only (see below).
  - This must be cached and unmodified in L1D (read by the CPU but not yet written to it).

  The L2 memory is typically shared across the two unified memory access ports, UMAP0 and UMAP1. This issue occurs only if the buffer is located in UMAP1. For the UMAP1 allocation on the TCI6487/8 devices, see Table 8.

Table 8. TCI6487/8 UMAP1 Allocation

L2 CONFIGURATION	UMAP1	ADDRESS RANGE	AFFECTED
Symmetric	N/A	N/A	No
Asymmetric:			
Core 0	RAM	0x00800000 - 0x008FFFFF	Yes
Core [2:1]	N/A	N/A	No

- The CPU reads from an external, cacheable address.
  - UMAP0 and UMAP1 are the two ports on the C64x+ Megamodule used to connect the L2 Memory controller and the physical RAMs. For the UMAP1 allocation on the TCI6487/8 devices, see Table 8
  - For information on L1D cache coherence protocol, see section 3.3.6, *Cache Coherence Protocol*, in the *C64x+ DSP Megamodule Reference Guide* (SPRU871).
  - DMA in the following description refers to all non-CPU requestors. This includes IDMA, EDMA, and any other master in the system.

Under the specific set of circumstances listed below, a snoop-write updates an L1D cache line other than the one intended. This leads to a corrupted line in L1D. Corruption only happens when the buffer in UMAP1 is cached in L1D while the CPU is consuming external, cacheable data.

The prerequisite before the window where the issue occurs is:

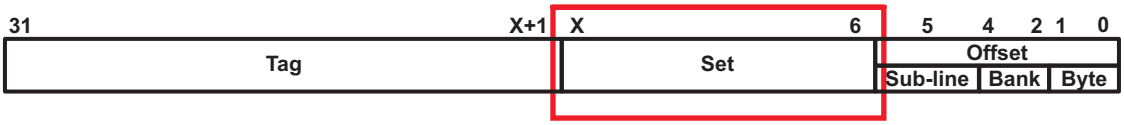
- The CPU reads an L2 location in UMAP1 and has not modified (written) to the same location before the window where the issue occurs.
  - Because of this, a 64B cache line is allocated clean in L1D (referred to here as Cache Line A).

The following steps must all occur concurrently to see the issue (note that the concurrency is within the cache subsystem, so events visible at the CPU or the DMA are not occurring during the same exact cycle):

1. The L1D is currently processing a snoop request or some other request that prevents it from accepting new snoops. This could have been caused by any of the following that is still being processed from previous actions:
   - DMA read/write
   - L1D read/invalidate
   - L1D read + victim
2. The DMA writes to Cache Line A, mentioned in the prerequisite above. This means that it is not necessarily the same exact address, but must be within the same 64B cache line.
   - As a result, a snoop-write request is generated but it is blocked because the L1D is still busy with Step 1.

3. The CPU reads from a cacheable, external memory (e.g., DDR) that is a set match to Cache Line A (referred to here as Cache Line B).

   Determining if two addresses are a set match can be done by comparing certain bits of two addresses. The mapping of an address to a location in L1D cache is shown in Figure 3.



The value X is determined by how large the L1D cache is in the particular application (see Table 9).

Figure 3. L1D Cache Address Mapping

Table 9. Value of X for L1D Cache

AMOUNT OF L1D CACHE	X BIT POSITION
0KB	N/A
4KB	10
8KB	11
16KB	12
32KB	13

If you use the default configuration, 32KB, as an example, bits [13:6] are a set match if they are identical in two different addresses. Some examples of set matches are shown below:

- 0x0080 2A80 00000000100000000010101010000000
- 0x8000 2A80 10000000100000000010101010000000
- 0x0080 2A8A 00000000100000000010101010001010
- This results in a cache miss from the CPU for an external address and sends a read request to L2 cache for the line (and possibly to the external source on an L2 cache miss or if no L2 cache is present).

The results of the above cause the following:

L2 sends both the return data for the L1D read miss request (response of Step 3 above) and the data for the snoop-write (response of Step 2 above). The L1D commits the snoop-write data after the L2 return data.

As a result, L1D now holds the wrong data for the external address (Cache Line B) and commits the data to cache. Cache Line B remains marked "clean." If the program does not write to the uncorrupted portion of the line and does not read the corrupted portion of the line, the corruption goes unnoticed. If the program writes to the uncorrupted portion of the line, the corrupted data gets written back to L2 cache and/or external memory. Otherwise, the corruption disappears when L1D discards the line.

Cache lines holding external addresses are the only cache lines that exhibit corruption. Corruption only happens when DMA buffers in UMAP1 get cached in L1D. Additionally, corruption only happens when the DMA buffer is clean, meaning that it gets discarded without generating a victim. Thus, this affects buffers where the DMA writes and the CPU reads. It does not affect buffers that the CPU only writes and/or DMA only reads.

One can identify this issue unambiguously by examining the corrupted memory range in CCStudio using the cache tag viewer. The corrupted data shows up in the include L1D view in a memory window, but not in the exclude L1D view. The cache tag viewer should indicate that the line is also "clean" and the corrupt data should also be visible in its intended destination, which must be in UMAP1 and map to the same L1D set as the corrupted line.

Figure 4 shows the flow of these operations, the incorrect order that causes the issue, and the correct order. The blue line is Cache Line A and the yellow line is Cache line B.

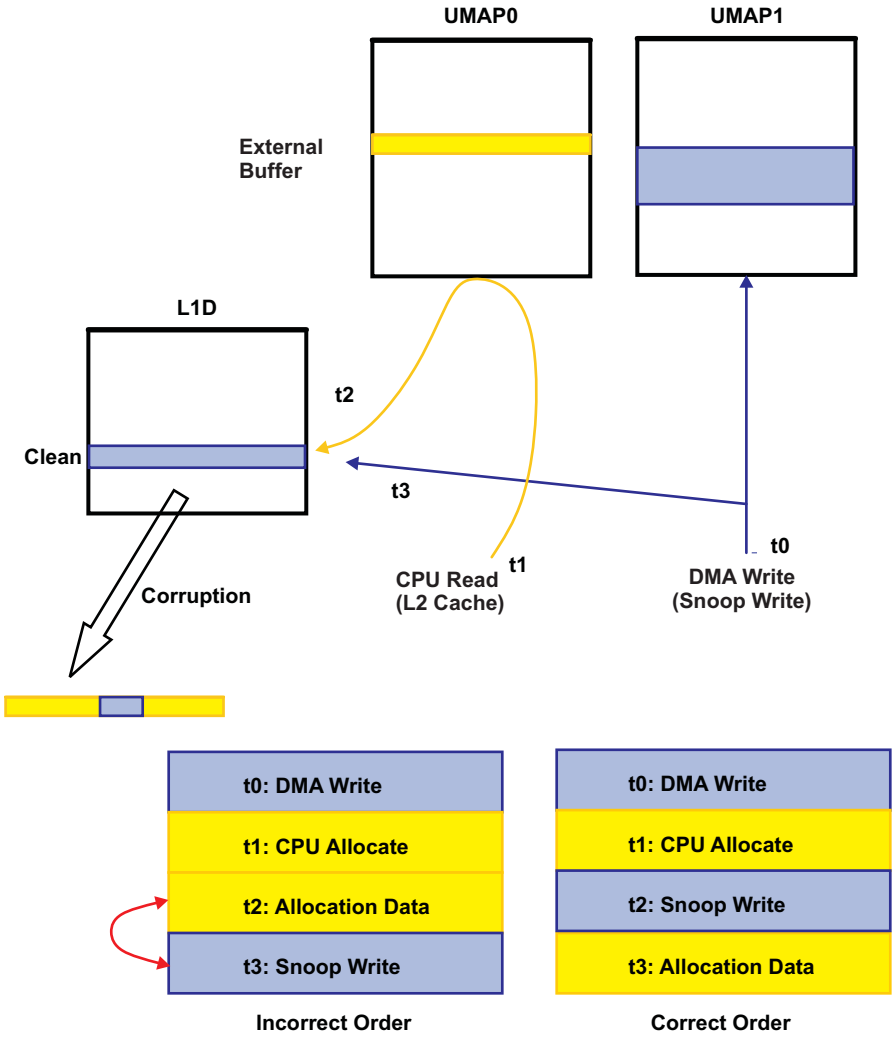


Figure 4. Cache Line Operations Flow

Workarounds: In the issue described above, all of the conditions must be true for the issue to occur. The workarounds focus on picking one of the conditions and removing it so that you do not need to worry about the other conditions.

TI proposes starting with workaround 1 as an immediate fix. The other workarounds that follow may provide a solution with reduced overhead and/or simplified implementation, depending on the customer's system.

Workaround 1: Write Back and Invalidate DMA Buffers

L1D corruption occurs when the DMA writes to a buffer in UMAP1 that is also cached in L1D at the same time the L1D is discarding the buffer. Thus, this affects buffers where the DMA writes and the CPU reads. It does not affect buffers that the CPU only writes and/or the DMA only reads.

To prevent this sort of race condition, programs should discard inbound DMA buffers in UMAP1 immediately after use and keep a strict policy of "buffer ownership" such that a given buffer is owned only by the CPU or the DMA at any given time.

This model assumes the following:

1. The DMA fills the buffer during a period when the CPU does not access it.
2. The DMA engine or other mechanism signals to the CPU that it has finished filling the buffer.
3. The CPU operates on the buffer, reading and writing to it, as necessary. The DMA does not access the buffer at this time.
4. The CPU relinquishes control of the buffer so that DMA may refill it. (This may be an implicit step in many implementations if the period between refills is much longer than the time it takes the CPU to process the refilled buffer.)

To implement this workaround, programmers must write back and invalidate the buffer from L1D cache after Step 3 and before Step 4. This eliminates the prerequisite for the issue to occur should another DMA, in the future, be a set match to the reads that the CPU just performed.

There are multiple mechanisms for doing this, but the most straightforward is to use the L1D block cache writeback-invalidate mechanism via L1DWIBAR/L1DWIWC.

The recommended implementation of this workaround requires calling the l1d_block_wbinv.asm function (see the L1D Block Writeback-Invalidate Routine below). It can be invoked as follows:

```
void l1d_block_wbinv(void *base, size_t byte_count);
```

To writeback-invalidate a C array, one could then do:

```
/* ... */

l1d_block_wbinv(&array[0], sizeof(array));
```

Programmers should insert such a call whenever the code is done with a particular DMA buffer in UMAP1, before the DMA controller can refill it. The l1d_block_wbinv() function is non-interruptible. Its overhead is proportional to the size of the buffer.

NOTE:

1. To ensure complete effectiveness, DMA buffers must always start on an L1D cache-line boundary (64-byte boundary) and occupy a multiple of 64 bytes. This may require increasing the size of some DMA buffers slightly. This is necessary to prevent accesses to an unrelated buffer or variable from bringing a portion of the DMA buffer back into the L1D cache.
2. If the buffer under consideration is a small number of read-only flags to the CPU, then Workaround 4 may be more applicable.

L1D Block Writeback-Invalidate Routine

```
;; ======================================================================== ;;
;; L1D Block Writeback-Invalidate                                           ;;
;;                                                                          ;;
;;   l1d_block_wbinv(void *base, size_t byte_count);                        ;;
;;                                                                          ;;
;; Performs a block writeback-invalidate from L1D to L2. It can be used     ;;
;; on any address range (L2 or external), but it only operates on L1D       ;;
;; cache.                                                                   ;;
;;                                                                          ;;
;; Maximum block size is 256K. Exact maximum byte count depends on the      ;;
```

```
;; alignment of the block.                                                  ;;
;;                                                                         ;;
;; Interrupts are disabled during the block writeback operation.           ;;
;; ======================================================================= ;;

     .asg 0x01844030, L1DWI          ; L1D Block Wb-Inv; BAR at 0, WC at 1
     .global _l1d_block_wbinv
     .text
     .asmfunc
_l1d_block_wbinv:

     MVC DNUM, B0               ; \_ Get global alias prefix
     ADDK 0x10, B0              ; /
     SHRU A4, 24, B2            ; Get prefix from address
     CMPEQ B0, B2, B0           ; Check if address prefix is global
[ B0] EXTU A4, 8, 8, A4         ; Remove global prefix from address
     MVKL L1DWI, B6             ;

    CLR A4, 0, 5, A1            ; Align to L1D cache line boundary
 || ADD A4, B4, B1              ; Compute end of buffer

    ADDK 63, B1                 ; \_ Round to next L1D cache line
    CLR B1, 0, 5, B1            ; /

    SUB B1, A1, B1              ; Count cache-line span in bytes
 || MVKH L1DWI, B6              ;

    SHR B1, 2, B1               ; Convert to "word count"
 || DINT                        ; Disable interrupts

    STW A1, *B6[0]              ; Store base address
    STW B1, *B6[1]              ; Store word count

    ; Note: The following loop is intentionally low-rate to avoid
    ; interfering with the block writeback operation.
loop: LDW *B6[1], B1            ; Read remaining word-count
    NOP 4
[ B1] BNOP loop, 5              ; Loop until done

    RINT                        ; Reenable interrupts
    RETNOP B3, 5                ; Return to caller

.endasmfunc

;; ======================================================================= ;;
;; End of file: l1d_block_wbinv.asm                                        ;;
;; ======================================================================= ;;
```

Workaround 2: Make DMA Buffers Dirty After Use

The errant snoop-write occurs only when the DMA buffer in L1D has not been modified. This is due to the additional snoop-checking mechanisms associated with tracking victims as they leave L1D.

Therefore, another workaround is to mark DMA buffers as "dirty" before releasing them. This generates additional victims whenever the buffer gets pushed out of L1D. It also blocks the errant snoop-write.

This workaround assumes a similar model to Workaround 1, but uses the make_dirty() function (see the Mark Buffer Dirty Routine below). The make_dirty() function reads one byte from each cache line of the buffer and writes the same value back to it immediately.

The function is called as follows:

```
void make_dirty(void *base, size_t byte_count);
```

Mark Buffer Dirty Routine

```
;; ======================================================================= ;;
;; Make a block of data "dirty" in L1D                                     ;;
;;                                                                         ;;
;;   make_dirty(void *base, size_t byte_count);                            ;;
;;                                                                         ;;
;; ======================================================================= ;;
```

```
    .global _make_dirty
    .text
    .asmfunc
_make_dirty:
    ADDK 63, B4
    SHR  B4, 6, B4
    MVC  B4, ILC
    MVK  64, A5
    MVK  64, B5
    MV   A4, B4
    NOP
    SPLOOP 1
    LDBU *A4++[A5], A1
    NOP  4
    MV.L A1, B1
    STB  B1, *B4++[B5]
    SPKERNEL

    RETNOP B3, 5

    .endasmfunc

;; ========================================================================= ;;
;; End of file: make_dirty.asm                                               ;;
;; ========================================================================= ;;
```

NOTE:

1. This workaround is ***not*** acceptable if the DMA could be writing to the buffer at the same time make_dirty() function gets called. The process of making the cache line dirty requires reading and writing within the buffer and, so, the CPU's writes could overwrite the inbound data from the DMA.
2. This workaround may cause the application to be affected by the issue described in Advisory 2.0.6, DMA Corruption of L2 RAM Data.

Workaround 3: Do Not Cache Data From External Memory in L1D

If your program only makes a small number of data accesses to external memory, consider marking the data portions of external memory as non-cacheable. This prevents caching copies of external memory in L1D cache.

Alternately, to prevent the line from allocating in L1D, freeze the L1D cache around each access to an external address. The long_dist_load_word function (see the Long Distance Load Word Routine below) is suitable for isolated accesses. For larger accesses, such as reading a block, other techniques may be more appropriate.

The incorrect snoop-write only occurs when the L1D read miss involved is to an external address. The snoop-write corrupts the newly cached copy in L1D. If all accesses to external data memory are non-cacheable or occur while L1D is frozen, this prevents copies from being stored in L1D.

Long Distance Load Word Routine

```
;; ========================================================================= ;;
;; Long Distance Load Word                                                   ;;
;;                                                                           ;;
;;   int long_dist_load_word(volatile int *addr)                             ;;
;;                                                                           ;;
;; This function reads a single word from a remote location with the L1D     ;;
;; cache frozen. This prevents L1D from sending victims in response to       ;;
;; these reads, thus preventing the L1D victim lock from engaging for the    ;;
;; corresponding L1D set.                                                    ;;
;;                                                                           ;;
;; The code below does the following:                                        ;;
;;                                                                           ;;
;;   1. Disable interrupts                                                   ;;
;;   2. Freeze L1D                                                           ;;
;;   3. Load the requested word                                              ;;
;;   4. Unfreeze L1D                                                         ;;
```

```
;;   5. Restore interrupts                                                  ;;
;;                                                                          ;;
;; Interrupts are disabled while the cache is frozen to prevent affecting   ;;
;; the performance of interrupt handlers. Disabling interrupts during       ;;
;; the long distance load does not greatly impact interrupt latency,        ;;
;; because the CPU already cannot service interrupts when it's stalled by   ;;
;; the cache. This function adds a small amount of overhead (~20 cycles)    ;;
;; to that operation.                                                       ;;
;;                                                                          ;;
;; ======================================================================== ;;

     .asg 0x01840044, L1DCC ; L1D Cache Control
     .global _long_dist_load_word
     .text
     .asmfunc
; int long_dist_load_word(volatile int *addr)
_long_dist_load_word:
     MVKL L1DCC,  B4
     MVKH L1DCC,  B4
||   DINT                ; Disable interrupts
||   MVK 1,   B5
     STW B5, *B4         ; \_ Freeze cache
     LDW *B4, B5         ; /
     NOP 4
     SHR B5, 16, B5      ; POPER -> OPER
||   LDW *A4, A4         ; read value remotely
     NOP 4
     STW B5, *B4         ; \_ Restore cache
     RET B3
||   LDW *B4, B5         ; /
     NOP 4
     RINT                ; Restore interrupts
    .endasmfunc

;; ======================================================================== ;;
;; End of file: ldld.asm                                                    ;;
;; ======================================================================== ;;
```

Workaround 4: Allocate DMA buffers in L1D RAM or UMAP0

If possible, move DMA buffers that the CPU reads directly out of UMAP1 to either UMAP0 or L1D RAM. DMA buffers that the CPU does not access directly can remain in UMAP1 safely, as these do not generate snoops.

If your set of in-bound DMA buffers does not fit in L1D RAM and UMAP0 statically, consider paging buffers from UMAP1 to either UMAP0 or L1D RAM. That is, allow the DMA to write to buffers in UMAP1 freely, but never read them directly from the CPU. Instead, use the IDMA to copy a buffer from UMAP 1 to either UMAP0 or L1D RAM before using it.

The IDMA1 utility functions (see the IDMA Channel 1 Block Copy Routine below) can be used for copying data with the IDMA controller.

IDMA Channel 1 Block Copy Routine

```
;; ======================================================================== ;;
;; TEXAS INSTRUMENTS INC.                                                   ;;
;;                                                                          ;;
;; Block Copy with IDMA Channel 1                                           ;;
;;                                                                          ;;
;; REVISION HISTORY                                                         ;;
;;   13-Feb-2009 Initial version . . . . . . . . . . . J. Zbiciak          ;;
;;                                                                          ;;
;; DESCRIPTION                                                              ;;
;;   The following macro functions are defined in this file:                ;;
;;                                                                          ;;
;;     idma1_copy(void *dst, void *src, int word_count)                     ;;
;;     idma1_wait(IDMA_PEND or IDMA_ACTV)                                   ;;
;;                                                                          ;;
;; NOTE: The last arg is WORD count, not byte count. 1 word = 4 bytes.      ;;
;;                                                                          ;;
;; ------------------------------------------------------------------------ ;;
;; Copyright ©) 2009 Texas Instruments, Incorporated.                      ;;
```

```
;; All Rights Reserved.                                                      ;;
;; ======================================================================== ;;

     .asg 0x01820100,                 IDMA1_STATUS
     .asg 0x01820108,                 IDMA1_SOURCE
     .asg 0x0182010C,                 IDMA1_DEST
     .asg 0x01820110,                 IDMA1_COUNT
     .asg 0x01820100,                 IDMA1_BASE
     .asg (IDMA1_STATUS - IDMA1_BASE), OFS_IDMA1_STATUS
     .asg (IDMA1_SOURCE - IDMA1_BASE), OFS_IDMA1_SOURCE
     .asg (IDMA1_DEST - IDMA1_BASE),   OFS_IDMA1_DEST
     .asg (IDMA1_COUNT - IDMA1_BASE),  OFS_IDMA1_COUNT

;; ------------------------------------------------------------------------ ;;
;; IDMA1_COPY: Copy a block of words to dst from src with IDMA channel 1    ;;
;;                                                                          ;;
;; USAGE                                                                    ;;
;;   idma1_copy( <dest address>, <source address>, <word count>)            ;;
;;                                                                          ;;
;;   Both source and destination addresses must be word aligned.            ;;
;;                                                                          ;;
;;   The IDMA gets issued at top priority. Only bits 13:0 of the word       ;;
;;   count are significant.                                                 ;;
;; ------------------------------------------------------------------------ ;;

     .global _idma1_copy
     .asmfunc
_idma1_copy:
;    Point to IDMA channel 1's base
     RET B3              ; return; also protect from interrupts
||   MVKL IDMA1_SOURCE, A7
     MVKH IDMA1_SOURCE, A7

;    Write second argument to "source" register
     STW B4, *A7++(IDMA1_DEST - IDMA1_SOURCE)

;    Write first argument to "destination" register
     STW A4, *A7++(IDMA1_COUNT - IDMA1_DEST)

;    Write last argument to "count" register.
     EXTU A6, 18, 16, A6        ; truncate word count to 14 bits
     STW A6, *A7
     .endasmfunc

;; ------------------------------------------------------------------------ ;;
;; IDMA1_WAIT: Wait for IDMA "pend" or "actv" slot to free up.              ;;
;;                                                                          ;;
;; USAGE                                                                    ;;
;;   idma1_wait(IDMA_PEND) Waits for just PEND to be 0                      ;;
;;   idma1_wait(IDMA_ACTV) Waits for ACTV (and PEND) to be 0                ;;
;;                                                                          ;;
;; NOTE                                                                     ;;
;;   IDMA_PEND = 2                                                          ;;
;;   IDMA_ACTV = 3                                                          ;;
;;                                                                          ;;
;; ------------------------------------------------------------------------ ;;
     .global _idma1_wait
     .asmfunc
_idma1_wait:
     MVKL IDMA1_STATUS, A6
     MVKH IDMA1_STATUS, A6
||   MVK 1, A0
loop?:
  [ A0] LDW *A6, A0
||[ A0] BNOP.1 loop?, 4
;    The 'AND' below is safe because IDMA never returns 10b in 2 LSBs
     AND.L A4, A0, A0

     RETNOP B3, 5
     .endasmfunc

;; ======================================================================== ;;
;; End of file: idma1_util.asm                                              ;;
;; ======================================================================== ;;
```

Advisory 2.1.6 *L1P$ Miss May Block SDMA Accesses (Asymmetric L2 Configuration Only)*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details: This advisory is an update to Advisory 1.3.1 and Advisory 2.0.7 in this document. Advisory 1.3.1 and Advisory 2.0.7 list the following five blocking conditions to trigger an SDMA/IDMA stall:

1. Bursts of writes to non-cacheable locations.
2. L1D read miss generating victim traffic to L2 (cache or SRAM) or external memory.
3. L1D read request missing L2 (going external) while another L1D request is pending.
4. L2 victim traffic to external memory during any pending L1D request.
5. L2 victim traffic due to L2 block writeback during any pending CPU request.

NOTE: Items 1, 2, 3, and 4 shown in the list above and in Table 10 below are actually labeled as 1, 2a, 2b, and 2c in Advisory 1.3.1. Item 5 is described in Advisory 2.0.7.

This advisory covers one more blocking condition:

6. L1P$ miss may stall SDMA accesses.

For silicon revisions 1.0, 1.1, 1.2, and 1.3 that contain the original SDMA/IDMA blocking errata, this is a sixth way to encounter the issue in addition to the previously communicated five errata conditions in Advisory 1.3.1 and Advisory 2.0.7.

No additional deadlock risk potential is created by the addition of the new condition to silicon revisions 1.0, 1.1, 1.2, and 1.3 that currently contain the SDMA/IDMA blocking conditions 1-4. This means that this issue can lead to a deadlock in the same manner that the other four conditions can. On silicon revisions 2.0 and 2.1, without the original stall conditions 1-4, this creates a deadlock condition that is identical to the previous revisions.

Table 10. Stall Conditions on Silicon Revisions

SILICON REVISIONS	STALL CONDITIONS					
	1	2	3	4	5	6
1.0	YES	YES	YES	YES	YES	YES
1.1	YES	YES	YES	YES	YES	YES
1.2	YES	YES	YES	YES	YES	YES
1.3	YES	YES	YES	YES	YES	YES
2.0	NO	NO	NO	NO	YES	YES
2.1	NO	NO	NO	NO	NO	YES

Under certain conditions, L2 accesses to external memory resulting from an L1P$ miss can block SDMA/IDMA accesses during CPU/DMA requests. There are several transactions that must occur to cause an SDMA/IDMA to stall because of this condition:

1. A DMA access to UMAP0.

   Note that this transfer is not needed to see a fail on the device. A fail may occur only with transactions 2-5.
2. An L1D$ read miss from UMAP0.

   Note that if the software is currently running in L1D$ freeze mode during this transaction, transaction 1 is not needed to reproduce this issue.
3. An L1D$ write or victim to UMAP1. This happens as a result of one of the following:
   - An L1D victim (through L1D writeback or writeback-invalidate) to UMAP1.
   - An L1D read+victim (through L1D read miss resulting in a writeback) to any L2.

   The victim generated still needs to go to UMAP1. The reason that the L1D$ read can be to any L2 address (UMAP0 or UMAP1) is that there is no way of knowing if the least recently used cache line that will be evicted is in UMAP0 or 1.

- An L1D write miss (write-through to an uncached line).

4. An L1P$ miss that results in an L2 access to external memory (an L2 victim can create a deadlock or preceding long distance write).

   This step may not be necessary if a long-distance write to external memory is currently pending.

5. An SDMA access to UMAP1.

   It is also important to note that without item 5, this issue does not exist. That means that if the resolution of the pipeline is completed before item 5, then the issue is not seen.

   If an SDMA access to UMAP0 occurs before transaction item 5, the pipeline is flushed and this issue is not seen.

The SDMA in item 1 sets up a bank conflict for the L1D$ read in item 2. (Note that this transfer is not needed to see a fail on the device. A fail may occur only with transactions 2-5.) The L1D$ allocate in item 2 prevents the L1D$ write/victim (item 3) from advancing, so it is stuck in the pipeline. This occurs at the same time as an L1P$ allocate that also results in an L2 access to external memory (item 4), which is also in the same pipeline stage as the L1D$ write/victim (item 3). At this point, the L1P$ allocate (item 4) advances to the next pipeline stage but the L1D$ write/victim (item 3) is still stuck waiting on the L1D$ allocate (item 2). This now sets up the pipeline for the stall condition, which is actually triggered by an SDMA to UMAP1 (item 5). This is what causes further SDMAs to stall. After the L1P$ allocate (item 4) is complete, item 2 resolves, allowing item 3 to resolve, thus, freeing the SDMA pipeline again. Therefore, the stall is effectively for the length of the L1P$ allocate in item 4.

Note that the above four conditions do not guarantee that you will see a stall; it may stall depending on the timing between the transactions. Items 2 and 3 must occur within two CPU cycles of each other and items 3 and 4 must occur within five CPU cycles of each other. Figure 5 shows the timing relationship.

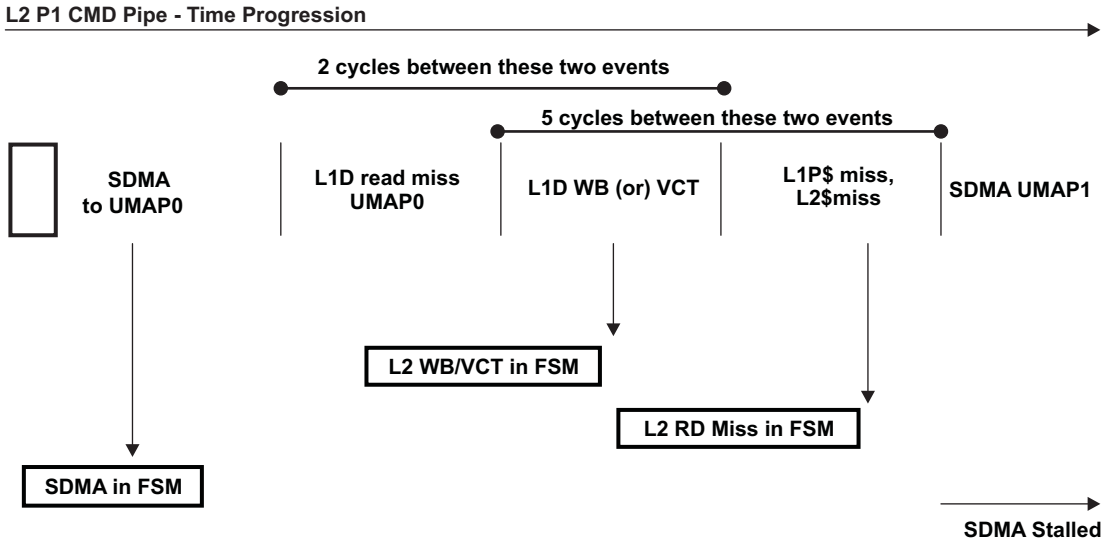


Figure 5. Timing Between Transactions

Workarounds:

Workaround 1: Leave in previous SDMA/IDMA stall workarounds (for devices with the original SDMA/IDMA stall)

For silicon revisions 1.0, 1.1, 1.2, and 1.3 that were already affected with the other four conditions of the SDMA/IDMA stall issue from Advisory 1.3.1, there is no additional workaround needed. If all of the deadlock avoidance steps listed in Advisory 1.3.1 have been followed, there is no risk for a deadlock because of this issue. Methods to reduce stalling due to this issue are also already covered in Advisory 1.3.1.

For silicon revision 2.0 that fixed the initial four conditions of SDMA/IDMA stall issue, the deadlock avoidance steps that are already listed in Advisory 1.3.1 for previous revisions of silicon should be followed to ensure that there is no chance of a deadlock. The workarounds to avoid stalls are also the same as communicated in previous revisions of the device with the issue.

Workaround 2: Do not place program code in external memory

This issue can be avoided by either ensuring that all program code is in L1P or L2 SRAM or SL2 SRAM. This eliminates the possibility of creating an L1P$ miss that generates an L2 read from external memory.

Workaround 3: Allocate all CPU-writeable DMA buffers/variables in UMAP0 or L1D RAM

NOTE: DMA in this case refers to EDMA and other masters external to the C64x+ Megamodule.

If possible, move DMA buffers that are also writeable by the CPU to completely reside in UMAP0 or L1D RAM. This prevents SDMA traffic to multiple UMAP ports.

Workaround 4: Allocate CPU data buffers/variables in UMAP0

If possible, move CPU data buffers/variables out of UMAP1 to UMAP0. This eliminates the CPU data accesses to/from UMAP1.

Workaround 5: Allocate CPU-readable data buffers/variables in UMAP1

NOTE: Since the L2$ is located in UMAP0, this workaround assumes that L2$ is disabled.

If possible, move CPU-readable data buffers/variables out of UMAP0 to UMAP1. This eliminates the CPU data reads from UMAP0. CPU writes are OK to UMAP1.

Advisory 2.1.7 *SPLOOP CPU Cross-Path Stall*

Revision(s) Affected: 2.1 and earlier

Details: If the following three rules are met, a stall is seen when an SPKERNEL instruction is executed.

1. **Cross-path instruction rule:** An instruction reading a register via the cross path in the first cycle after SPKERNEL instruction.
2. **Data dependence rule:** An instruction in the SPLOOP body that writes to the above cross-path read register. This instruction can be anywhere in the SPLOOP body.
3. **Functional unit rule:** No instruction in parallel with the SPKERNEL instruction that uses the same functional unit as the cross-path read instruction mentioned in rule 1 above.

This results in a one CPU cycle stall for each iteration of the loop. The following are three examples of code that are affected by this issue:

Example 1

```
SPLOOP 1
MV .S1 A0, A1 ;stalls every iteration due to MV after loop
SPKERNEL
MV .S2X A1, B2
```

Example 2

```
PLOOP 14
MV .S1 A0, A1 ;stalls every iteration due to MV after loop
NOP 9
NOP 9
NOP 9
NOP 9
SPKERNEL
MV .S2X A1, B2
```

Example 3

```
SMV .S1 A0, A1 ;stalls every iteration due to MV after loop
SPKERNEL
||NEG .L2 B3, B4 ;Qualifies for rule 3, functional unit rule
MV .S2X A1, B2
```

The following three examples are not affected by this issue:

Example 1

```
;No stalls: No cross path in instruction after SPKERNEL
SPLOOP 1
MV .S1 A0, A1
SPKERNEL
MV .S1 A1, A2
```

Example 2

```
;No stalls: A1 not written to in loop body
SPLOOP 1
MV .S1 A0, A2
SPKERNEL
MV .S2X A1, B2
```

Example 3

```
;No stalls: Instruction in parallel with SPKERNEL prevents bug since
;it's in the same unit as the instruction that uses the cross-path.
SPLOOP 1
MV .S1 A0, A1
SPKERNEL
||NEG .S2 B3, B4 ;masks the bug
MV .S2X A1, B2
```

Workaround(s): The way SPLOOP code is scheduled is controlled by the compiler. Therefore, there are no direct workarounds for non-assembly source code. There are new revisions of the latest compilers that ensure that these three conditions are never met. The following compiler releases include the fix:

- 6.0.25 or later
- 6.1.15 or later
- 7.0.2 or later
- 7.1.0B2 or later
- 7.2.0A or later.

Advisory 2.1.8 *DMA Corruption of L1D$ Allocation*

Revision(s) Affected: 2.1 and 2.0

Details: Under a specific set of circumstances, a snoop-write updates unintended data being allocated into L1D$ from external, cacheable memory. This can lead directly to program misbehavior. If that line is then modified by CPU accesses, a subsequent victim writeback from L1D could commit this corrupted line to lower levels of memory. The key requirements for this issue are:

- Two clean lines in L1D$.
  - This means that a CPU has read two L2 or external, cacheable addresses and has not modified them.
- One more allocated line in L1D$ that can be clean or dirty.
  - Dirty means that a CPU has read and written to any L2 or external, cacheable address.
- Two more parallel CPU reads (occurring in the same CPU cycle).
  - One of the reads must create an L2$ hit (implying an external, cacheable address) and must be a set match to one of the clean lines already in L1D$.
  - The other can be from an L2 SRAM address or an external, cacheable address and must be a set match to the L1D$ cache line mentioned above as clean or dirty.
- Two DMA writes to buffers in L2 SRAM that are a set match to the two clean lines in L1D$.

NOTE:

1. For information on L1D cache coherence protocol, see section 3.3.6, *Cache Coherence Protocol*, in the *C64x+ DSP Megamodule Reference Guide* (SPRU871).
2. The DMA in the following description refers to all non-CPU requestors. This includes IDMA, EDMA, and any other master in the system.

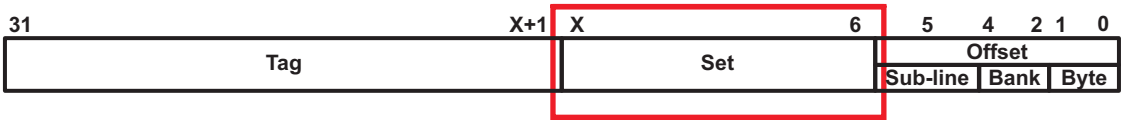
Under a specific set of circumstances listed below, a snoop-write results in data corruption of L1D$. The issue occurs when there is a DMA to L2 for one of the allocated (clean) lines that is also in the process of being replaced by an allocation from external, cacheable memory (implying there was a set match between the two); this is along with another allocation and a DMA to the other allocated (clean) line. L2 sends the DMA requests as snoop-writes to the L1D cache. When the error occurs, the line the second snoop-write was destined for has already been replaced by the allocation from external, cacheable memory. The logic to kill the snoop-write did not get sensitized and the snoop-write ends up corrupting the line that was allocated. Subsequent writes to the corrupted line cause this to get committed to lower levels of memory.

The prerequisite before the window where the issue occurs is:

- The CPU reads two L2 locations that are not a set match to each other and have not been modified since then (CPU/DMA has not written to it). For a description on how to determine if you have a set match or not, see below.
  - These are now two separate 64B cache lines allocated and clean in L1D (referred to here as Cache Lines B and E).
- The CPU reads another L2 location that is not a set match to Cache Lines B and E. It does not matter whether this particular cache line is modified or not before the issue window arrives.
  - Because of this, another 64B cache line is allocated in L1D as clean or dirty (referred to here as Cache Line A).
  - Note that both ways for this particular set must be occupied. It may require more than one read to this particular cache set.

How to determine if two addresses are a set match:

Determining if two addresses are a set match can be done by comparing certain bits of two addresses. The mapping of an address to a location in L1D cache is shown in Figure 6.



The value X is determined by how large the L1D cache is in the particular application (see Table 11).

Figure 6. L1D Cache Address Mapping

Table 11. Value of X for L1D Cache

AMOUNT OF L1D CACHE	X BIT POSITION
0KB	N/A
4KB	10
8KB	11
16KB	12
32KB	13

If you use the default configuration, 32KB, as an example, bits [13:6] are a set match if they are identical in two different addresses. Some examples of set matches are shown below:

- 0x0080 2A80 00000000100000000010101010000000
- 0x8000 2A80 10000000100000000010101010000000
- 0x0080 2A8A 00000000100000000010101010001010

The following steps must all occur in a very tight window to see the issue:

1. The DMA writes to Cache Line E. This means that it is not necessarily the same exact address, but within the same 64B cache line.
   - As a result, a snoop- write request is generated.
2. The DMA writes to Cache Line B. This means that it is not necessarily the same exact address, but within the same 64B cache line.
   - As a result, a snoop-write request is generated but not immediately issued as it is blocked by the snoop-write issued in the previous Step 1.
   - Once the snoop-write from Step 1 is complete, this snoop-write is processed.
3. The CPU reads from any address in external, cacheable memory that is a set match to Cache Line B. This must also create an L2$ hit (referred to here as Cache Line D).
   - This results in a cache miss from the CPU and sends a read request to L2 cache for the line.
   - Assuming this was also mapped to the same way as Cache Line B, this results in a replacement of Cache Line B since it was clean in L1D$.
   - Note that there is no method to determine what particular way is used, so it is not possible to tell whether this replacement would actually happen for a particular operation. This is why only a set match is mentioned here.
4. In parallel (the same CPU cycle) with Step 3, the CPU reads from any address in L2 SRAM that is a set match to Cache Line A, mentioned in prerequisite Step 2 (referred to here as Cache Line C).
   - This results in a cache miss from the CPU and sends a read request to L2 SRAM for the line.
   - Assuming this was also mapped to the same way as Cache Line A, this results in a replacement of Cache Line A if it was clean in L1D$. If Cache Line A was dirty, an eviction would occur before the allocation completed.
   - Note that there is no method to determine what particular way is used, so it is not

possible to tell whether this replacement would actually happen for a particular operation. This is why only a set match is mentioned here.

The results of the above cause the following:

(A) The snoop-write to Cache Line E, from Step 1 above, is now in process and blocking the snoop-write to Cache Line B from Step 2.

(B) While Step A is going on, Cache Line A has either now been evicted and/or replaced by Cache Line C from Step 4 above and Cache Line B (the intended target of the delayed snoop-write) is now replaced with Cache Line D from Step 3 above.

(C) Once the first snoop-write from operation C1 completes, the second (delayed) snoop-write mentioned in Step A to Cache Line B should be killed since Cache Line B was replaced in the operation in Step B. Instead, it is not killed and the line cached (which is now actually Cache Line D) is now updated incorrectly.

As a result, the following is true:

1. Cache Line D now holds data that was corrupted by the operation in Step C above (as a result of Step 2 above).
   - A subsequent read of this data returns a corrupted value.
   - Subsequent writes to this cache line also cause the corrupted values to be committed to lower levels of memory.

Figure 7 shows the sequence of events.

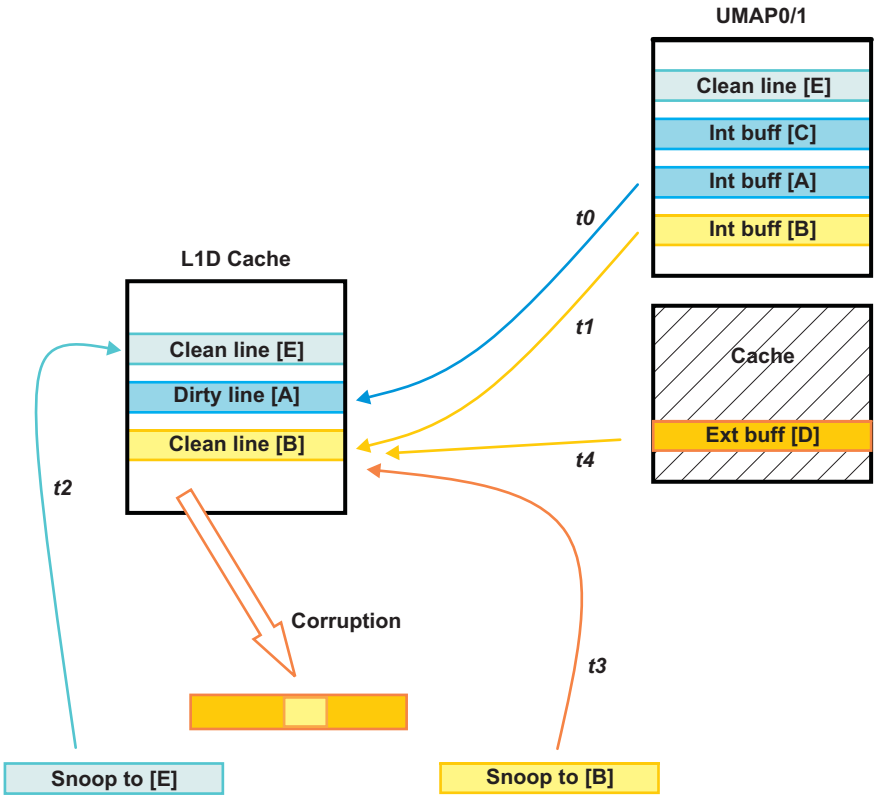


Figure 7. Sequence of Events

Workaround(s): A compiler flag (--c64p_dma_l1d_workaround) has been added to the latest Code Generation Tools to resolve this potential issue. This flag can be utilized for all code in the system or used on particular files/functions that may be susceptible to the conditions listed in this advisory.

Advisory 2.1.9 *Error Detection and Correction Incorrectly Reporting Error*

Revision(s) Affected: 2.1 and earlier

Details:

The C64x+ Megamodule L2 Memory Controller provides support for error detection and correction (EDC). The primary purpose of this is to protect code and largely static data held in L2 memory. Because the likelihood of a bit error on a given bit is proportional to the time since it was last written, and program images are rarely written, the focus of EDC is on those portions of L2 that are written rarely but must be correct when read.

The EDC implements a distance-3 "detect 2, correct 1" Hamming code. The L2 controller always performs a full Hamming code check on 256-bit reads, regardless of whether the fetch is from L1D controller, L1P controller, IDMA, or DMA. There is a parity value associated with every 256 bits (32B) of L2 memory and a valid bit to qualify each parity value. EDC uses parity RAM to store this parity information. Parity is calculated and made valid in the parity RAM for following operations:

- 256 bits IDMA write
- 256 bits DMA writes through SDMA
- L2 cache allocate (both read and write allocate, except for the line to which the write allocate writes).

Parity is made invalid in the parity RAM for the following operations:

- DMA writes through SDMA ***or*** IDMA writes for less than 256 bits.
- All L1D writes to L2, either cache or SRAM.
- L1D writes that cause an L2 write allocate on the line that gets written (part of the L2 cache line).
- All L1D victims.

EDC configuration registers are available to enable EDC individually for each of the L2 memory pages. Status registers are also available to report the address that shows the EDC error as well as the type of the error, whether it is 1-bit error or multiple-bit error. It also indicates whether it is corrected or not.

Problem Symptoms:

EDC is reporting EDC error (parity error) even when there is no error present in L2 memory. The error is random and the status register reports either 1-bit or multiple-bit error. It is also not consistent that after some defined iterations EDC reports an error. The EDC error can occur at any time and at any location in the memory. The error is a false positive; i.e., there is actually no error present in the memory, but EDC reports an error. There are two dedicated events (event 116, corrected bit error, and event 117, uncorrected bit error) going from EDC to the megamodule INTC. If interrupt is enabled and configured for those events, then the CPU reports an EDC interrupt.

Problem Prerequisites:

The following two operations must happen in parallel for this error to occur:

- L2 block coherence operation (WB and WBInv Only)
- L1D victim generation.

When there is an L2 block coherence operation going on (it could be either L2_WB or L2_WBInv) and before that operation is complete, if the CPU does the operations that generates the L1D victims, then it is possible that the L1D victim operation will mark the parity valid bit to be 1, which is incorrect behavior. This can easily occur when there are interrupts happening during the L2 Block WriteBack (L2_WB) or L2 WriteBackInvalidate (L2_WBInv) operation. The error does not occur during block invalidate operation. As mentioned above, it is a random occurrence that the L1D victim could validate the parity and generate the EDC interrupt.

Correct Behavior:

- L2 coherence operation in progress

  and
- L1D victim generated
- L1D victims are not EDC protected and, so, the parity valid bit should get reset to 0 and junk should be written to parity RAM.

Incorrect Behavior:

- L2 coherence operation in progress

  and
- L1D victim generated
- L1D victims are not EDC protected but the parity valid bit is marked valid with no parity calculated and junk written to parity RAM.
- Any subsequent reads to this cache line cause the L2 EDC error. EDC protection is performed as per junk parity data on that cache sub-line (256 bits) and it can corrupt the data in that cache sub-line.

Workaround(s):

Workaround 1:

Disable interrupts during L2 block coherence operations. If there are large block coherence operations and disabling the interrupt during those coherence operations is not feasible, then divide the big coherence operation into multiple, small coherence operations and protect each of them against allowing interrupts during two coherence operations.

Workaround 2:

Allow interrupts, but put the L1D cache in freeze mode before starting L2 block coherence operation so that L1D victims are not generated during the L2 block coherence operation. Un-freeze the L1D cache as soon as the L2 block coherence operation is complete.

Advisory 2.1.10 *SRIO May Fail to Send Interrupt for Completed TX or RX Message*

Revision(s) Affected: 2.1 and earlier

Details: The interrupt clearing/setting mechanism for the RXU/TXU gives priority to clearing the interrupt rather than setting it. The sequence of the peripheral for handling buffer descriptors of a completed message is to: write the buffer descriptor info, set the ICSR interrupt bit, and, finally, write the completion pointer (CP). As software processes the buffer descriptors during an ISR, it ends the process by writing the CP register to indicate to the peripheral what was the last buffer descriptor processed. This clears the interrupt, if both peripheral and software are at the same point; i.e., the interrupt is not cleared and will fire again once the pacing register has completed its countdown.

Due to the implementation of the interrupt clearing/setting, where priority is given to clearing the interrupt, if software writes the CP (which the peripheral compares to it's CP and matches) causing the interrupt to be cleared on the same internal clock cycle as the peripheral trying to set the interrupt bit for the next buffer descriptor, the interrupt bit is cleared and the interrupt for that next packet is lost. Note that no data is actually lost, the interrupt simply does not occur. Once an additional message is processed and the descriptor is completed, the interrupt is fired as normal and all descriptors can be processed at that point. Although not guaranteed, it is possible for this missed interrupt condition to occur with every ISR that attempts to write the TX or RX CP. However, since the missed interrupt descriptor can be processed during the next interrupt ISR, the only concern is added latency. For systems with a steady flow of messages, this added latency is usually insignificant, but it is evident on scenarios where it occurs on the last buffer descriptor in a group of messages since nothing is behind it to cause another interrupt. For example, if the RX queue received 10 messages and the tenth interrupt is lost, and no other messages were ever routed to that same RX queue, it will never fire another interrupt.

Workaround: Change the ISR as shown in the following steps and in Figure 8. Every time an interrupt is received:

1. Determine that the interrupt is related to CPPI. If not, call another handler.
2. Fetch the next descriptor (software maintains a current pointer, SW_Pointer).
3. Check the ownership bit for this next descriptor:
   (a) If it is not owned by software, go to Step 6.
   (b) If it is owned by software, then check the “CC” code and perform the remaining packet processing.
   (c) If EOQ is reached, write the completion pointer and go to Step 8.
   (d) Otherwise, continue with Step 4.
4. Move the SW_Pointer to point to this next descriptor.
5. Go back to Step 3.
6. Write the completion pointer based on the current SW_Pointer value.
7. Check the ownership bit for the next descriptor again:
   (a) If it is owned by software, go to Step 3b.
   (b) Otherwise, continue with Step 8.
8. Write the interrupt pacing register to enable the next interrupt.

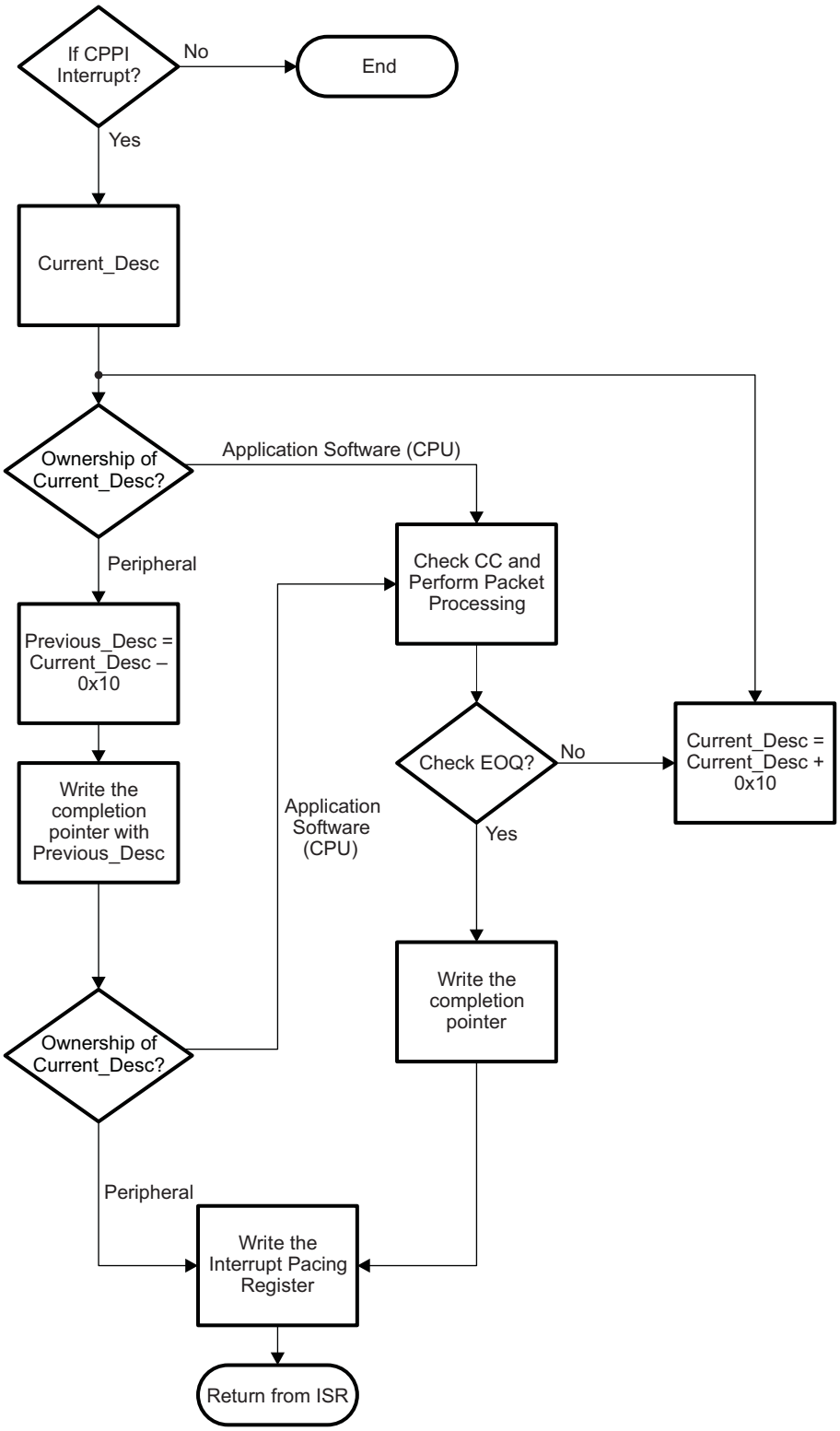


Figure 8. ISR Workaround Flowchart

Advisory 2.1.11 *Serial RapidIO Internal Digital Loopback is Not Always Stable*

Revision(s) Affected: 2.1, 2.0, 1.3, 1.2, 1.1, 1.0

Details: A digital loopback control function provides testability features with the ability to loop a port's transmit data back to the receive side. Digital loopback is controlled through bit 25 of the RIO_PER_SET_CNTL register. This single bit control affects every 1X port, or all lanes of a 4X port, depending on the supported mode of the device. This loopback is done in the digital logic domain and is before the SerDes. An issue was discovered where ports that are in digital loopback exhibit sporadic errors and are unreliable. In these instances, the ports are unable to maintain Port_ok status and may encounter multiple various error stopped states.

In digital loopback, the normal physical layer RX FIFO is bypassed altogether for data. The data is actually handed from TX to RX via a separate path. This handoff is being performed correctly, however, the RX FIFO sideband signals that indicate under/over run conditions are erroneously being evaluated by the digital logic, instead of being ignored. This means that the RX state machine continues acting upon the under/over run signals that can be affected by external signals or even noise coming in on the device pins. For example, if the SerDes device pins are connected to a link partner's active transmitter, the port is not able to remain initialized in loopback since the under/over run signals are following the link traffic. Unreliable digital loopback has also been observed without an active transmitting device attached.

Workaround: Avoid using the digital loopback mode. TX-to-RX loopback is also supported within the SerDes macros themselves. This internal SerDes loopback mode incorporates the complete RapidIO data path (including the RX FIFO) and eliminates the above mentioned issue. SerDes loopback is very stable and can be enabled with the following bits in the RapidIO SerDes registers:

RIO_SERDES_CFG1_CNTL[7:6] = 0b10
RIO_SERDES_CFGRXn_CNTL[1] = 0b1
RIO_SERDES_CFGTXn_CNTL[1] = 0b1

Note that loopback needs to be individually enabled for each port, or each lane of a 4X port, by setting bit 1 of the appropriate RIO_SERDES_CFGRXn_CNTL and RIO_SERDES_CFGTXn_CNTL register.

3 Silicon Revision 2.0 Usage Notes and Known Design Exceptions to Functional Specifications

3.1 Silicon Revision 2.0 Usage Notes

Silicon revision 2.0 applicable usage notes have been found on later silicon revisions; for more detail, see Section 2.1.

3.2 Silicon Revision 2.0 Known Design Exceptions to Functional Specifications

Table 12 lists the silicon revision 2.0 known design exceptions to functional specifications. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. Therefore, advisory numbering may not be sequential.

All other known design exceptions to functional specifications for silicon revision 2.0 still apply and have been moved up; see Table 3 and Section 2.2.

Table 12. Silicon Revision 2.0 Advisory List

Advisory 2.0.6 *DMA Corruption of L2 RAM Data*

Revision(s) Affected: 2.0; Fixed in revision 2.1

Details: Under a specific set of circumstances, a snoop-write updates an unintended L2 RAM location. This is a result of a corrupted L1D cache writeback, and can lead directly to program misbehavior. If that line is then modified by CPU accesses, a subsequent victim writeback from L1D could commit this corrupted line to lower levels of memory. Three key requirements for this issue are:

- The DMA reads or writes to buffers in L2 SRAM.
  - This must be cached and modified in L1D (read and written by the CPU).
- The CPU reads from any L2 or external, cacheable address.
- A second DMA write to the same cache line address (within 64B) in L2 RAM that the CPU is reading from.

NOTE:

1. For information on L1D cache coherence protocol, see section 3.3.6, *Cache Coherence Protocol*, in the *C64x+ DSP Megamodule Reference Guide* (SPRU871).
2. The DMA in the following description refers to all non-CPU requestors. This includes IDMA, EDMA, and any other master in the system.

Under the specific set of circumstances listed below, a snoop-write results in a data corruption in L2 RAM. This issue exists only when L1D evicts a dirty line from its cache while allocating a new line to the same set/way. Both lines must be from L2 SRAM in either UMAP0 or UMAP1. The issue occurs when there is a DMA to L2 for the allocated (clean) line and a DMA to or from the victim (dirty) line. The L2 sends the DMA request as a snoop-read or -write to the L1D cache after it allocates the new line. When the issue occurs, the snoop-write to the allocated line corrupts the line being evicted instead. The L2 writes this corrupted victim back to L2 SRAM.

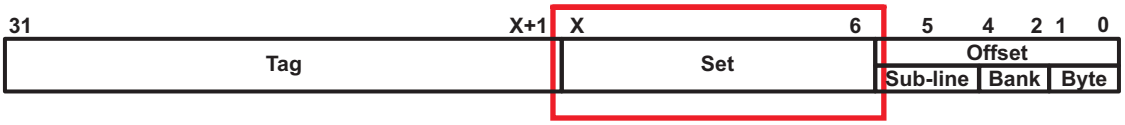
The prerequisite before the window where the issue occurs is:

- The CPU reads an L2 location and has modified (written to) the same cache line location before the window where the issue occurs. That means that it is not necessarily the same exact address that is written to, but within the same 64B cache line.
  - Because of this, a 64B cache line is allocated and dirty in L1D (referred to here as Cache Line A).

The following steps must all occur concurrently to see the issue:

1. The CPU reads from any address in L2 SRAM that is a set match to Cache Line A. (To determine if you have a set match, see below.)
   - The set match to Cache Line A is referred to here as Cache Line B.
   - This results in a cache miss from the CPU and sends a read request to L2 cache for the line (and possibly an external source if it was through L2 cache or if no L2 cache is present).
   - Since Cache Line A is dirty, a victim is prepared to be sent after Cache Line B is allocated and is held in a temporary victim data buffer.

   Determining if two addresses are a set match can be done by comparing certain bits of two addresses. The mapping of an address to a location in L1D cache is shown in Figure 9.



The value X is determined by how large the L1D cache is in the particular application (see Table 13).

Figure 9. L1D Cache Address Mapping

Table 13. Value of X for L1D Cache

AMOUNT OF L1D CACHE	X BIT POSITION
0KB	N/A
4KB	10
8KB	11
16KB	12
32KB	13

If you use the default configuration, 32KB, as an example, bits [13:6] are a set match if they are identical in two different addresses. Some examples of set matches are shown below:

- 0x0080 2A80 00000000100000000010101010000000
- 0x8000 2A80 10000000100000000010101010000000
- 0x0080 2A8A 00000000100000000010101010001010

2. The DMA read or writes from/to Cache Line A, mentioned in the prerequisite above. This means that it is not necessarily the same exact address, but within the same 64B cache line.
   - As a result, a snoop-read/-write request is generated.
3. The DMA writes to Cache Line B, mentioned in Step 1. This means that it is not necessarily the same exact address, but within the same 64B cache line as Step 1.
   - As a result, a snoop-write request is generated but not immediately issued, as it is blocked by the snoop-read/-write issued in Step 2.

The results of the above cause the following:

(A) The L1D controller receives the new line (B) back from the L2 Controller.

(B) If Step 2 above was a write, the snoop-write to Cache Line A updates the victim buffer correctly. If it was a read, the snoop-read returned the correct data to the DMA.

(C) The snoop-write to Cache Line B (Step 3 above) incorrectly updates the victim buffer instead of the newly allocated line that was returned in Step A.

As a result, the following is true:

1. Cache Line A now holds data that was corrupted by Steps 3 and C above.
   - A subsequent read of this data returns a corrupted value.
2. Cache Line B now holds stale data, as it was never updated with the data it was supposed to get from Steps 3 and C.
   - The CPU gets stale data (not updated).

Corruption only happens when the DMA accesses an L1D cache line that the CPU also writes to. This results in DMAs that may match victim lines leaving L1D. Thus, it can affect buffers that the CPU fills with writes and the DMA reads, as well as buffers where both the DMA and CPU write. It does not affect DMA buffers that the CPU only reads.

Figure 10 shows the sequence of events.

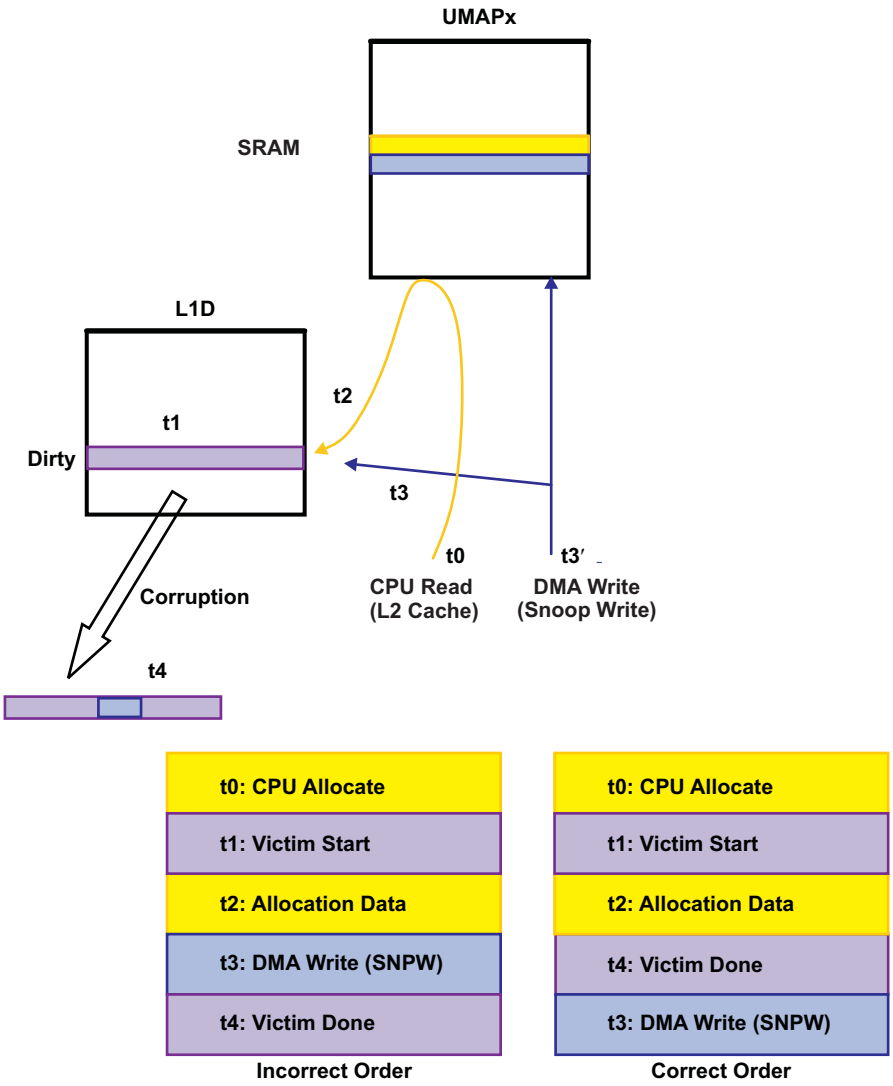


Figure 10. Sequence of Events

Table 14 shows the expected data values after this sequence completes and the actual values that are now present because of this issue.

Table 14. Expected vs. Actual Data Values[(1)]

	EXPECTED	ACTUAL
Buffer A	A‴	B″
Buffer B	B″	B

(1) Key:
A, B = Original data
A′ = CPU-written data
A″, B″ = DMA-written data
A‴ = CPU- and DMA-written data, properly merged

With all the steps above, it is fairly painful to determine if a particular buffer has the potential to see this issue. Figure 11 is a simple decision tree to help make a determination for a particular buffer.

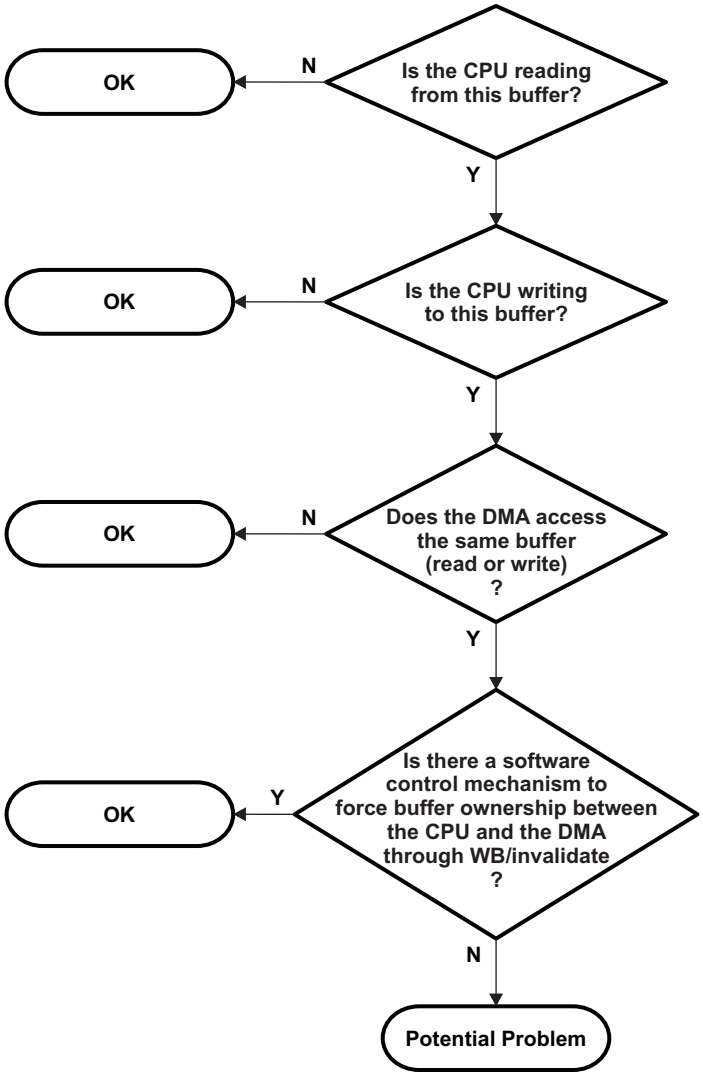


Figure 11. Decision Tree

If you approach one of the "OK" fields, then the buffer should not have a potential of being affected. If you arrive at "Potential Problem," see the workarounds below.

NOTE: Figure 11 assumes that each buffer is aligned to a 64B boundary and spans a multiple of 64B. This is because the cache line size of our L1D is 64B. If that is not the case, there is a chance that you might still see this issue even if you get to an OK state in the diagram (see the Workaround for "False-sharing" section below).

Workarounds: The issue occurs when the CPU writes within the same L1D cache line that the DMA reads or writes. This can happen for multiple reasons. The following sections detail workarounds for three scenarios:

1. The CPU writes to a buffer that the DMA then reads. This could either be due to an "in-place" algorithm that operates on data brought to it by DMA or an "out-of-place" algorithm where the CPU fills a buffer that the DMA then reads. In either case, the

CPU and DMA explicitly synchronize.

2. The CPU and DMA are updating distinct or unrelated objects that happen to share a cache line. (This is sometimes called "false sharing.") Because the objects are unrelated, the DMA and CPU are not synchronized.
3. The CPU and DMA are both writing to the same structure without external synchronization. This pattern often underlies software synchronization implementations and lockless multiprocessing algorithms.

Workaround for Synchronizing DMA and CPU Access to Buffers

The CPU potentially triggers this issue when it reads and later writes to a buffer that the DMA also accesses (read or write). The issue can happen when the DMA accesses the affected line when the L1D cache writes it back to L2. To avoid this issue, programmers can explicitly manage coherence on the buffer so that the buffer is not present and dirty in L1D when the DMA accesses it.

To explicitly manage coherence on the buffer, programmers should adhere to the programming model described earlier: Programs should write back or discard in-bound DMA buffers immediately after use and keep a strict policy of buffer ownership such that a given buffer is owned only by the CPU or the DMA at any given time.

This model assumes the following:

1. The DMA fills the buffer during a period when the CPU does not access it.
2. The DMA engine or other mechanism signals to the CPU that it has finished filling the buffer.
3. The CPU operates on the buffer, reading and writing to it, as necessary. The DMA does not access the buffer at this time.
4. The CPU relinquishes control of the buffer so that DMA may refill it. (This may be an implicit step in many implementations if the period between refills is much longer than the time it takes the CPU to process the refilled buffer.)

To implement this workaround, programmers must write back (and optionally invalidate) the buffer from L1D cache after Step 3 and before Step 4. There are multiple mechanisms for doing this, but the most straightforward is to use the L1D block cache writeback mechanism via L1DWBAR/L1DWWC or the L1D block cache writeback-invalidate mechanism via L1DWIBAR/L1DWIWC.

The recommended implementation of this workaround requires calling the l1d_block_wb.asm and l1d_block_wbinv.asm functions (see the L1D Block Writeback and L1D Writeback-Invalidate Routines below). The functions can be invoked as follows:

```
void l1d_block_wb(void *base, size_t byte_count);
```

or

```
void l1d_block_wbinv(void *base, size_t byte_count);
```

To writeback a C array, one could then do:

```
char array[SIZE];

/* ... */

l1d_block_wb(&array[0], sizeof(array));
```

The above example could be used to writeback-invalidate as well by calling the other function.

Programmers should insert such a call whenever the CPU code is done with a particular DMA buffer, before the DMA controller can refill it. The l1d_block_wb() and l1d_block_wbinv() functions are non-interruptible. The overhead is proportional to the size of the buffer.

NOTE: To ensure complete effectiveness, ensure that the DMA buffers always start on an L1D cache-line boundary (64-byte boundary) and occupy a multiple of 64 bytes. This may require increasing the size of some DMA buffers slightly. This is necessary to prevent accesses to an unrelated buffer or variable from bringing a portion of the DMA buffer back into the L1D cache.

L1D Block Writeback Routine

```
;; ========================================================================= ;;
;; L1D Block Writeback                                                       ;;
;;                                                                           ;;
;;   l1d_block_wb(void *base, size_t byte_count);                            ;;
;;                                                                           ;;
;; Performs a block writeback from L1D to L2. It can be used                 ;;
;; on any address range (L2 or external), but it only operates on L1D        ;;
;; cache.                                                                    ;;
;;                                                                           ;;
;; Maximum block size is 256K. Exact maximum byte count depends on the       ;;
;; alignment of the block.                                                   ;;
;;                                                                           ;;
;; Interrupts are disabled during the block writeback operation.             ;;
;; ========================================================================= ;;

     .asg 0x01844040, L1DW ; L1D Block Wb; BAR at 0, WC at 1
     .global _l1d_block_wb
     .text
     .asmfunc
_l1d_block_wb:

     MVC DNUM, B0        ; \_ Get global alias prefix
     ADDK 0x10, B0       ; /
     SHRU A4, 24, B2     ; Get prefix from address
     CMPEQ B0, B2, B0    ; Check if address prefix is global
 [ B0] EXTU A4, 8, 8, A4 ; Remove global prefix from address
     MVKL L1DW, B6 ;

     CLR A4, 0, 5, A1    ; Align to L1D cache line boundary
||   ADD A4, B4, B1      ; Compute end of buffer

     ADDK 63, B1         ; \_ Round to next L1D cache line
     CLR B1, 0, 5, B1    ; /

     SUB B1, A1, B1      ; Count cache-line span in bytes
||   MVKH L1DW, B6 ;

     SHR B1, 2, B1       ; Convert to "word count"
||   DINT ; Disable interrupts

     STW A1, *B6[0]      ; Store base address
     STW B1, *B6[1]      ; Store word count

     ; Note: The following loop is intentionally low-rate to avoid
     ; interfering with the block writeback operation.
loop: LDW *B6[1], B1     ; Read remaining word-count
     NOP 4
 [ B1] BNOP loop, 5      ; Loop until done

     RINT                ; Reenable interrupts
     RETNOP B3, 5        ; Return to caller

     .endasmfunc

;; ========================================================================= ;;
;; End of file: l1d_block_wb.asm                                             ;;
;; ========================================================================= ;;
```

L1D Block Writeback-Invalidate Routine

```
;; ======================================================================== ;;
;; L1D Block Writeback-Invalidate                                           ;;
;;                                                                          ;;
;;   l1d_block_wbinv(void *base, size_t byte_count);                        ;;
;;                                                                          ;;
;; Performs a block writeback-invalidate from L1D to L2. It can be used     ;;
;; on any address range (L2 or external), but it only operates on L1D       ;;
;; cache.                                                                   ;;
;;                                                                          ;;
;; Maximum block size is 256K. Exact maximum byte count depends on the      ;;
;; alignment of the block.                                                  ;;
;;                                                                          ;;
;; Interrupts are disabled during the block writeback operation.            ;;
;; ======================================================================== ;;
     .asg 0x01844030, L1DWI          ; L1D Block Wb-Inv; BAR at 0, WC at 1
     .global _l1d_block_wbinv
     .text
     .asmfunc
_l1d_block_wbinv:

     MVC DNUM, B0                    ; \_ Get global alias prefix
     ADDK 0x10, B0                   ; /
     SHRU A4, 24, B2                 ; Get prefix from address
     CMPEQ B0, B2, B0                ; Check if address prefix is global
 [ B0] EXTU A4, 8, 8, A4             ; Remove global prefix from address
     MVKL L1DWI, B6 ;

     CLR A4, 0, 5, A1                ; Align to L1D cache line boundary
||   ADD A4, B4, B1                  ; Compute end of buffer

     ADDK 63, B1                     ; \_ Round to next L1D cache line
     CLR B1, 0, 5, B1                ; /

     SUB B1, A1, B1                  ; Count cache-line span in bytes
||   MVKH L1DWI, B6 ;

     SHR B1, 2, B1                   ; Convert to "word count"
||   DINT                            ; Disable interrupts

     STW A1, *B6[0]                  ; Store base address
     STW B1, *B6[1]                  ; Store word count

     ; Note: The following loop is intentionally low-rate to avoid
     ; interfering with the block writeback operation.
loop: LDW *B6[1], B1                 ; Read remaining word-count
     NOP 4
 [ B1] BNOP loop, 5                  ; Loop until done

     RINT                            ; Reenable interrupts
     RETNOP B3, 5                    ; Return to caller

     .endasmfunc

;; ======================================================================== ;;
;; End of file: l1d_block_wbinv.asm                                         ;;
;; ======================================================================== ;;
```

Workaround for "False Sharing"

This issue can occur when the CPU and the DMA both access distinct objects that share a single L1D cache line. This is often referred to as "false sharing."

To avoid false sharing, ensure that the DMA buffers begin on 64-byte boundaries and occupy a multiple of 64 bytes. This may require increasing the size of some DMA buffers. If an application has many small DMA buffers, consider packing these together to limit the overall growth in DMA buffer space implied by this workaround.

Workaround for Buffers that the CPU and DMA Access Asynchronously

While this situation is rare in most programs, there are some cases where both the CPU and the DMA both access the same structure without explicit synchronization. In some

cases, this is due to the fact that said accesses are part of an algorithm that implements a synchronization primitive. Regardless of the purpose, these accesses potentially trigger this issue.

The easiest way to avoid the issue with this case is to freeze the L1D whenever the CPU reads this buffer. This prevents the buffer from allocating in the L1D cache so that the DMA never sends a snoop (read or write) to the DMC on behalf of this buffer.

Alternately, programs can always invalidate the line in L1D after reading it so that all writes to the line miss L1D and the line is never present and dirty in L1D cache. Programs can use the L1D block invalidate (L1DIBAR/L1DIWC) or L1D block writeback-invalidate (L1DWIBAR/L1DWIWC) to perform these explicit coherence operations.

Advisory 2.0.7 *L2 Victim Traffic Due To L2 Block Writeback During Any Pending CPU Request*

Revision(s) Affected: 2.0, 1.3, 1.2, 1.1, 1.0

Details: This advisory is an update to Advisory 1.3.1 in this document. Advisory 1.3.1 lists the following four blocking conditions to trigger an SDMA/IDMA stall:

1. Bursts of writes to non-cacheable locations.
2. L1D read miss generating victim traffic to L2 (cache or SRAM) or external memory.
3. L1D read request missing L2 (going external) while another L1D request is pending.
4. L2 victim traffic to external memory during any pending L1D request.

NOTE: Items 1, 2, 3, and 4 shown in the list above and in Table 15 below are actually labeled as 1, 2a, 2b, and 2c in Advisory 1.3.1.

This advisory covers one more blocking condition:

5. L2 victim traffic due to L2 block writeback during any pending CPU request.

For silicon revisions 1.0, 1.1, 1.2, and 1.3 that contain the original SDMA/IDMA blocking errata, this is a fifth way to encounter the issue in addition to the previously communicated four errata conditions in Advisory 1.3.1.

No additional deadlock risk potential is created by the addition of the new condition to silicon revisions 1.0, 1.1, 1.2, and 1.3 that currently contain the SDMA/IDMA blocking conditions 1-4. This means that this issue can lead to a deadlock in the same manner that the other four conditions can. On silicon revision 2.0, without the original stall conditions 1-4, this creates a deadlock condition that is identical to the previous revisions.

Table 15. Stall Conditions on Silicon Revisions

SILICON REVISIONS	STALL CONDITIONS				
	1	2	3	4	5
1.0	YES	YES	YES	YES	YES
1.1	YES	YES	YES	YES	YES
1.2	YES	YES	YES	YES	YES
1.3	YES	YES	YES	YES	YES
2.0	NO	NO	NO	NO	YES

Under certain conditions, L2 victim traffic due to a block writeback can block SDMA/IDMA accesses to UMAP0 during CPU requests. Table 16 shows the UMAP0 and UMAP1 address ranges for symmetric and asymmetric modes of the device.

Table 16. UMAP0 and UMAP1 Address Ranges

Devices	UMAP0	ADDRESS RANGE	UMAP1	ADDRESS RANGE
TCI6487/88 (Symmetric)	RAM	0x00800000 - 0x008FFFFF	N/A	N/A
TCI6487/88 (Asymmetric)				
Core 0	RAM	0x00900000 - 0x0097FFFF	RAM	0x00800000 - 0x008FFFFF
Core 1	RAM	0x00800000 - 0x008FFFFF	N/A	N/A
Core 2	RAM	0x00800000 - 0x0087FFFF	N/A	N/A

There are four transactions that must happen to cause an SDMA/IDMA to stall because of this condition:

1. L1D/L1P needs to create an L2$ hit. This happens as a result of one of the following:
   - An L1D victim (through L1D writeback or writeback-invalidate).
   - An L1D read+victim (through L1D read miss resulting in a writeback).
   - An L1D write miss (write-through to an uncached line).
   - An L1D read miss.
   - An L1P fetch miss.
2. A user-initiated L2 block writeback must occur involving the same cache set as the L1D/L1P cache accesses in the previous bullet.
3. An SDMA access to UMAP0.
4. The CPU also accesses the same cache set as the L1D/L1P cache accesses and the L2 block writeback as described in the first two bullets. This happens as a result of a CPU LDx/STx instruction that causes one of the following:
   - An L1D victim (through L1D writeback or writeback-invalidate).
   - An L1D write miss (write-through to an uncached line).
   - An L1D read miss.
   - An L1P fetch miss.

As a result of the four items above, any further SDMAs to UMAP0 are blocked. SDMAs to UMAP1 are unaffected. Note that three of these items ***must*** involve the same L2$ set in order to see the issue and, thus, is not as likely as the other conditions listed in the original errata. The stall persists until the operations above are complete.

Workarounds:

Workaround 1: Leave in previous SDMA/IDMA stall workarounds

For silicon revisions 1.0, 1.1, 1.2, and 1.3 that were already affected with the other four conditions of the SDMA/IDMA stall issue from Advisory 1.3.1, there is no additional workaround needed. If all of the deadlock avoidance steps listed in Advisory 1.3.1 have been followed, there is no risk for a deadlock because of this issue. Methods to reduce stalling due to this issue are also already covered in Advisory 1.3.1.

For silicon revision 2.0 that fixed the initial four conditions of SDMA/IDMA stall issue, the deadlock avoidance steps that are already listed in Advisory 1.3.1 for previous revisions of silicon should be followed to ensure that there is no chance of a deadlock. The workarounds to avoid stalls are also the same as communicated in previous revisions of the device with the issue.

Workaround 2: Do not use L2$

Systems that do not use L2$ are not affected by this issue.

4 Silicon Revision 1.3 Usage Notes and Known Design Exceptions to Functional Specifications

4.1 Silicon Revision 1.3 Usage Notes

Usage Notes highlight and describe particular situations where the device's behavior may not match presumed or documented behavior. This may include behaviors that affect device performance or functional correctness. These notes will be incorporated into future documentation updates for the device (such as the device-specific data manual), and the behaviors they describe will not be altered in future silicon revisions.

Some silicon revision 1.3 applicable usage notes have been found on later silicon revisions; for more detail, see Section 2.1.

4.1.1 Bootloader: Multicore Boot Takes Core1 and Core 2 Out of Reset

For silicon revisions 1.x on the TCI6487/8 device, the bootloader checks that the base of L2 for core1 and core2 is 0. If they are 0, the idle code (0x0001E000) is written to the base of the L2 (address 0x11800000 for core1 and 0x12800000 for core2). At the end of the bootloader before application code starts to run, the bootloader takes core1 and core2 out of reset.

If the first-level bootloader does not put any valid code into core1 and core2 L2 memory space and relies on the secondary level bootloader to load application code into core1 and core2 L2 RAM, the second-level bootloader in core0 needs to perform the following to make sure the core1 and core2 application software can run properly:

- Load the application software into core1 and core2 from the base address of the L2 RAM.
- At the end of the secondary bootloader in core0, apply Timer64_4 and Timer64_5 in watchdog mode. To ensure that both core1 and core2 start execution cleanly, use Timer64_4 to reset core1 and Timer64_5 to reset core2 before exiting the secondary bootloader. For timer watchdog configuration, see the *TMS320TCI6487/8 Communications Infrastructure Digital Signal Processor* (literature number SPRS358).

4.2 Silicon Revision 1.3 Known Design Exceptions to Functional Specifications

Table 17 lists the silicon revision 1.3 known design exceptions to functional specifications. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. Therefore, advisory numbering may not be sequential.

All other known design exceptions to functional specifications for silicon revision 1.3 still apply and have been moved up; see Table 3 and Section 2.2 or Section 3.2 .

Table 17. Silicon Revision 1.3 Advisory List

Advisory 1.3.1 *DSP SDMA/IDMA: Unexpected Stalling of SDMA/IDMA Access to L2 SRAM*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details:

NOTE: Only when DSP level 2 (L2) memory is configured as non-cache (RAM), unexpected stalling may occur on DSP SDMA/IDMA accesses. If DSP L2 memory is used only as cache ***or*** if L2 RAM is ***not*** accessed by IDMA or via the SDMA interface during run-time, then this exception does not apply.

The C64x+ megamodule has a Master Direct Memory Access (MDMA) bus interface and a Slave Direct Memory Access (SDMA) bus interface. The MDMA interface provides DSP access to resources outside the C64x+ megamodule (i.e., DDR2 memory). The MDMA interface is used for CPU/cache accesses to memory beyond the level 2 (L2) memory level. These accesses include cache line allocates, write-backs, and non-cacheable loads and stores to/from system memories. The SDMA interface allows other master peripherals in the system to access level 1 data (L1D), level 1 program (L1P), and L2 RAM DSP memories. The masters allowed accesses to these memories are DMA controllers, EMAC, RAC back-end interface (TCI6488 only), and SRIO. The DSP Internal Direct Memory Access (IDMA) is a C64x+ megamodule DMA engine used to move data between internal DSP memories (L1, L2) and/or the DSP peripheral configuration bus. The IDMA engine shares resources with the SDMA interface.

The C64x+ megamodule has an L1D cache and an L2 caches, both of which implement write-back data caches. The C64x+ megamodule holds updated values for external memory as long as possible. It writes these updated values, called *victims*, to external memory when it needs to make room for new data, when requested to do so by the application, or when a load is performed from a non-cacheable memory for which there is a set match in the cache (i.e., the non-cacheable line would replace a dirty line if cached). The L1D sends its victims to L2. The caching architecture has pipelining, meaning multiple requests could be pending between L1, L2, and MDMA. For more details on the C64x+ megamodule and its MDMA and SDMA ports, see the *TMS320C64x+ Megamodule Reference Guide* (literature number SPRU871).

Ideally, the MDMA (the blue lines in Figure 12) and SDMA/IDMA paths (the orange lines in Figure 12) operate independently with minimal interference. Normally, MDMA accesses may stall for extended periods of time (clock cycles) due to expected system level delays (e.g., bandwidth limitations, DDR2 memory refreshes). However, when using L2 as RAM, SDMA and/or IDMA accesses to L2/L1 may experience unexpected stalling in addition to the normal stalls seen by the MDMA interface. For latency-sensitive traffic, the SDMA stall can result in missing real-time deadlines.

NOTE: SDMA/IDMA accesses to L1P/D will not experience an unexpected stall if there are no SDMA/IDMA accesses to L2. Unexpected SDMA/IDMA stalls to L1 happen only when they are pipelined behind L2 accesses.

Figure 12 is a simplified view for illustrative purposes only. The IDMA/SDMA path (orange lines) can also go to L1D/L1P memories and IDMA can go to the DSP CFG peripherals. MDMA transactions (blue lines) can also originate from L1P or L1D through the L2 controller or directly from the DSP.

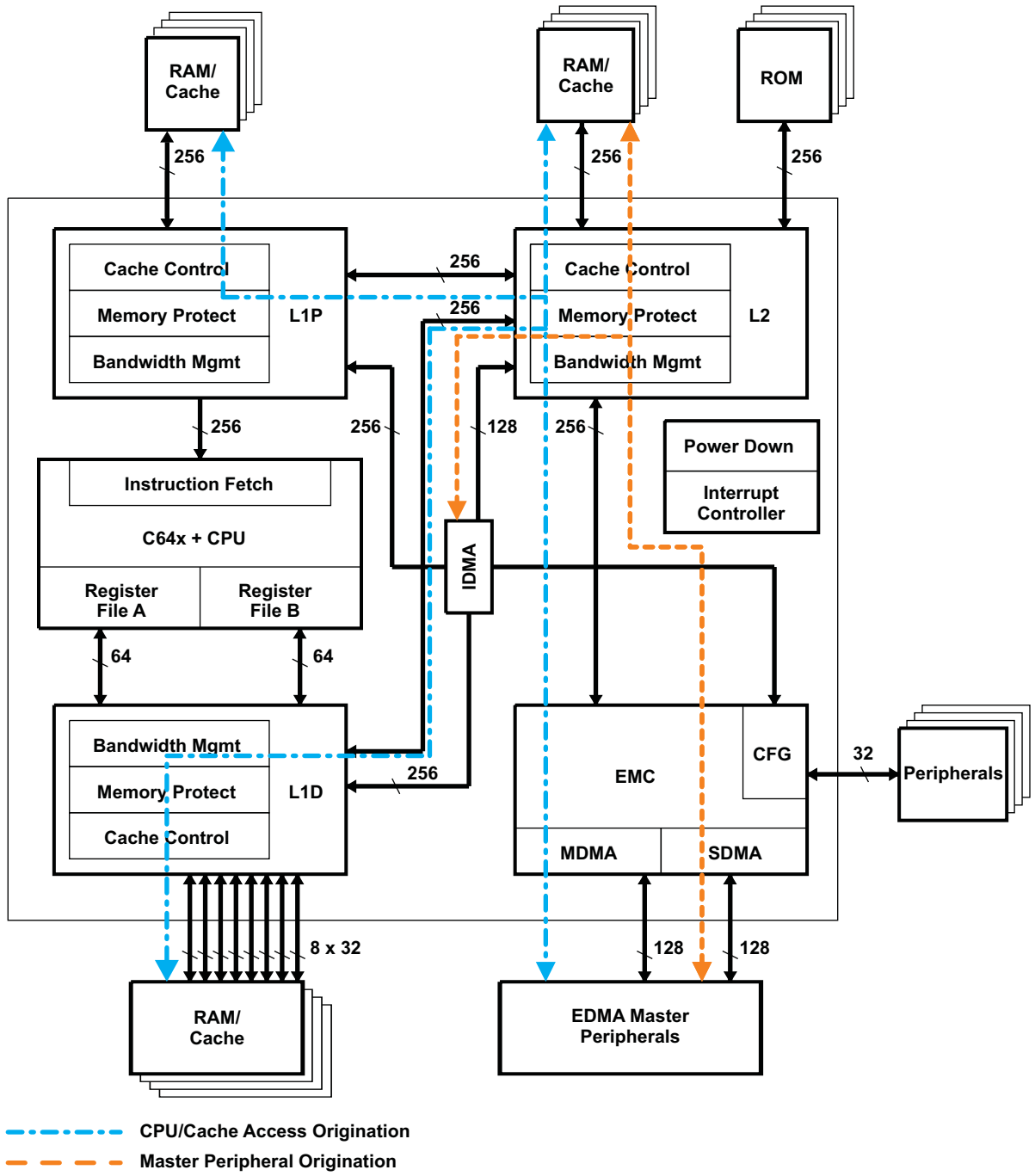


Figure 12. IDMA, SDMA, and MDMA Paths

SDMA/IDMA stalls may occur during the following scenarios. Each of these scenarios describes expected normal DSP functionality, but the SDMA/IDMA access potentially exhibits additional unexpected stalling.

1. Bursts of writes to non-cacheable MDMA space (i.e., DDR2). The DSP buffers up to 4 non-cacheable writes. When this buffer fills, SDMA/IDMA is blocked until the buffer is no longer full. Therefore, bursts of non-cacheable writes longer than three writes can stall SDMA/IDMA traffic.
2. Various combinations of L1 and L2 cache activity:
   (a) L1D read miss generating victim traffic to L2 (cache or SRAM) or external memory. The SDMA/MDMA may be stalled while servicing the read miss and the victim. If the read miss also misses L2 cache, the SDMA/IDMA may be stalled until data is fetched from external memory to service the read miss. If the read access is to non-cacheable memory there will still potentially be an L1D victim generated even though the read data will not replace the line in the L1D cache.
   (b) L1D read request missing L2 (going external) while another L1D request is pending. The SDMA/IDMA may be stalled until the external memory access is complete.
   (c) L2 victim traffic to external memory during any pending L1D request. The SDMA/IDMA may be stalled until external memory access and the pending L1D request are complete.

The duration of the SDMA/IDMA stalls depends on the quantity/characteristics of the L1/L2 cache and the MDMA traffic in the system. In cases 2a, 2b, and 2c, stalling may or may not occur depending on the state of the cache request pipelines and the traffic target locations. These stalling mechanisms may also interact in various ways, causing longer stalls. Therefore, it is difficult to predict if stalling will occur and for how long.

SDMA/IDMA stalling and any system impact is most likely in systems with excessive context switching, L1/L2 cache miss/victim traffic, and heavily loaded EMIF.

Use the following steps to determine if SDMA/IDMA stalling is the cause of real-time deadline misses for existing applications. Situations where real-time deadlines may be missed include loss of McBSP samples and low peripheral throughput.

1. Determine if the transfer missing the real-time deadline is accessing L2 or L1D memory. If not, then SDMA/IDMA stalling is not the source of the real-time deadline miss.
2. Identify all SDMA transfers to/from L2 memory (e.g., EDMA transfer to/from L2 from/to a McBSP or from/to AIF, TCP, or VCP). If there are no SDMA transfers going to L2, then SDMA/IDMA stalling is not the source of the problem.
3. Redirect all SDMA transfers to L2 memory to other memories using one of the following methods:
   - Temporarily transfer all the L2 SDMA transfers to L1D SRAM.
   - If not all L2 SDMA transfers can be moved to L1D memory, temporarily direct some of the transfers to DDR memory and keep the rest in L1D memory. There should be ***no*** L2 SDMA transfers.
   - If neither of the above approaches are possible, move the transfer with the real-time deadline to the EMAC CPPI RAM. If the EMAC CPPI RAM is not big enough, a two-step mechanism can be used to page a small working buffer defined in the EMAC CPPI RAM into a bigger buffer in L2 SRAM. The EDMA module can be setup to automate this double buffering scheme without CPU intervention for moving data from the EMAC CPPI RAM. Some throughput degradation is expected when the buffers are moved to the EMAC CPPI RAM.

     Note: Note that EMAC CPPI RAM memory is word-addressable only and, therefore, must be accessed using an EDMA index of 4 bytes.

If real-time deadlines are still missed after implementing any of the options in Step 3, then SDMA/IDMA stalling is likely ***not*** the cause of the problem. If real-time deadline misses are solved using any of the options in Step 3, then SDMA/IDMA stalling ***is*** likely the source of the problem.

An extreme manifestation of the IDMA/SDMA stall issue is the C64x+ MDMA-SDMA deadlock that requires a device reset or power-on reset in order for the system to recover. The following summarizes the deadlock conditions:

- Master(s) on a single main MSCR port write to the GEM's SDMA followed by a write to slaveX
- The GEM issues victim traffic or a non-cacheable write to slaveX
- Any one of the following:
  - A write data path pipelined in main MSCR between master(s) and the GEM's SDMA
  - A bridge exists between master(s) and the main MSCR
  - Master(s) are able to issue a command to slaveX concurrent with the write to the GEM's SDMA.

A load (either cacheable or non-cacheable) from another core's L1D or L2 memory can additionally create a deadlock condition. When the load is issued the read command is propagated to the SDMA port of the other core through a bridge that is shared with the EDMA TC1, EMAC, RapidIO (both data and CPPI), and other GEM MDMA. When the load is issued, if a victim is generated in L1D cache, then the SDMA may stall until the load completes. If other masters are issuing commands through the shared bridge, then the bridge may fill due to the stalled SDMA before the read command can propagate through the bridge and complete. In summary, a deadlock can occur if the following is true:

- GEMx issues a read to GEMy or GEMz L1D or L2 SRAM
- Any of the following are issuing commands to GEMx L2: TC1, EMAC (data or CPPI), RapidIO (data or CPPI), GEMy, or GEMz.

Workarounds:

Method 1

Issues such as dropped McBSP samples can be worked around by moving latency-sensitive buffers outside the C64x+ megamodule. For example, rather than placing buffers for the McBSP into L1/L2, those buffers can instead be placed in other memory, such as the EMAC CPPI RAM.

Note: Note that EMAC CPPI RAM memory is word-addressable only and, therefore, must be accessed using an EDMA index of 4 bytes.

Method 2

To reduce the SDMA/IDMA stalling system impact, perform any of the following:

1. Improve system tolerance on DMA side (SDMA/IDMA/MDMA):
   - Understand and minimize latency-critical SDMA/IDMA accesses to L2 or L1P/D.
   - Directly reduce critical real-time deadlines, if possible, at peripheral/IO level (e.g., increase word size and/or reduce bit rates on serial ports).
   - To reduce DSP MDMA latency:
     – Increase the priority of the DSP access to DDR2 such that MDMA latency of MDMA accesses causing stalls is minimized.
       Note: Other masters may have real-time deadlines that dictate higher priority than the DSP.
     – Lower the PRIO_RAISE field setting in the DDR2 memory controller's burst priority register. Values ranging between 0x10 and 0x20 should give decent performance and minimize latency; lower values may cause excessive SDRAM row thrashing.
2. Minimize offending scenarios on DSP/caching side:
   - If the DSP performing non-cacheable writes is causing the issue, insert protected non-cacheable reads (as shown in the last list item below) every few writes to allow the write buffer to empty.
   - Use explicit cache commands to trigger cache writebacks during appropriate times (L1D Writeback All, L2 Writeback All). ***Do not*** use these commands when real-time deadlines must be met.
   - Restructure program data and data flow to minimize the offending cache activity.
     – Define the read-only data as *const*. The const C keyword tells the compiler not to write to the array. By default, such arrays are allocated to the .const section as opposed to BSS. With a suitable linker command file, the developer can link the .const section off chip, while linking .bss on chip. Because programs initialize .bss at run time, this reduces the program's initialization time and total memory image.
     – Explicitly allocate lookup tables and writeable buffers to their own sections. The #pragma DATA_SECTION (label, *section*) directive tells the compiler to place a particular variable in the specified COFF section. The developer can explicitly control the layout of the program with this directive and an appropriate linker command file.
     – Avoid directly accessing data in slow memories (e.g., flash); copy at initialization time to faster memories.
   - Modify troublesome code.
     – Rewrite using DMAs to minimize data cache writebacks. If the code accesses a large quantity of data externally, consider using DMAs to bring in the data, using double buffering and related techniques. This will minimize cache write-back traffic and the likelihood of SDMA/IDMA stalling.
     – Re-block the loops. In some cases, restructuring loops can increase reuse in the cache and reduce the total traffic to external memory.
     – Throttle the loops. If restructuring the code is impractical, then it is reasonable to slow it down. This reduces the likelihood that consecutive SDMA/IDMA blocks stack up in the cache request pipelines, resulting in a long stall.
   -

Protect non-cacheable reads from generating an SDMA stall by freezing the L1D cache during the non-cacheable read access(es). The following example code contains a function that protects non-cacheable reads, avoids blocking during the reads, and, therefore, avoids the deadlock state.

```
;; ======================================================================== ;;
;;  Long Distance Load Word                                                 ;;
;;                                                                          ;;
;;      int long_dist_load_word(volatile int *addr)                         ;;
;;                                                                          ;;
;;  This function reads a single word from a remote location with the L1D   ;;
;;  cache frozen.  This prevents L1D from sending victims in response to    ;;
;;  these reads, thus preventing the L1D victim lock from engaging for the  ;;
;;  corresponding L1D set.                                                  ;;
;;                                                                          ;;
;;  The code below does the following:                                      ;;
;;                                                                          ;;
;;      1.  Disable interrupts                                              ;;
;;      2.  Freeze L1D                                                      ;;
;;      3.  Load the requested word                                         ;;
;;      4.  Unfreeze L1D                                                    ;;
;;      5.  Restore interrupts                                              ;;
;;                                                                          ;;
;;  Interrupts are disabled while the cache is frozen to prevent affecting  ;;
;;  the performance of interrupt handlers.  Disabling interrupts during     ;;
;;  the long distance load does not greatly impact interrupt latency,       ;;
;;  because the CPU already cannot service interrupts when it's stalled by  ;;
;;  the cache.  This function adds a small amount of overhead (~20 cycles)  ;;
;;  to that operation.                                                      ;;
;;                                                                          ;;
;; ======================================================================== ;;

        .asg    0x01840044,     L1DCC           ; L1D Cache Control
        .global _long_dist_load_word
        .text
        .asmfunc
; int long_dist_load_word(volatile int *addr)
_long_dist_load_word:
        MVKL    L1DCC,          B4
        MVKH    L1DCC,          B4
||      DINT                                    ; Disable interrupts
||      MVK     1,              B5
        STW     B5,             *B4             ; \_ Freeze cache
        LDW     *B4,            B5              ; /
        NOP     4
        SHR     B5,     16,     B5              ; POPER -> OPER
||      LDW     *A4,            A4              ; read value remotely
        NOP     4
        STW     B5,             *B4             ; \_ Restore cache
        RET     B3
||      LDW     *B4,            B5              ; /
        NOP     4
        RINT                                    ; Restore interrupts
        .endasmfunc

;; ======================================================================== ;;
;;  End of file: ldld.asm                                                   ;;
;; ======================================================================== ;;
```

In the TCI6487/8 multicore device, when one GEM is accessing another GEM's L1 or L2 memory it is an MDMA access, so the potential SDMA/IDMA stall can occur. The stall can be avoided by using the EDMA to transfer data from one GEM's memory to another.

Method 3

Entirely eliminate the exception by removing all SDMA/IDMA accesses to L2 SRAM. For example, EMAC descriptors and EMAC payload cannot reside in L2. Master peripherals like the EDMA/QDMA, IDMA, and SRIO cannot access L2. There are no issues with the CPU itself accessing code/data in L2. This issue only pertains to SDMA/IDMA accesses to L2.

Deadlock Avoidance

To avoid the manifestation of a C64x+ deadlock, several Workarounds: are suggested depending on the VBUSM master in question:

VBUSM MASTER	WORKAROUND
GEM	GEMs should not write to the memory of any other GEM. This will cause complications across any master peripheral that is transferring data to multiple L2s. GEMs must not directly read from the memory of any other GEMs without providing the L1D cache disable workaround mentioned in Method 2 to ensure that the load will not stall itself indefinitely and hang the system.
EDMA3TCx	Inbound and outbound traffic should be programmed on different TC ports (i.e., two different EDMA queues, since a given queue maps to a given TC). Note that in-/out-bound direction is defined as the write direction, meaning that a DDR2-to-DDR2 transfer is outbound and L2-to-L2 is inbound. Any TC used to write to DDR should not be used to write to a GEM even when the TC writing to the DDR is also reading from DDR.
EMAC	EMAC should write to the GEM's memory or the DDR, but not both. This includes buffers and buffer descriptors. EMAC CPPI descriptors should be placed wholly in the local wrapper memory, any combination of wrapper and L2 memory (must match other master transactions), or any combination of wrapper and DDR2 SDRAM (must match other master transactions). Buffer descriptors should not be placed in any combination of L2 and DDR2 SDRAM.
SRIO	SRIO should transfer payload data to only GEM memories or to DDR2 SDRAM, but not both. This includes any direct I/O writes as well as any inbound RX messaging transfer.
SRIO CPPI	SRIO CPPI descriptors should be placed wholly in the local wrapper memory, any combination of wrapper and L2 memory, or any combination of wrapper and DDR2 SDRAM. Buffer descriptors should not be placed in any combination of L2 and DDR2 SDRAM.
RAC Back End 0	RAC Back End 0 should transfer payload data to only the DDR or to the GEM, but not both (no dependency on RAC Back End 1).
RAC Back End 1	RAC Back End 1 should transfer payload data to only DDR or to GEM, but not both (no dependency on RAC Back End 0).

Advisory 1.3.2 *Potential Data Corruption on SCR Bridge*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details:

The issue manifests when a master writes to a bridge and the same master also writes to another slave endpoint. When the slave endpoint causes writes to stall, it may result in the separation of the command request (creq) and write data request (wreq) from the SCR to the bridge. This separation between command request and write data request results in bridge failure, causing write data corruption. Multiple masters writing to the bridge do not cause the problem unless one of the masters is also writing to another Slave endpoint, causing the write stall.

All VBUSM-to-VBUSP bridges configured for cut-through mode of operation (i.e., low latency to maximize performance) are susceptible to this issue as they are not tolerant to the delay between the command request and the write data request. The problem occurs when the SCR submits a write command request and does not submit the corresponding write data request within the timing window expected by the bridge. Hence, the bridge assumes that valid write data is available as soon as the previous write has completed since the write command was presented during the previous write transaction.

The bridge commits to a write burst on the VBUSP output as soon as the write data FIFO has one element. When there is a gap between the command request and the write data request on a second write command, the bridge sees a write FIFO data valid when it really is the last valid data for the previous write. This behavior causes a write data shift on the VBUSP side of the bridge.

The bridge cannot recover without a reset. There is no software indication of this nor a means to reset only the bridge. Therefore, the situation must be avoided. The affected bridges are: TCP (bridge 12), VCP (bridge 11), AIF write port (bridge 24), and DMA bridge-to-configuration bus (bridge 10, where key endpoints beyond the configuration bus could include the Semaphore configuration port, EMAC configuration ports, PaRAM configuration port, or SRIO configuration ports).

Workarounds:

There are two ways to avoid this issue:

1. Partition the transfers such that any master used for the bridge writes does not perform writes to any other slave endpoints. Reads from any of the slaves can be performed as the read stall does not cause this problem.
2. Ensure that bridge writes are not initiated until all previous slave writes of that master are completed. For example, with the EDMA this can be achieved by waiting for the normal mode EDMA transfer completion interrupt or chaining event before starting the next transfer. This guarantees that the bridge always gets the write data immediately after the write command.

This problem will be fixed in silicon revision 2.0.

Advisory 1.3.3 *Potential Insertion or Deletion of 2 Bits in SerDes Data Stream*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details:

For arbitrary phase mode, a FIFO function is integrated into the SerDes TX serializer. This FIFO has three states (minus1, center, plus1) and is supposed to be reset to the center state at startup. From this position, the SerDes is then tolerant to variations of phase between the input clock (TXBCLKIN) and the SerDes internal clock, caused by temperature and voltage variations. However, as a result of a logic issue, the possibility exists that under some circumstances, the FIFO may not start in the center state. When this happens, there is a risk that the FIFO may subsequently overflow or underflow.

Whether the FIFO fails to initialize to the center state depends on the timing relationships between several signals, including the SerDes internal clock. Even if the FIFO fails to initialize to the center state, the FIFO will only underflow or overflow if the phase relationship between the TXBCLKIN input and the internal SerDes clock vary (due to temperature or voltage changes) in such a way as to cause their edges to cross in one particular direction. Overflow results in two bits being added to the data stream. Underflow results in two bits being deleted. If overflow or underflow occurs at all, it only happens once per TX lane because after it has occurred the FIFO is configured exactly as if it had initialized to the center state at startup.

The precise silicon process of the device will also be a factor in whether the overflow or underflow occurs. Some devices may exhibit this behavior at some particular PVT combinations, others may never exhibit it. It is not possible to predict whether, or under what conditions, a device is susceptible. If overflow or underflow occur, it could be at any time ranging from immediately after startup to weeks, months, or years later.

Workarounds:

The issue can be worked around by software control of two ports on the SerDes. At initialization, cycling of bits resets the circuit and resolves the issue.

- AIF has a software workaround as follows:

  The software workaround limits restart to per macro, not per lane. There is one set of software control bits for the B8 and another for the B4. For details, see the device-specific data manual, *TMS320TCI6487/8 Communications Infrastructure Digital Signal Processor* (literature number SPRS358). There are new recommendations for the initialization sequence that is shown in the following code example:

```
//Enable the Tx Link
CSL_FINST(hAifLink[0]->regs->LCFG[1].LINK_CFG, AIF_LINK_CFG_TX_LINK_EN, ENABLED);

//Set the Link Rate
if (aCommoncfg[0].linkRate == CSL_AIF_LINK_RATE_1x){
      CSL_FINST(hAifLink[0]->regs->LCFG[1].LINK_CFG, AIF_LINK_CFG_LINK_RATE, 1X);
}
else if (aCommoncfg[0].linkRate == CSL_AIF_LINK_RATE_2x)
      CSL_FINST(hAifLink[0]->regs->LCFG[1].LINK_CFG, AIF_LINK_CFG_LINK_RATE, 2X);
}
else if (aCommoncfg[0].linkRate == CSL_AIF_LINK_RATE_4x)
      CSL_FINST(hAifLink[0]->regs->LCFG[1].LINK_CFG, AIF_LINK_CFG_LINK_RATE, 4X);
}

//Toggle the ENFTP bit
CSL_FINS( hAifLink[0]->regs->AI_SerDes0_TST_CFG, AIF_AI_SerDes0_TST_CFG_INVPATT,
1);
CSL_FINS(hAifLink[0]->regs->AI_SerDes0_TST_CFG, AIF_AI_SerDes0_TST_CFG_INVPATT,
0);
```

  CSL version 3.0.6.2 for the TCI6487/8 device has a new hardware control command (CSL_AIF_CMD_ENABLE_DISABLE_TX_LINK_SI1_1) that has the fix for this advisory.

- EMAC has a software workaround and an auto-recovery for this advisory as follows:
  There is a new recommendation for initialization sequence as shown in the following code example. This example code should be used with CSL version 03.00.06.01.

```
SgmiiCfg.masterEn   = 0x1;
SgmiiCfg.loopbackEn = 0x1;
SgmiiCfg.auxConfig  = 0x0000000b;

if (0 == SGMII_config(&SgmiiCfg))
        printf("SGMII config successful........\n");
else
        printf("SGMII config NOT successful........\n");

LocalTicks = 0;
while(LocalTicks !=3); // wait for 2us

SgmiiCfg.txConfig   = 0x00000e21; // enable transmitter
SgmiiCfg.rxConfig   = 0x00081021;

if (0 == SGMII_config(&SgmiiCfg))
        printf("SGMII config successful........\n");
else
        printf("SGMII config NOT successful........\n");

SgmiiCfg.txConfig   = 0x00010e21; // toggle the ENFTP bit

if (0 == SGMII_config(&SgmiiCfg))
        printf("SGMII config successful........\n");
else
        printf("SGMII config NOT successful........\n");

SgmiiCfg.masterEn   = 0x1;
SgmiiCfg.loopbackEn = 0x1;
SgmiiCfg.txConfig   = 0x00000e21; // toggle the ENFTP bit

if (0 == SGMII_config(&SgmiiCfg))
        printf("SGMII config successful........\n");
else
        printf("SGMII config NOT successful........\n");

// wait for the Auto-negotiation Complete
SGMII_REGS->CONTROL |= 0x1; // Loopback mode is selected

// set full dupex and Gig bits
SGMII_REGS->MR_ADV_ABILITY = 0x9801;
```

- SRIO has an auto-recovery as follows:
  Auto-recovery resets the link and re-exposes the issue. TI is working to understand the likelihood of repeated recovery and whether there could be performance impacts due to repeated recovery.

The software workaround is enabled with a partial fix for AIF and EMAC. A complete fix will be available in silicon revision 2.x.

Advisory 1.3.4 *MAC EOI Register Write Causes Potential CPU Lockup*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0; Fixed in CSL version 03.00.06.01

Details: An issue has been found affecting multiple cores trying to write the EOI register via the MAC interface. It causes a lockup of one of the three MAC interfaces that is attempting to write the EOI register.

When multiple cores try to access the MAC interface one of the three cores that requested the EOI write gets locked up. This situation occurs when the MAC interface receives the EOI register write requests from multiple cores like "X" write followed by "Y" write at same clock, the EOI register updates only one write at a time, X or Y, and ignores the other write. The EOI write request that was ignored by the MAC locks up the CPU that requested the write.

Workarounds: Semaphores can be used to fix the EOI issue. There are two new APIs added to write the EOI register, one for receive and the other for transmit. The application can make use of those APIs with the semaphore module, to protect the EOI write when all the 3 cores try to access EOI register at same time.

This workaround is required only when more than one core requests the EOI write. Code examples for receive and transmit writes are shown below.

- Before and after `rxEoiWrite`, the semaphore APIs are called:

```
/* Check Whether Handle opened successfully and then read module status*/
if(hSemHandle!= NULL){
      /* Check whether semaphore resource is Free or not */
      do{
      /* Get the semaphore*/
      CSL_semGetHwStatus(hSemHandle,CSL_SEM_QUERY_DIRECT,&response);
      }while(response.semFree != CSL_SEM_FREE);

/* write the EOI register */
      EMAC_rxEoiWrite(coreNum);

/* Release the semaphore*/
CSL_semHwControl(hSemHandle, CSL_SEM_CMD_FREE_DIRECT,NULL);
```

- Before and after `txEoiWrite` the semaphore APIs are called:

```
/* Check Whether Handle opened successfully and then read module status*/
if(hSemHandle!= NULL){
/* Check whether semaphore resource is Free or not*/
      do {
      /* Get the semaphore*/
CSL_semGetHwStatus(hSemHandle,CSL_SEM_QUERY_DIRECT,&response);
      } while (response.semFree != CSL_SEM_FREE);

/* write the EOI register */
      EMAC_txEoiWrite(coreNum);

/* Release the semaphore*/
CSL_semHwControl(hSemHandle, CSL_SEM_CMD_FREE_DIRECT,NULL);
```

This issue is fixed in the CSL version 03.00.06.01

Advisory 1.3.6 *I2C: Slave Boot Aborts*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details: I2C Slave Boot is intended to speed the boot process for a system with more than two devices by allowing a single master read of the I2C EEPROM followed by a broadcast by that master to all remaining devices on the I2C bus. However, during the I2C slave boot process an internal exception is encountered, causing the boot sequence to abort on the slave device(s). Consequently, I2C slave boot does not complete.

Workaround: Use I2C master boot for all devices in the system. Other boot modes with SRIO or EMAC may also be utilized, if available on system.

Advisory 1.3.7 *RAC: Potential Corruption of RAC Statistics*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details:

A potential corruption of the RAC statistics for the first output descriptor generated after the RACPM or RACPD tasks can occur when:

- statistics are enabled in RACPM or RACPD
- the application code has a non-RACPM/RACPD task written out by the RAC between the RACPM/RACPD statistic tasks
- statistic channel ID 0 is used

In these instances, the statistics for the first output descriptor generated after the RACPM/RACPD tasks are different than the ones that correspond to the associated RAC output results.

The problem is due to an incorrect initialization value in the hardware code of the RAC Back-End Interface. The statistic channel ID pipeline register is currently reset to 0. This register is the history ID number for the statistics. It should be reset to 0x1FF, a non-relevant ID value. When the PM or PD task sees that this pipeline register is 0, it assumes that there were some statistics already completed for ID 0; it takes into account the previous statistics, which is incorrect. This causes the corruption of the RAC statistics for the first output descriptor generated after RACPM or RACPD tasks.

The RAC output results written in the output block buffer are correct. Only the statistics performed on this data are not.

This issue only happens with statistic ID 0, but due to a constraint when defining a page start index in RACPM or RACPD functional libraries (it must be a multiple of 4), the statistic channel IDs 0-3 should not be used. This issue does not affect the remaining statistic channel IDs (4-255).

Workaround:

The statistics post-processing is performed on each output channel. This means that the mean value accumulator and the highest peak have intermediate values that are stored within the associated statistics table.

The statistics table contains the intermediate statistics values for a given output channel. It is addressed through the pointer contained in the OCT. The table has 256 entries.

Since the statistics table entry 0 could potentially be corrupted, the workaround is to avoid using it.

In current implementation of RACFL, each GCCP has an S resource container. An S resource container contains 128 S resource elements indexed from 0 to 127. The S resource elements 0-127 of the S container of GCCP0 correspond to statistics table entries 0-127. The S resource elements 0-127 of the S container of GCCP1 correspond to statistics table entries 128-255. Since statistics table entry 0 could be corrupted, it should not be used by the application. For RACFL, GCCP0 S container resource element ID 0 should not be used. Since the S resource has a step of 4, GCCP0 S resource of ID 0-3 should not be used. For GCCP1 S container all the elements are available to the application.

From the above analysis we know that the only performance impact of this workaround is that only 124 S resource elements are available for the application that uses GCCP0. For the application using GCCP1, there is no performance impact; all 128 S resource elements are available.

The following is an example of the application code related to the workaround:

```
    RACPD_CPreambleDetector    *RACPDPtr;
RACPD_CPreambleDetector_SCorrSpacePagesAddReq  CorrPAddReqObj;
    RACPD_CPreambleDetector_SCorrSpacePagesAddRes   CorrPAddResObj;

    RACCM_TCorrElemTIndex  SIndx = 4;
    CorrPAddReqObj.corrSpaceId      = 0;
    ... ...
    CorrPAddReqObj.pageSNum         = 1;
      CorrPAddReqObj.pageSIndexPtr    =

        RACPD_CPreambleDetector_corrSpacePagesAdd (RACPDPtr,
                          & CorrPAddReqObj,
                          & CorrPAddResObj );
```

Alternatively, an assertion on the RACPD/RACPD corrSpacePagesAdd has been added to RACFL. If the corrSpace is located in GCCP#0 and if pageSNum == 1, there is an assertion when pageSIndexPtr[0] == 0.

Advisory 1.3.8 *EMAC Boot Issue*

NOTE: Supersedes Advisory 1.0.4 - SGMII EMAC Boot Start-up Issue

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details: The EMAC ready announcement frame is not transmitted at correct time when the TCI6487/8 device is booted in master, slave, and force modes.

When the DSP is booted in EMAC master/slave boot modes (boot modes 4, 5), the DSP transmits an Ethernet Ready Announcement (ERA) frame in the form of a BOOTP request. The BOOTP request is intended to inform the host server that the DSP is ready to receive boot packets. The ERA frame packet is described in more detail in the *TMS320TCI648x Bootloader User's Guide* (literature number SPRUEA7).

Silicon Revision 1.0	Silicon Revision 1.1, 1.2, and 1.3	Silicon Revision 2.0 and later
Ethernet-ready announcement packet was sent out without checking link status.	Ethernet-ready annoucement packet is sent out after link is up and after SERDES PLL lock is obtained.	Ethernet-ready annoucement packet is sent out after link is up and after SERDES PLL lock is obtained.

Texas Instruments will fix the Ethernet Ready Announcement frame transmission in the next silicon revision for TCI6487/8 devices.

Workaround 1: Have the host that is responsible for sending the boot packets broadcast a small boot table with the program that is shown in the example below. This will cause any TCI6487/8 device to restart the EMAC boot procedure (without configuring the MAC peripheral again) and re-transmit the ERA.

Re-send ERA packet code:

```
.text

BOOT_REENTRY_ADDR  .equ 03c000110h
BOOT_EMAC_OPT      .equ 01088480Ah
NETIF_CHANNEL_OPEN .equ 03c0040ACh

;; ------------------------------------------------------------------------ ;;
;;  MVKx:   Helper MVK that issues MVK or MVKL/MVKH as needed.              ;;
;; ------------------------------------------------------------------------ ;;

MVKx    .macro  xcst, xdst

        .if ((:xcst:) & 0xFFFF8000) == 0
        MVK     xcst, xdst
        .elseif ((:xcst:) & 0xFFFF8000) == 0xFFFF8000
        MVK     xcst, xdst
        .else
        MVKL    xcst, xdst
        MVKH    xcst, xdst
        .endif
        .endm

.def _c_int00

_c_int00:

      MVKx BOOT_EMAC_OPT, A1
      MVKx 0x26, A4               ; overwrite option field in EMAC bootparam
      STH  A4, *A1
      NOP  4

      MVKx NETIF_CHANNEL_OPEN, A1
      MVKx 0, A4
      MVKx 0, B4
```

```
      B    A1

      ADDKPC LABELIT,B3,4

LABELIT:

      MVKx BOOT_REENTRY_ADDR,  B3
      BNOP B3, 5
```

Workaround 2: The host server would need to rely on prior knowledge of the DSP MAC address to transmit boot packets to the correct DSP. The DSP will be ready to receive EMAC boot packets within 2 ms following deassertion of reset.

In the scenario where the boot server reads the MAC address of the DSP from the ERA packet, the procedure would need to be changed. After some customer TBD delay where the ERA is not received, the host sends the broadcast packet with the payload described in Workaround 1.

Advisory 1.3.9 *IP Block Containing CIC, CFGC, DTF, and IPC Registers Does Not Return Write Request Correctly*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details:

For every write transaction issued by a master peripheral to a slave, the bus protocol internal to the device (VBUS) makes the slave send a write status back to the master that initiated the transfer. This acknowledges that the write actually completed. For each write transaction, the master sends a transaction ID (xid) with each write that is issued to a slave. The slave returns the same ID that was used for the write transaction along with an end-of-transaction signal.

The master endpoint on the SCR, to which status is being returned, has its own arbitration block. This arbitration block sends a ready/not-ready signal to each slave denoting whether it is ready to receive status or not. If it responds as ready to a slave, the status is forwarded that clock cycle to the master. When it is not ready, the transaction repeats until it is ready.

This issue was found in a chip-level block that consolidates the following modules into a single slave endpoint: CIC, CFGC, DTF, and IPC. When this block sends write status back to a master via SCR F, the status signal from SCR F is ignored. This means that if SCR F sends back a not-ready signal during a write status transaction, the write status is lost since it is not forwarded to SCR D and it is no longer generated by the slave. If the block was behaving properly, the status should have been re-sent the next clock cycle.

The SCR F write status arbitration block returns a not-ready signal when two or more slaves return status on the same clock cycle. One slave wins the arbitration and the others see a not ready.

Each master behaves differently when write status is lost. The most common interface that is writing to these registers is GEM. Note that if GEM misses a write status response, it eventually stalls the CFG port. This happens exactly three CFG transactions (read or write) later.

The GEM CFG bus uses transaction IDs 0x0 to 0xB for IDMA transfers and 0xC to 0xF for other transfers. As transactions are issued, they recycle through the IDs in a round-robin fashion. The GEM does not reuse an ID until it gets write status back for that ID. Table 18 shows an example of how the write status never makes it back to the GEM, it issues transactions for the other three IDs and, when it gets to the one that did not see status, the GEM hangs and never issues further commands because it is stuck waiting for the status.

Table 18. GEM Transaction IDs

TRANSACTION	R/W	XID	WSTATUS ID RETURNED
1	W	D	D
2	W	E	E
3	R	F	<N/A>
4	W	C	C
5	W	D	NO STATUS RETURNED
6	R	E	<N/A>
7	W	F	F
8	W	C	C

The next transaction is never issued since the GEM is expecting to use xid D and no status is ever returned. Note that this means that the CFG port is now completely stalled and no reads or writes can be issued. In addition, once the CFG port stalls, the MDMA port can stop issuing writes as well.

NOTE: This is a description of how GEM issues transactions since it is the most common master of these blocks. Other masters writing to these blocks are somewhat more tolerant, but stall (stop issuing commands) eventually. The write accesses in the workaround below refer to all masters, not only the GEM.

Workaround: There are three ways to avoid this issue. Workaround 3 is the recommended method.

Workaround 1

Do not issue any writes to the affected peripherals. Note that reads are acceptable, assuming the affected block (CIC, CFGC, DTF, or IPC) is not written to. Systems that already follow this rule are not affected by this issue. The following lists how to avoid writes on each of these peripherals if they are currently used:

- **CIC** – The main writes to this module should be limited to configuration writes. Therefore, systems should program this block during system initialization. Most systems use the CIC as only a means of routing an event. This means that the CIC is programmed only to route an event input to a particular event output. The other use model of this block is to use the combiner that allows you to combine multiple events into a single output. For systems only utilizing the routing feature and not using the combiner, no writes are necessary during run-time operation. When using the combiner, an interrupt flag is set that requires a software write to clear the flag. The document infers that all events set flags that need to be cleared by the software. This is not true as it is limited only to when the combiner is used. Do not use the combiner feature of the CIC and program during system initialization time and this block should be write-free during run time.
- **CFGC** – There should only be need to write to this block during initialization of the device. Perform these initial configurations early before writes to the other blocks occur. The only writable register in this block is the DEVCFG1 register that controls enabling of internal McBSP, CLKS, and SYSCLKOUT.
- **DTF** – There should be no writes to this block so it should be unaffected.
- **IPC** – These registers are the most widely used in this set of peripherals. The EDMA can be used to avoid writing to the IPC registers. Set up a dummy EDMA transfer and use the EDMA completion interrupt as your signal to the intended core. The CIC cannot be used as it would require a write across SCR F to manually trigger the interrupt, thus creating a potential violation of this issue. This may add some additional latency but it is the only way to generate an interrupt to another core without writing to the IPC registers.

Workaround 2

Do not write to the FSYNC or SEM block during run time. If your system performs no writes to these modules during run time, it is not affected by this issue.

By studying the write status behavior of each of the modules on SCR F, we have found that as long as there are no writes to the FSYNC or SEM, it is safe to write to any of the other modules on SCR F. All modules, except the FSYNC and SEM, respond back with write status on the cycle following the write. On SCR F, since there is only a single master port, all writes are forced to be back to back. Therefore, all write status responses are back to back as well. The FSYNC and SEM blocks respond back three cycles after a write. For this reason, writing to these blocks poses potential write status conflicts.

Figure 13 shows the SCR F write requests to and write staus from the modules. The following descibes the actions shown in the figure:

- On cycle x, there is a write request from SCR F to FSYNC/SEM. Its write status is expected to come after three cycles.
- On cycle x + 1, no command is issued in this cycle.
- On cycle x + 2, there is one write request from SCR F to CIC/CFGC/DTF/IPC. The SCR F sends a not-ready signal to the CIC/CFGC/DTF/IPC, but the CIC/CFGC/DTF/IPC sends write status irrespective of the not-ready signal, assuming that it responded to the write request and SCR F received it properly.
- The SCR F accepts the write status from the FSYNC/SEM and the write status from the CIC/CFGC/DTF/IPC is lost.

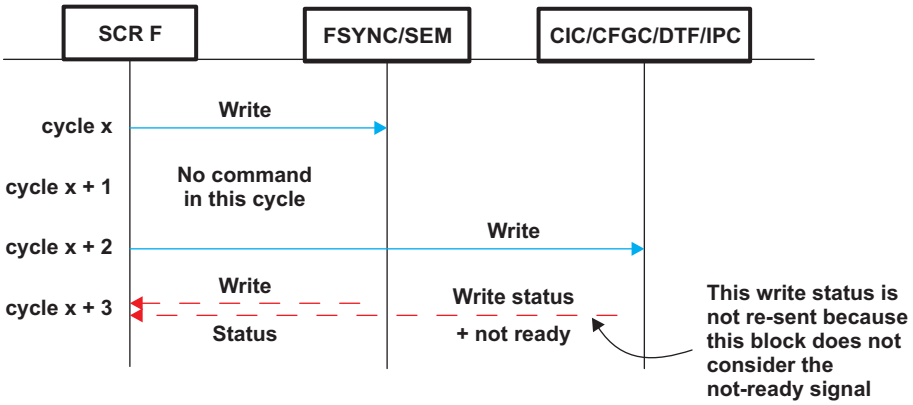


Figure 13. SCR F Write Requests/Write Status

Workaround 3 (Recommended Workaround)

This workaround involves aspects of both workarounds 1 and 2 to create a workaround for systems that utilize the CIC combiner, IPC registers, SEM, and FSYNC. The following are the basic rules:

- Guarantee single master access for CIC, FSYNC, or IPC writes using a SEM.
- A single global semaphore in the SEM module should be assigned for all write accesses to CIC, IPC, FSYNC, and SEM. Note that the write to free the global SEM used to protect these blocks does not have to be protected. All other writes to the SEM block must be protected by the global SEM.
- When using the SEM, you must use the direct or combination access methods for the global semaphore. Any mode can be used for other semaphores as long as the writes are protected.
- If you are following any single write or a block of writes to the FSYNC and/or SEM with writes to the CIC or IPC modules, perform a dummy read from the FSYNC or SEM block before performing the write to the CIC or IPC modules. Note that the dummy read must be from the last block that was written to and must use the same master.
- Try not to hold on to the global semaphore for this workaround for too long as other masters may be stalled waiting for the semaphore as well.
- Ensure that any interrupts taken during this time do not try to access any of these blocks (CIC, FSYNC, IPC, or SEM).

Before writing to the CIC, FSYNC, IPC registers, or SEM, use a single global semaphore in the SEM module to secure the space to all of these modules exclusively. You must use the direct or combination methods as described in the *TMS320TCI6487/8 Semaphore User's Guide* (SPRUEF6). Since this requires no writes to gain a semaphore, it is safe to use the SEM block here. Do not use the indirect mode as this requires a write. If you perform writes to FSYNC and/or SEM while you have the global SEM, you should insert a dummy read after the writes are completed. Note that you are

allowed a block of writes to the FSYNC and/or SEM block, but the last one must be followed by the dummy read. In addition, you are allowed to mix FSYNC/SEM writes with CIC/IPC writes, but ensure that a dummy read always follows the FSYNC/SEM writes before writes are allowed to anything else, especially the CIC/IPC. This dummy read must also be from the same block that the last write was issued to using the same master. Freeing the global SEM involves a write but the write status is guaranteed to land back at the master before the SEM block accepts a read from another master that would grant it access to the CIC, FSYNC, IPC, or other SEM writes. Again, note that this is only legal SEM write allowed without being protected. All other SEM writes need to be managed or protected using this workaround as well. It is highly recommended that you keep the semaphore "checked-out" for as little time as possible as other masters may be pending its availability.

Examples:

- Single Write to CIC/IPC
  - The master reads the global semaphore from the SEM module using direct or combination mode to ensure exclusive access to the CIC/IPC register space.
  - Once granted, the master performs a write to the destination module.
  - Free the global semaphore by writing to the SEM module.
- Single Write to FSYNC/SEM
  - The master reads the global semaphore from the SEM module using direct or combination mode to ensure exclusive access to the FSYNC/SEM register space.
  - Once granted, the master performs a write to the destination module.
  - The same master performs a dummy read from the same module (FSYNC or SEM).
  - Free the global semaphore by writing to the SEM module.
- Multiple Writes to CIC/IPC
  - The master reads the global semaphore from the SEM module using direct or combination mode to ensure exclusive access to the CIC/IPC register space.
  - Once granted, the master performs a writes to the destination module(s). Writes to these two modules can be interleaved.
  - Free the global semaphore by writing to the SEM module.
- Multiple Writes to FSYNC/SEM
  - The master reads the global semaphore from the SEM module using direct or combination mode to ensure exclusive access to the FSYNC/SEM register space.
  - Once granted, the master performs a writes to the destination module(s). Writes to these two modules can be interleaved.
  - The same master performs a dummy read from the same module as the last write (FSYNC or SEM).
  - Free the global semaphore by writing to the SEM module.
- Multiple Writes to CIC/FSYNC/IPC/SEM
  - The master reads the global semaphore from the SEM module using direct or combination mode to ensure exclusive access to the CIC/FSYNC/IPC/SEM register space.
  - Once granted, the master performs a writes to the destination module(s). Note that writes to modules may be interleaved, but a dummy read must be performed after each single or block of FSYNC/SEM writes. The dummy write must be from the same module as the last write (FSYNC or SEM) and use the same master.
  - Free the global semaphore by writing to the SEM module.

Advisory 1.3.12 *DMA Access to L2 SRAM May Stall When the DMA Has Lower Priority Than the CPU*

Revision(s) Affected: 1.3, 1.2, 1.1, 1.0

Details: The L2 memory controller in the GEM has programmable bandwidth management features that are used to control bandwidth allocation for all requestors. There are two parameters to control this, command priority and arbitration counter MAXWAIT values. Each requestor has a command priority and the requestor with the higher priority wins. However, there are also counters associated with each requestor that track the number of cycles each requestor loses arbitration. When this counter reaches a threshold (MAXWAIT), which is programmed by the user (or default value), the losing requestor gets an arbitration slot and wins for that cycle. There are four such requestors: CPU, DMA (SDMA and IDMA), user cache coherency operation, and global cache coherence. Global-coherence operations are highest priority, while user-coherence operations are lowest priority. However, there is active arbitration done for the CPU and the DMA (SDMA/IDMA) commands. The priority for DMA commands comes from an external master as part of the SDMA command or a programmable register, IDMA1_COUNT, in the GEM for IDMA commands. The priority for CPU accesses to L2 is in a programmable register, CPUARBU, in the GEM. For the default priority values, see Table 19.

More details on the bandwidth management feature can be found in the *C64x+ Megamodule Reference Guide* (SPRU871).

Table 19. TCI6487/8 Default Master Priorities

MASTER	DEFAULT MASTER PRIORITIES (0 = Highest priority, 7 = Lowest priority)	PRIORITY CONTROL
EDMA3TCx	0	QUEPRI.PRIQx (EDMA3 register)
SRIO (Data Access)	0	PER_SET_CNTL.CBA_TRANS_PRI (SRIO register)
SRIO (Descriptor Access)	1	PRI_ALLOC.SRIO_CPPI
EMAC	1	PRI_ALLOC.EMAC
RAC Back-End (TCI6488 only)	7	RAC register
C64x+ Megamodule (MDMA port)	7	MDMAARBE.PRI (C64x+ Megamodule register)
C64x+ Megamodule (CPU Arbitration control to L2)	1	CPUARBU (C64x+ Megamodule register)
C64x+ Megamodule (IDMA channel 1)	0	IDMA1_COUNT (C64x+ Megamodule register)

NOTE: When the SDMA has finished sending all of its commands to the L2 controller, the C64x+ Megamodule drops the transfer priority down to 7 if no further commands are in the pipeline. This condition happens when there is a single-word access, a burst of <32B with no other SDMA commands pending or for the last 64B only of a burst that is >64B with no other SDMA commands pending. This effective priority level is what the L2 controller uses to arbitrate these SDMA commands with the CPU, irrespective of the master peripheral's actual programmed priority value. Therefore, priority 7 is not a valid priority level for the CPU. If, for any reason, this "demoted" transfer is still pending upon initiation of another transfer, it automatically inherits the priority of that new transfer and is pushed through such that it does not stall the new transfer.

To enable bandwidth management, the L2 memory controller has an internal (non-user visible) counter that counts MAXWAIT every cycle that a DMA command is blocked because of a CPU access. When the internal counter reaches the MAXWAIT threshold, it is supposed to stay saturated at that value and force the DMA access to win arbitration over the CPU. In the case where DMA priority is less than CPU priority, the internal counter does not saturate at the MAXWAIT threshold value. Instead, it wraps around and keeps counting, thereby, giving more bandwidth to the CPU than intended by the MAXWAIT threshold value. The result is that the DMA may lose to the CPU over multiple arbitration cycles. This typically happens when CPU accesses keep the L2 memory controller busy every cycle; for example, a continuous stream of L1D write misses to L2.

Workaround: Set the CPU at a lower priority than the DMA commands to L2. The priority for CPU accesses to L2 is in a programmable register, CPUARBU, in the GEM. However, lowering the CPU priority may impact the performance since CPU accesses to L2 may stall due to DMA accesses, in case of contention.

The CPUARBU should not be set to 7 (see the Note above). This means that since there can be cases where the DMA will be at priority 7, there is no complete workaround for this issue. The workaround recommended below ensures that only the types of accesses mentioned in the note can potentially create an issue, that is single-word accesses, bursts of <32B, or the last 64B of a burst >64B. The remainder of any bursts >64B are not affected. This condition is only present when there is heavy CPU activity that can block these SDMAs. A gap of even 1 CPU cycle between the CPU accesses to L2 allows any of these transfers to slip into the L2, regardless of priority.

The recommended workaround is to place the CPU priority to L2 (CPUARBU) to priority 6. That leaves priorities 0-5 free for other accesses to L2 (see Table 20).

Table 20. TCI6487/8 Valid Priority Settings

CPU PRIORITY	ALLOWED SDMA PRIORITIES
0	-
1	0
2	0-1
3	0-2
4	0-3
5	0-4
6	0-5
7	-

This issue has been fixed on silicon revision 2.0.

5 Silicon Revision 1.2 Usage Notes and Known Design Exceptions to Functional Specifications

NOTE: TCI6487/8 Silicon Revision 1.2 was a manufacturing process change. No design changes were made from revision 1.1 to revision 1.2. All usage notes and advisories for TCI6487/8 silicon revision 1.1 also apply to revision 1.2.

5.1 Silicon Revision 1.2 Usage Notes

Silicon revision 1.2 applicable usage notes have been found on later silicon revisions; for more detail, see Section 4.1 and Section 2.1.

5.2 Silicon Revision 1.2 Known Design Exceptions to Functional Specifications

Table 21 lists the silicon revision 1.2 known design exceptions to functional specifications. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. Therefore, advisory numbering may not be sequential.

All other known design exceptions to functional specifications for silicon revision 1.2 still apply and have been moved up; see Table 3 and Section 2.2 , Section 3.2, or Section 4.2.

Table 21. Silicon Revision 1.2 Advisory List

Advisory 1.2.8 *Potential Random E-fuse Blow*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: In the final stages of screening the TCI6487/8 device for qualification, a subtle issue has been uncovered involving e-fuses being inadvertently blown during power up if an improper power and clock sequence is applied to the device. The e-fuse controller on the TCI6487/8 device may unintentionally blow e-fuses during power up when an invalid power sequence is used:

- The e-fuse controller has a defect that gates the output of an accidental programming prevention circuit with a clocked register.
- If proper sequencing of supplies and clocks is not maintained, then the program enable on the e-fuse ROM will be active until a valid reset (SYS_INITZ) is propagated to the register.
- The result is susceptibility to inadvertent blowing of e-fuses.

If the 1.1-V CV_{DD} scaled supply ramps before the 1.8-V and 1.1-V fixed supplies, the logic in the CV_{DD} domain powers up in random state. In this random state, there is small probability that conditions are met for inadvertent blowing of e-fuses.

The possible impact is that e-fuses that are not supposed to be blown are blown.

- Which e-fuses could inadvertently be blown is random.
- The type of e-fuse randomly blown will determine the end impact to the system, ranging from no impact to severe impact.
- Each power up event is a new opportunity for exposure to the issue in which e-fuses could be unintentionally blown.
- The probability of this is low, but not low enough.

Workaround: Guarantee that the 1.8-V DV_{DD} device input clocks and clock selects are active before a 1.1-V CV_{DD} scaled supply ramps (see Table 22 and Figure 14).

TI's TMS test procedure follows the above recommendation to ensure that no devices are shipped with unintentionally blown e-fuses.

NOTE: E-fuses are used in multiple areas within the device. They are used for memory repair, device ID, EMAC ID, etc.

Table 22. Device Input Clock Timing Parameter Descriptions

NO.	PARAMETERS	MIN	MAX	UNITS
1	DV_{DD18} at 0.8 V to CV_{DD} at 0.4 V	0.5		ms
2	Stable clock to CV_{DD} at 0.4 V	100		µs
3	DV_{DD11} at 0.8 V to CV_{DD} at 0.4 V	0.5		µs

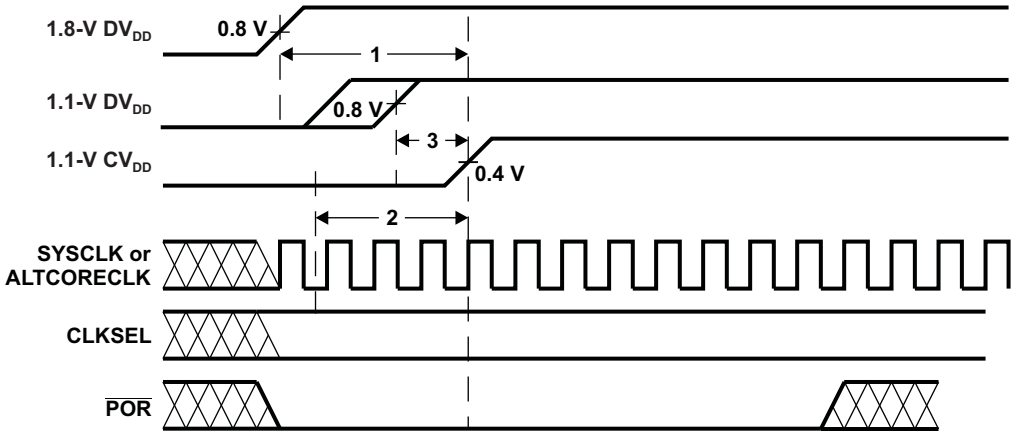


Figure 14. Correct Device Input Clocks, Clock Selects, and Scaled Supply Timings

Advisory 1.2.10 *EDMA3CC COMPACTV Issue*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: A issue has been found inside the EDMA3 channel controller (EDMA3CC). The logic for decrementing the completion request active (COMPACTV) counter is incorrect for devices having six or more EDMA3 transfer controllers (EDMA3TCs). Therefore, the TCI6487/8 devices are affected by this issue.

The COMPACTV field inside the channel controller status register (CCSTAT) indicates the count for the number of outstanding transfer requests requiring completion status that have been submitted to the transfer controllers. The channel controller increments this count every time a transfer request (TR) is submitted and is programmed to report completion (the TCINTEN or TCCHEN or the ITCINTEN or ITCCHEN bits in OPT in the parameter entry associated with the TR are set). The counter decrements for every valid transfer completion code (TCC) received back from the transfer controllers. The issue occurs because the channel controller decrements the counter by an insufficient value when multiple responses are received concurrently from multiple (two or more) transfer controllers. Thus, the counter may gradually increase over time until it saturates at 0x3F. If at any time the count reaches a value of 0x3F, the channel controller does not service new TRs until the count is less than 0x3F (which will happen when a transfer completion code is received from a transfer controller for an in-flight request). Once the state is reached where the counter is close to the saturation value of 0x3F, the performance of the EDMA decreases dramatically. This decreased performance happens because the channel controller will artificially limit its number of TRs in flight to the COMPACTV saturation value thereby preventing full usage of the available TCs. When the count reaches 0x3F, the TCCERR bit is set in the channel controller error register (CCERR) causing an error interrupt when enabled.

Workaround: The workaround is achieved by having the DSP directly program one of the transfer controllers (bypassing the channel controller) with a transfer request that requires completion. This request avoids the COMPACTV increment (because TC is programmed directly) and forces a COMPACTV decrement when the TC responds to the CC with the completion signaling.

A specific transfer controller and a specific TCC value should be dedicated in the system for this workaround. Table 23 lists the suggested TC to use for each device and the gives the reasoning for this suggestion. However, the specific TC should be selected based on the end-system usage.

Table 23. Suggested TC Use

DEVICE	TC SUGGESTION	REASON
TCI6487	TC2 or TC5	TC2 is suggested because TC0 and TC1 can replace its connectivity. TC0 can be used for TCP/VCP transfers. TC5 is suggested because TC4 can replace its connectivity. TC4/TC3 can be used for AIF transfers.
TCI6488	TC5	TC5 is suggested because TC4 can replace its connectivity. TC4/TC3 can be used for AIF transfers.

The DSP should poll the COMPACTV field often enough such that the counter is not allowed to exceed 0x30. The actual COMPACTV polling interval may need to be set through experimentation on the specific end system, since the rate of increment of the counter is system and load specific.

Upon polling, if the value of the COMPACTV field is greater than a certain threshold (0x20 is suggested), then the DSP should program the TC with a COMPACTV decrement transfer. Upon completion of that transfer (as signaled in the CC IPR register) the COMPACTV field should be re-checked, and another COMPACTV decrement transfer submitted until the value of the counter is less than the threshold.

NOTE: Care must be taken such that the software does not over-decrement the counter since at the time of polling multiple requests may be in flight in the system and may result in additional decrements compared to the current observed value. If too many decrements occur, the counter may roll under from 0x0 to 0x3F and accidentally result in saturation of the counter. This is why a value of 0x20 is suggested as the threshold value (sufficiently large with respect to the number of actual requests that may be outstanding).

This workaround requires that a specific TC instance is dedicated to the COMPACTV decrement transfer. The reason is that, depending on the nature of the traffic on a given queue/TC, it may be difficult to control the timing of the normal CC TR submission to that TC versus the DSP programming of that TC. There is no hardware protection to prevent corruption of the TC registers in the case that both CC and DSP software attempt to program the TC simultaneously.

For the base addresses of the TCs, see the device-specific data manual, *TMS320TCI6487/8 Communications Infrastructure Digital Signal Processor* (literature number SPRS358). A brief summary of the TC registers to be configured are provided in Table 24.

Table 24. TC Registers Summary

ADDRESS	REGISTER DESCRIPTION	SUGGESTED VALUE
TCx Base + 0x0200	Prog Set Options	See the Prog Set Options Register description below.
TCx Base + 0x0204	Prog Set Src Address	See Prog Set Src/Dst Address Register description below.
TCx Base + 0x0208	Prog Set Count	0x00010004 (ACNT = 4 and BCNT =1)
TCx Base + 0x020C	Prog Set Dst Address	See Prog Set Src/Dst Address Register description below.
TCx Base + 0x0210	Prog Set B-Dim Idx	0x0 (don't care since BCNT=1). Writing to the PBIDX register triggers the transfer. Thus, this register should be written.

Note: The five registers listed in Table 24 should be written in the sequence shown (i.e., top to bottom). The last write, to the Prog Set B-Dim Idx register, triggers the transfer.

Prog Set Options Register

The Prog Set Option register is shown in Figure 15. The TCINTEN bit should be set to 0x1. The TCC code should be set to some known value that is not used by other requests in the system. The other fields should be set to 0x0. Upon completion of the transfer, the TCC value will be set in the corresponding bit in the IPR/IPRH registers. The software should poll for this bit in the IPR/IPRH registers and then clear it with the ICR/ICRH registers before programming the next COMPACTV decrement transfer.

Figure 15. Prog Set Options Register

31	23	22	21	20	19	18	17	16
Reserved		TCCH_EN	Rsvd	TCINT_EN	Reserved		TCC	
R-0		R/W-0	R-0	R/W-0	R-0		R/W-0	

15	12	11	10	8	7	6	4	3	2	1	0
TCC		Rsvd	FWID		Rsvd	PRI		Reserved		DAM	SAM
R/W-0		R-0	R/W-0		R-0	R/W-0		R-0		R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Prog Set Src/Dst Address Register

Although the user can specify any address for src/dst, one of the following settings is suggested:

1. Set the src/dst address as 0x31000000. This is a reserved location and transfer to this address takes less latency. However, the bus error (BUSERR) bit in the TCx error status register(ERRSTAT) will be set (the TCx error details register (ERRDET)

will also be set). This TCx error should be ignored. This error is localized to the dedicated TC for this transfer and will not affect the system. Also, by default, the BUSERR will not cause the EDMA3TC error interrupt. This interrupt gets generated only when the TCx error enable (ERREN) register is set.

2. The other option is to set the src/dst address to the EDMA3TCx or the EDMA3CC peripheral ID (PID) register location. This transfer has more latency when compared to option 1, but will not cause TCx BUSERR condition.

Example code for programming the TC for this workaround (this example uses TC5):

```
#include <csl_edma3.h>
#include <soc.h>

#define EDMA3TC_POPT_REG (*(volatile Uint32*)(CSL_EDMA3TC_5_REGS + 0x200))
#define EDMA3TC_PSRC_REG (*(volatile Uint32*)(CSL_EDMA3TC_5_REGS + 0x204))
#define EDMA3TC_PCNT_REG (*(volatile Uint32*)(CSL_EDMA3TC_5_REGS + 0x208))
#define EDMA3TC_PDST_REG (*(volatile Uint32*)(CSL_EDMA3TC_5_REGS + 0x20C))
#define EDMA3TC_PBIDX_REG (*(volatile Uint32*)(CSL_EDMA3TC_5_REGS + 0x210))
#define COMPACTV_XFER_ADDRESS (0x31000000)
#define COMPACTV_XFER_COMPLETION_CODE (63) /* dedicate one TCC value for this */

void triggerCompactvDecTransfer ()
{
       EDMA3TC_POPT_REG = CSL_EDMA3_OPT_MAKE(FALSE, FALSE, FALSE, TRUE,\
              COMPACTV_XFER_COMPLETION_CODE, FALSE,\
              CSL_EDMA3_FIFOWIDTH_NONE, FALSE, FALSE,\
              CSL_EDMA3_ADDRMODE_INCR, CSL_EDMA3_ADDRMODE_INCR);

       EDMA3TC_PSRC_REG = COMPACTV_XFER_ADDRESS;
       EDMA3TC_PCNT_REG = CSL_EDMA3_CNT_MAKE(4, 1);
       EDMA3TC_PDST_REG = COMPACTV_XFER_ADDRESS;
       EDMA3TC_PBIDX_REG = CSL_EDMA3_BIDX_MAKE(0, 0);
}
```

6 Silicon Revision 1.1 Usage Notes and Known Design Exceptions to Functional Specifications

6.1 Silicon Revision 1.1 Usage Notes

Silicon revision 1.1 applicable usage notes have been found on later silicon revisions; for more detail, see Section 4.1 and Section 2.1.

6.2 Silicon Revision 1.1 Known Design Exceptions to Functional Specifications

All known design exceptions to functional specifications for silicon revision 1.1 still apply and have been moved up; see Table 3 and Section 2.2, Section 3.2, Section 4.2, or Section 5.2.

7 Silicon Revision 1.0 Usage Notes and Known Design Exceptions to Functional Specifications

7.1 Silicon Revision 1.0 Usage Notes

Silicon revision 1.0 applicable usage notes have been found on later silicon revisions; for more detail, see Section 4.1 and Section 2.1.

7.2 Silicon Revision 1.0 Known Design Exceptions to Functional Specifications

Table 25 lists the silicon revision 1.0 known design exceptions to functional specifications. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. Therefore, advisory numbering may not be sequential.

All other known design exceptions to functional specifications for silicon revision 1.0 still apply and have been moved up; see Table 3 and Section 2.2, Section 3.2, Section 4.2, or Section 5.2.

Table 25. Silicon Revision 1.0 Advisory List

Advisory 1.0.1 *SRIO: Packet Forwarding Cannot Be Used With NREAD Response Packets Greater Than 16 Bytes*

Revision(s) Affected: 1.0; Fixed in revision 1.1

Details: Packet forwarding uses programmable look-up tables to direct incoming packets to an outbound port when the packets do not belong to the local device. Packet forwarding is carried out at the logical layer of the Serial RapidIO (SRIO) without the interaction of the CPU. The SRIO logical layer copies incoming packets from an inbound buffer to an outbound buffer. When used for packet forwarding, it forwards all types of packets, including response, maintenance, DOORBELL, and message packets.

The current SRIO design fails to correctly copy response packets with a payload greater than 16 bytes from the inbound to the outbound buffer. The first 16 bytes are correctly copied, but the remainder of the payload is discarded. This issue affects only NREAD response packets since their payloads can be up to 256 bytes. Packet types with small responses, such as NWRITE_R, maintenance, message, and DOORBELL packets are not affected by this issue.

Workaround 1: Use a data *push* model, where each device in the daisy chain only submits write requests. Using this approach will avoid the issue and provide the lowest latency solution.

Workaround 2: Two options exist if NREAD response packets cannot be avoided; for example, when reading core dump information from an unresponsive processor which is unable to initiate traffic by itself. The first option is to use software to segment read requests into 16-byte NREADs. Note that this option will work functionally, but may take too much time.

The second option is illustrated in Figure 16.

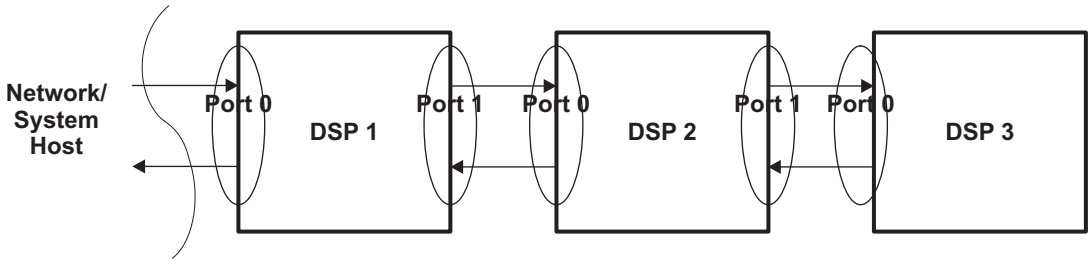


Figure 16. Daisy-Chain Example

In this example, assume that DSP3 is down and the system host wants to do a large NREAD of DSP3 to examine the core dump. The issue discussed above prohibits the NREAD from completing correctly because, as the response packets from DSP3 are sent back, they are corrupted by DSP2 and DSP1 packet forwarding. Instead, the system host needs to request that the adjacent DSP (DSP2) generates the NREAD request to DSP3. The NREAD responses are sent to DSP2 and temporarily stored in memory. Then, DSP2 can generate NWRITE/NWRITE_R/SWRITE packets to the system host with the needed payload. These packets are correctly forwarded by DSP1 to the system host since they are request packets and not responses.

Advisory 1.0.3 *I2C ROM Boot Fails When in Big Endian Mode*

Revision(s) Affected: 1.0; Fixed in revision 1.1

Description When the TCI6487/8 device is reset with both of the following boot options selected: Big Endian mode and one of the I2C ROM boot modes, the device fails to load and execute the I2C ROM image.

Workaround: A change to the internal boot ROM of the TCI6487/8 device was made to ensure that I2C ROM boot modes work when the devices are in Big Endian mode. Do not use I2C ROM boots in Big Endian mode with silicon revision 1.0.

Appendix A Revision History

This silicon errata revision history highlights the technical changes made to the document.

Scope: Applicable updates relating to the TCI6487/8 device have been incorporated.

Table 26. TCI6487/8 Revision History

SEE	ADDITIONS/MODIFICATIONS/DELETIONS
Section 1.3	Silicon Updates: Modified Table 3, Silicon Revisions 1.0, 1.1, 1.2, 1.3, 2.0, 2.1 Updates
Section 2.2	Silicon Revision 2.1 Known Design Exceptions to Functional Specifications: Added the following new advisories: Advisory 2.1.7 - SPLOOP CPU Cross-Path Stall Advisory 2.1.8 - DMA Corruption of L1D$ Allocation Advisory 2.1.9 - Error Detection and Correction Incorrectly Reporting Error Advisory 2.1.10 - SRIO May Fail to Send Interrupt for Completed TX or RX Message Advisory 2.1.11 - Serial RapidIO Internal Digital Loopback is Not Always Stable
Section 5.2	Silicon Revision 1.2 Known Design Exceptions to Functional Specifications: Modified Advisory 1.2.8 - Potential Random E-fuse Blow

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 TI 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. Information of third parties may be subject to additional restrictions.

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.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com	Wireless	www.ti.com/wireless-apps
RF/IF and ZigBee® Solutions	www.ti.com/lprf		
	TI E2E Community Home Page		e2e.ti.com

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