TMS320C6472
Digital Signal Processor
Silicon Revisions 2.1, 1.2, 1.1, 1.0

Silicon Errata

TEXAS
INSTRUMENTS

Contents

List of Figures

List of Tables

TMS320C6472 DSP
Silicon Revisions 2.1, 1.2, 1.1, 1.0

1 Introduction

This document describes the silicon updates to the functional specifications for the TMS320C6472 digital signal processor; see the device-specific data manual, *TMS320C6472 Fixed-Point Digital Signal Processor* (literature number SPRS612).

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., TMS320C6472CTZ). Texas Instruments recommends one of two 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 CTZ and ZTZ packages is shown in Figure 1. Figure 1 also shows an example of C6472 package symbolization.
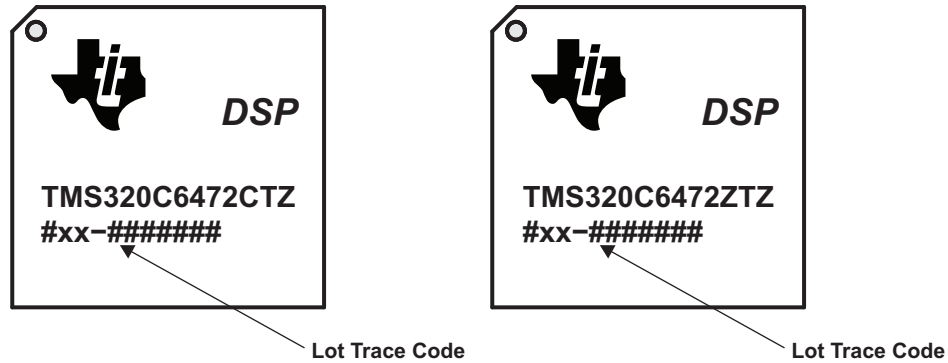
Figure 1. Lot Trace Code Examples for TMS320C6472 (CTZ/ZTZ Packages)

Silicon revision correlates to the lot trace code marked on the package. This code is of the format #xx-#######. If xx is "10", then the silicon is revision 1.0. Table 1 lists the silicon revisions associated with each lot trace code for the C6472 devices.

Table 1. Lot Trace Codes

LOT TRACE CODE (xx)	SILICON REVISION	COMMENTS
21	2.1	Silicon revision 2.1
12	1.2	Silicon revision 1.2
11	1.1	Silicon revision 1.1
10	1.0	Initial silicon revision

The C6472 device contains multiple read-only register fields that report revision values. The Silicon Revision ID Register and the JTAG ID Register are chip-level revision registers. The Silicon Revision ID Register provides the silicon revision in explicit fields. The silicon revision can be read directly from the MAJOR REVISION and MINOR REVISION fields within this register. The Silicon Revision ID Register is at address location 02A8 070Ch. Table 2 shows the contents of the Silicon Revision ID Register for each silicon revision. More details on the Silicon Revision ID Register can be found in the *TMS320C6472 Fixed-Point Digital Signal Processor* (literature number SPRS612).

The JTAD ID Register provides a VARIANT field that is associated with the silicon revision. The JTAG ID Register is at address location 02A8 0008h. Only the VARIANT field changes in this register. Table 2 shows the contents of the JTAG ID Register and the VARIANT field for each silicon revision. More details on the JTAG ID Register can be found in the *TMS320C6472 Fixed-Point Digital Signal Processor* (literature number SPRS612).

Table 2. Silicon Revision Registers

SILICON REVISION	SILICON REVISION ID REGISTER (02A8 070Ch)	JTAG ID REGISTER VALUE (02A8 0008h)
2.1	0021 0091h MAJOR REVISION (bits 23:20): 2h MINOR REVISION (bits 19:16): 1h	3009 102Fh VARIANT (bits 31:28): 3h
1.2	0012 0091h MAJOR REVISION (bits 23:20): 1h MINOR REVISION (bits 19:16): 2h	1009 102Fh VARIANT (bits 31:28): 1h
1.1	0011 0091h MAJOR REVISION (bits 23:20): 1h MINOR REVISION (bits 19:16): 1h	0009 102Fh VARIANT (bits 31:28): 0h
1.0	0010 0091h MAJOR REVISION (bits 23:20): 1h MINOR REVISION (bits 19:16): 0h	0009 102Fh VARIANT (bits 31:28): 0h

The 64x+ Megamodule contains a revision register and the 64x+ CPU core within the Megamodule also has a revision field in a status register. The Megamodule Revision ID Register provides the Megamodule revision in explicit fields. The Megamodule revision information can be read directly from the VERSION and REVISION fields within this register. The Megamodule Revision ID Register is at local address location 0181 2000h. Table 3 shows the contents of the Megamodule Revision ID Register for each silicon revision. More details on the Megamodule Revision ID Register can be found in the *TMS320C6472 Fixed-Point Digital Signal Processor* (literature number SPRS612).

The Control Status Register within the 64x+ CPU core provides CPU Revision fields that are associated with the silicon revision. The Control Status Resister (CSR) is part of the CPU's control register file. Only the CPU ID and REVISION ID fields in the CSR reflect CPU revision information. Table 3 shows the contents of the CPU ID and the REVISION ID fields in the CSR register for each silicon revision. More details on the CSR register can be found in the *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732).

Table 3. Megamodule Revision Registers

SILICON REVISION	MEGAMODULE REVISION ID REGISTER (0181 2000h)	CPU REVISION (CPU CSR Register)
2.1	0001 0005h VERSION (bits 31:16): 0001h REVISION (bits 15:0): 0005h	CPU ID (bits 31:24): 10h REVISION ID (bits 23:16): 00h
1.2	0001 0003h VERSION (bits 31:16): 0001h REVISION (bits 15:0): 0003h	CPU ID (bits 31:24): 10h REVISION ID (bits 23:16): 00h
1.1	0001 0003h VERSION (bits 31:16): 0001h REVISION (bits 15:0): 0003h	CPU ID (bits 31:24): 10h REVISION ID (bits 23:16): 00h
1.0	0001 0003h VERSION (bits 31:16): 0001h REVISION (bits 15:0): 0003h	CPU ID (bits 31:24): 10h REVISION ID (bits 23:16): 00h

1.3 Silicon Updates

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

Advisories are numbered in the order in which they were added to this document. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. If the design exceptions are no longer applicable or if the information has been documented elsewhere, those advisories are removed. Therefore, advisory numbering may not be sequential.

Table 4. Silicon Revisions 2.1, 1.2, 1.1, and 1.0 Updates

SILICON UPDATE ADVISORY	APPLIES TO SILICON REVISON				SEE
	2.1	1.2	1.1	1.0	
SRIO: Packet-Forwarding NREAD Operations Larger Than 16 Bytes Fail				X	Advisory 1
RGMII EMAC: Boot Start-Up Issue				X	Advisory 2
DDR2 EMIF: Clock Synchronization Issue				X	Advisory 3
TSIP: Receive Channel 4 Bitmap Corruption Issue				X	Advisory 4
Device Configuration: HOUT is Not Generated When HPI is Disabled				X	Advisory 5
DSP SDMA/IDMA: Unexpected Stalling of SDMA/IDMA Access to L2 SRAM		X	X	X	Advisory 6
Potential SerDes Clocking Issue		X	X	X	Advisory 7
Potential Insertion or Deletion of 2 Bits in SerDes Data Stream		X	X	X	Advisory 8
I2C Slave Boot Does Not Work			X	X	Advisory 9
Atomic Operations Fail to Complete		X	X	X	Advisory 10
EMU: Emulation Access Can Corrupt CPU Operation	X	X	X	X	Advisory 11
PMC: Local Reset (Ireset) Followed By Block Invalidate Hangs		X	X	X	Advisory 12
PMC: L1P Cache Not Invalidated During Ireset		X	X	X	Advisory 13
UMC: L2MPFAR Fails to Log CPU Protection Faults Under Certain Conditions		X	X	X	Advisory 14
L1D Cache: C64x+ L1D Cache May Lose Data or Hang DMA Operations Under Certain Conditions	X	X	X	X	Advisory 15
CPU: Back-to-Back SPLOOPs With Interrupts Can Cause Incorrect Operation on C64x+ CPU	X	X	X	X	Advisory 16
CPU: C64x+ CPU Incorrectly Generates False Exceptions for Multiple Writes	X	X	X	X	Advisory 17
PrivID For Non-CPU Masters Is Same as GEM0 CPU		X	X	X	Advisory 18
UTOPIA Lock-Up Issue		X	X	X	Advisory 19
DDR2 EMIF Buffers Not Totally Compensated by Default	X	X	X	X	Advisory 20
SRIO Port 0 Reset Affects Other Ports	X	X	X	X	Advisory 21
SRIO OUTBOUND_ACKID Field Not Read Correctly	X	X	X	X	Advisory 22
DMA Access to L2 SRAM May Stall When the DMA Has Lower Priority Than the CPU		X	X	X	Advisory 23
DMA Access to L2 SRAM May Stall When the DMA and the CPU Command Priority is Equal		X	X	X	Advisory 24
DMA Corruption of External Data Buffer	X	X	X	X	Advisory 25
SDMA/IDMA Blocking Issue Update: L2 Victim Traffic Due To L2 Block Writeback During Any Pending CPU Request		X	X	X	Advisory 26
L1P$ Miss May Block SDMA Accesses		X	X	X	Advisory 27
SPLOOP CPU Cross-Path Stall	X	X	X	X	Advisory 28
DMA Corruption of L1D$ Allocation	X				Advisory 29
Error Detection and Correction Incorrectly Reporting Error	X	X	X	X	Advisory 30
SRIO May Fail to Send Interrupt for Completed TX or RX Message	X	X	X	X	Advisory 31
Serial RapidIO Internal Digital Loopback is Not Always Stable	X	X	X	X	Advisory 32

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 sheet), and the behaviors they describe will not be altered in future silicon revisions.

2.1.1   Device: Heatsink/Airflow Recommended to Lower Case Temperature

It is strongly recommended that users complete system-level thermal analysis to account for details of heatsink requirements, airflow, and other factors in order to achieve the case temperature specification of 85°C. The latest power data for the TMS320TCI6486 device indicates that static power is a significant contributor to overall power. Since static power varies with case temperature and voltage, a lower case temperature can greatly impact the overall power consumption. Therefore, the use of a heatsink to lower the case temperature is an effective way to lower power consumption and help maintain the device at an operating temperature within datasheet specifications.

2.1.2   EMAC: Gigabit Mode Cannot Be Used With CPU Running at Speeds Lower Than 375 MHz

The EMAC internal bus frequency must be greater than or equal to the I/O bus frequency. The EMAC internal bus is clocked by SYSCLK7 of the PLL1 controller, which has a frequency equal to the CPU frequency divided by 3. The I/O bus frequency of the EMAC is determined by the bit rate being used: 1.25 MHz for 10 Mbps, 12.5 MHz for 100 Mbps, and 125 MHz for 1000 Mbps. This restriction applies whether RGMII or GMII mode is being used.

Note that if the CPU speed is less than 375 MHz, the gigabit mode of the EMAC (1000 Mbps) cannot be used since the SYSCLK7 frequency will be less than 125 MHz.

2.1.3   DDR2 EMIF: Delay Before CKE Goes High With Different Combinations of REFRESH_RATE and DDR Clock

The SDRAM refresh control register (SDRFC) contains a count value that is used for two purposes. At power up, it is used to control the delay before CKE goes high. Later, it is used to control the time between refreshes. The DDR2 JEDEC specification requires a 200 µs delay before CKE goes high during initialization. The calculation of the delay before CKE goes high involves the following:

CKE_DELAY = 16 * (0xD06) / 266.666 = 200.04 µs.

Note that the default value for REFRESH_RATE is 0xD06 after reset. Users must make sure that whenever the DDR2 is enabled the delay before CKE goes high is always longer than 200 µs. Table 5 lists a few typical calculated values for REFRESH_RATE to obtain the required 200-µs delay.

Table 5. 200-µs Delay Calculated Values

CLOCK PERIOD	REFRESH_RATE
3.75 ns	0xD04
5 ns	0x9C4
8 ns	0x61A

During normal operation, the DDR memories require a refresh cycle at an average interval of 7.8125 µs (MAX). The calculation of RERESH_RATE involves the following:

REFRESH_RATE = DDR2CLKOUT frequency × memory refresh period

If the DDR clock period is set at 3.75 ns, the RERESH_RATE would be:

REFRESH_RATE = 266.666 MHz × 7.8125 µs = 2082 = 0x822.

Table 6 lists a few typical calculated values (an average interval of 7.8125 µs).

Table 6. 7.8125-μs Interval Calculated Values

CLOCK PERIOD	REFRESH_RATE
3.75 ns	0x822
5 ns	0x61A
8 ns	0x3D0

If the DDR2 needs to be put into self-refresh mode or power-down mode, users need to write a new value to the REFRESH_RATE field of the SDRFC register to guarantee the 200-µs delay of CKE during power-up or self-refresh mode exit.

2.1.4 EMIF Read Incurs High Latency Under Certain Conditions

Reads can incur higher than expected latency under certain conditions even though they are higher priority than writes if there are continuous sequences of back-to-back single writes.

A DDR2 EMIF read with high priority could incur high latency under the following conditions:

- Continuous back-to-back single writes are issued to the DDR2 EMIF at an interval of $t_{RP} + t_{RCD} + t_{WR}$.
- While the high-priority read is waiting for t_{WTR} to expire, the t_{RAS} expires first, this causes the low-priority request to precharge the bank. The proper operation is that the low-priority request should not have issued a precharge if a high-priority request is still pending for that bank.
- A few other read requests from different masters arrive for the same bank at a particular timing offset.
- t_{RAS} is smaller than $t_{WTR} + 4 + $ CasLatency - 1.

The above conditions result in the following behaviors:

- Since t_{RAS} expires before t_{WTR}, the low-priority request to precharge the bank is performed before a read can be fired.
- After t_{RP} expires, the bank is re-activated due to the high-priority read.
- If a write comes in on the cycle just after the t_{RCD} expires, it will go through this loop again. If more single-word writes arrive just after the t_{RCD} expires, this keeps the high-priority read from happening until the pr_old_count expires.
- Since only single word writes are executed, the expiration of the pr_old_count for the high-priority read could be delayed by a factor of $(t_{RP} + t_{RCD} + t_{WR}) / 2$, in clock cycles. Normally, the pr_old_count of 255 impacts latency by (pr_old_count * clk_period * 2), worst case.

This issue has only been observed through an internal test, it appears to have a very low probability of affecting existing designs.

Increasing the programmed t_{RAS} value by a couple of clock cycles and programming a lower value of pr_old_count could eliminate or reduce the latency. The pr_old_count should not be reduced to an extreme value since it could dramatically impact system performance. For example, pr_old_count = 0 can reduce throughput by about 20%.

2.1.5 I2C Bus Hang After Master Reset

It is generally known that the I2C bus can hang if an I2C master is removed from the bus in the middle of a data read. This can occur because the I2C protocol does not mandate a minimum clock rate. Therefore, if a master is reset in the middle of a read while a slave is driving the data line low, the slave will continue driving the data line low while it waits for the next clock edge. This prevents bus masters from initiating transfers. If this condition is detected, the following three steps will clear the bus hang condition:

1. An I2C master must generate up to 9 clock cycles.
2. After each clock cycle, the data pin must be observed to determine whether it has gone high while the clock is high.
3. As soon as the data pin is observed high, the master can initiate a start condition.

2.1.6   EMAC Boot Using the RGMII Interface

For EMAC boot, whenever the RGMII interface is selected using the boot mode pins requires that the MAC-switch/PHY interface must operate at 1000Mbps. To boot using RGMII at a 10/100Mbps rate, the RGMII link must first be configured using the I2C boot mode pin selection. If an EMAC boot is attempted at a lower rate whenever the RGMII interface is selected using the boot mode pins, the EMAC boot will fail.

2.2   Silicon Revision 2.1 Known Design Exceptions to Functional Specifications

Table 7 lists the silicon revision 2.1 known design exceptions to functional specifications. Advisories are numbered in the order in which they were added to this document. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. If the design exceptions are no longer applicable or if the information has been documented elsewhere, those advisories are removed. Therefore, advisory numbering may not be sequential.

Table 7. Silicon Revision 2.1 Advisory List

Advisory 11　　　*EMU: Emulation Access Can Corrupt CPU Operation*

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

Details:　When debug software issues a low-priority emulation data interface (EDI) access to control register file registers concurrently with the CPU executing the compact instruction MVC R, ILC, there is a 1-cycle window that can cause the EMU access to take place and the assembly code MVC R, ILC to get dropped. The non-compact form of the MVC R, ILC instruction does not have the problem.

Workaround:　There is no workaround for this advisory.

　　　Debug software generally performs low-priority accesses only when you first start up CCStudio or run in real-time mode.

　　　Since the MVC R, ILC instruction is not a common instruction and there is only a 1-clock cycle window, the likelihood of encountering this issue is small.

Advisory 15   ***L1D Cache: C64x+ L1D Cache May Lose Data or Hang DMA Operations Under Certain Conditions***

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

Details: Under certain conditions, parallel loads with predication to the same cache line may cause victims to be dropped and/or the DMA to hang. All of the following conditions must be true in order for this problem to occur:

1. Two LD instructions are loaded in parallel.
2. Both LDs are to the same cache line (upper 26 address bits are the same).
3. The LD using T1 is predicated and the predicate is false.
4. The LD using T2 is either not predicated or is predicated and the predicate is true.
5. The cache line is absent from the cache.
6. The two other lines in the same L1D set are valid.
7. The LRU cache line in the set is dirty.

Results:

- L1D informs L2 to expect a victim for the affected set.
- L2 stalls DMAs with addresses that correspond to that set. Note: DMA includes accesses from IDMA, EDMA, and any external masters, such as other CPUs.
- L1D processes the true-predicated request correctly.
- L1D does not send the indicated victim.

Impacts:

If the load instruction reads a cacheable location:

- The updated data in the LRU line gets dropped.
- DMA accesses, whose addresses match the affected set, hang.

If the load instruction reads a non-cacheable location:

- L1D retains the updated data from the LRU line.
- DMA reads may see stale data if the LRU line's address is in L2 memory.

Workaround: Use Code Gen patch 6.0.3, Code Gen 6.0.4, or later version (available on update advisor) to recompile your source code and avoid this issue. Libraries supplied by TI have been re-released using the 6.0.3 compiler patch. Customer-generated libraries from TI's third-party supplier may also need to be recompiled. For existing object code and libraries, an available Perl script can determine locations of parallel predicated loads that may fail. The script is available at the same update advisor location as the Code Gen patch.

Advisory 16   *CPU: Back-to-Back SPLOOPs With Interrupts Can Cause Incorrect Operation on C64x+ CPU*

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

Details: Back-to-back software pipeline loops (SPLOOPs) with interrupts can cause incorrect operation on the C64x+ CPU. This issue occurs when the first SPLOOP is interrupted and there are less than two execute packets between the SPKERNEL of the first SPLOOP block (SPKERNEL instruction marks the end of the first SPLOOP block) and the SPLOOP instruction of the second SPLOOP block (SPLOOP instruction marks the beginning of the second SPLOOP block). The first SPLOOP block terminates abruptly (i.e., without completing the loop, even though the termination condition is false). The failure mechanism can be seen as a hang or by the first SPLOOP block draining for the interrupt and starting the second SPLOOP block without taking the interrupt or returning to complete the first SPLOOP block.

Workaround: The C6000 compiler release v6.0.6, and above, detects this problem. If there are fewer than two execute packets between the SPKERNEL and SPLOOP instructions, the compiler adds the appropriate number of NOP instructions following the SPKERNEL instruction. For example:

```
        SPKERNEL 0, 0
        NOP 1 ; SDSCM00012367 HW bug workaround
        MVK .L1 0x1,A0
[ A0] SPLOOPW 3 ;12
        NOP 1
```

The assembler detects sequences that could potentially trigger this issue and produces a remark. For example:

```
"neg_test.asm", REMARK at line 21 [R5001] SDSCM00012367 potentially triggered by
this execute packet sequence. SLOOP must be at least 2 EPs away from previous
SPKERNEL for safe interrupt behavior.
```

Note: The assembler tool, asm6x.exe, can be used to determine if a previous version of the compiler generated code that could potentially be affected by this silicon issue. The assembler can also be used on assembly source code to see if the source could be affected by this issue. Replace the old version of asm6x.exe with the v6.0.6 asm6x.exe in your current build setup and recompile or reassemble.

Advisory 17	***CPU: C64x+ CPU Incorrectly Generates False Exceptions for Multiple Writes***

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

Details: The C64x+ CPU may generate an incorrect resource conflict exception when taking an interrupt. This only affects applications that run with exceptions enabled. Applications enable exceptions by writing 1 to the GEE bit in the task state register (TSR). Applications that do not enable exceptions are not affected by this errata. The CPU generates this incorrect exception in the following scenario:

1. The CPU begins draining the pipeline as part of an interrupt context switch. During this time, the CPU annuls instructions in the pipeline that have not yet reached the E1 pipeline phase while it drains the pipeline.

2. The first annulled execute packet (resident in the DC pipeline stage at the time draining begins) writes to one or more predicate registers. Because it is annulled, the writes do not occur.

3. The second annulled execute packet (resident in the DP pipeline stage at the time draining begins) has a predicated single cycle instruction that uses a predicate written by the execute packet described in item 2. Because it is annulled, the write does not occur.

4. The value held in the predicate register would cause the instruction in the second annulled execute packet to write to some register in the same cycle as another instruction if it were not annulled. The conflicting writes would not happen if the first execute packet had not been annulled. The exception is not a valid exception. If the CPU executed instructions, described in items 2 and 3 above, rather than annulling them while draining the pipeline for an interrupt, the execute packet in item 2 would set the predicate(s) such that the writes in the subsequent execute packet do not conflict.

Examples of sequences that generate the incorrect exception are:

```
        ZERO A0
        ZERO B0
--------------------> interrupt occurs
        MVK 1, A0 ;(1st annulled EPKT)
        [!A0] MVK 2, A1 ;(2nd annulled EPKT) \_ Appears both MVKs write A1,
        ||[!B0] MVK 3, A1 ;(2nd annulled EPKT) / triggers invalid exception.
        ...
        ZERO A0
        [!A0] LDW *A4, A5
        NOP
        NOP
--------------------> interrupt occurs
        MVK 1, A0 ;(1st annulled EPKT)
        [!A0] MVK 2, A5 ;(2nd annulled EPKT) LDW writes A5 this cycle
        ...
        ZERO A0
        [!A0] DOTP2 A3, A4, A5
        NOP
------------------> interrupt occurs
        MVK 1, A0 ;(1st annulled EPKT)
        [!A0] MVK 2, A5 ;(2nd annulled EPKT) DOTP2 writes A5 this cycle
```

Workaround: The CPU only recognizes the incorrect exception while it drains the pipeline for an interrupt. As a result, the CPU begins exception processing upon reaching the interrupt handler. The NMI return pointer register (NRP) and the NMI task state register (NTSR) reflect the state of the machine upon arriving at the interrupt handler. Therefore, to identify the incorrect resource conflict exception in the software, verify the following conditions at the beginning of the exception handler prior to normal exception processing:

1. The exception occurred during an interrupt context switch.

    (a) In the NTSR register, verify that INT=1, SPLX=0, IB=0, CXM=00.
    (b) Verify that the NRP register points to an interrupt service fetch packet. That is, (NRP & 0xFFFF FE1F) == (ISTP & 0xFFFF FE1F).

2. The exception is a resource conflict exception. In IERR, verify that RCX == 1 and all other IERR bits == 0.

3. The exception is an internal exception. In EFR, verify that IXF == 1 and all other EFR bits == 0.

Upon matching the above conditions, suppress the exception as follows:

1. Clear the EFR.IXF bit by writing 2 to the ECR bit.

2. Resume the interrupt handler by branching to the NRP register. The above workaround identifies and suppresses all cases of the incorrect resource conflict exception. It resumes normal program execution when the incorrect exception occurs, and has minimal impact on the execution time of program code. The interrupted code sequence runs as expected when the interrupt handler returns. The workaround also suppresses a particular valid exception case that is indistinguishable from the incorrect case. Specifically, the code suppresses the exception generated by two instructions with different delay slots (e.g., LDW and DOTP2) writing to the same register in the same cycle, where the conflicting writes occur during the interrupt context switch.

An example of a sequence with incorrectly suppressed exception is:

```
LDW *A0, A1
DOTP2 A3, A2, A1
NOP
-----------------> interrupt occurs
NOP
NOP ; Both LDW and DOTP2 write to A1 this cycle
```

The workaround will not suppress these valid resource conflict exceptions if the multiple writes occur outside an interrupt context switch. That is, the workaround will not suppress the exception generated by the code above when it executes without an interfering interrupt.

For more details, see the following sections in the *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732):

• *Interrupt Service Table Pointer Register (ISTP)* describes the ISTP control register.

• *Nonmaskable Interrupt (NMI) Return Pointer Register (NRP)* describes the NRP control register.

• *TMS320C64x+ DSP Control Register File Extensions* describes the ECR, EFR, IERR, TSR, and NTSR control registers.

• *Pipeline* describes the overall operation of the C64x+ pipeline, including the behavior of the E1, DC and DP pipeline phases.

• *Actions Taken During Nonreset Interrupt Processing* describes the operation of the C64x+ pipeline during interrupt processing, including how it annuls instructions.

• *C64x+ CPU Exceptions* describes exception processing.

Advisory 20   *DDR2 EMIF Buffers Not Totally Compensated by Default*

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

Details: The output buffers on the DDR2 EMIF contain dynamic impedance compensation circuitry to maintain a constant output impedance across temperature, voltage, and silicon process variation. This impedance compensation circuitry must configure each individual output buffer. The output buffer compensation for each DDR2 output buffer is not complete until both a 1-to-0 and 0-to-1 transition has occurred on that output.

Until this compensation occurs, the output drive strength is probably less than ideal. The DDR2 EMIF cycles that occur before dynamic compensation is complete may fail. Since the mode register (MR) write cycles are the first cycles initiated by the DDR2 EMIF after a reset, these cycles are at risk. Similarly, after EMIF configuration, the first writes to DDR2 memory are also at risk.

This issue has not been seen in the field. It has only been observed on test fixtures at TI. Therefore, it appears to have a very low probability of affecting existing designs. The recommended topologies that only have one or two loads per DDR2 EMIF without VTT termination appear to be resilient to this condition. The possibility of failure can only be eliminated if the software workaround described below is implemented.

Some system start-up sequences improve the probability of robust operation, such as:

- Incomplete compensation may cause mode register (MR) writes to fail. This could result in DDR2 performance lower than expected or complete failure. The risk of this occurring is reduced through multiple MR write operations since compensation of most address and control output buffers and both clock output buffers are completed before the final MR write. Most DDR2 EMIF configuration sequences, including the one implemented in the CSL, result in multiple MR write operations. MR write cycles are also designed to complete on the slowest possible DDR2 memory devices. This is another reason these cycles have a high probability of success.

- Incomplete compensation may cause initial DDR2 memory writes to fail. This could cause the DSP to execute incorrectly if these initial writes are code or critical data. Many system implementations use a secondary bootloader to load the full binary image. The secondary bootloader is then discarded after boot completion. Therefore, any invalid writes would occur in the bootloader and the full application code is loaded after the DDR2 EMIF output buffers have been activated many times. (This is not a full guarantee of complete compensation but the probability is high that all of the output buffers are fully compensated by the time the secondary bootloader is written.)

- A better guarantee that this compensation has no latent impact is validation of the full binary image through some type of code checksum at the end of the boot process. If the code is verified in this way, the system is guaranteed to be robust.

Workaround: This workaround has to be executed every time the DDR2 EMIF is initialized. Since it should occur before valid mode register writes can be completed, the EMIF configuration has to be repeated after the output buffers are fully compensated. The sequence of steps listed below completes the dynamic compensation for all of the DDR2 EMIF output buffers. The CSL API function CSL_ddr2HwSetup is called during the normal bring-up process to trigger MR writes. Therefore, the workaround code can be put before the API function call.

Sample code:

```
/* Define the following variables. */
Uint32    tempData0, tempData1;
/* Define the following pointers to compensate the address buffers. */
Uint32     *pDdr2Data_temp0 = (Uint32 *) 0xEAAAAAA8;
Uint32     *pDdr2Data_temp1 = (Uint32 *) 0xE5555554;
/* Use 0xF5555554 on systems using 512MB of memory. */

/**********************************************************
The following code needs to be executed at the beginning of every DDR2 EMIF
initialization.
**********************************************************/

/* Enable self-refresh mode and set an appropriate REFRESH_RATE to guarantee
200us delay before CKE goes high. REFRESH_RATE value has to be calculated based
on DDR2 clock being used. */
hDdr2->regs->SDRFC = 0x80001388;

/* Disable self-refresh mode and set an appropriate REFRESH_RATE to have a
correct refresh cycle. REFRESH_RATE value has to be calculated based on DDR2
clock being used. */
hDdr2->regs->SDRFC = 0x00000753;

/*Write and read the first location with a 0xAAAAAAAA pattern.*/
tempData0 = 0xAAAAAAAA;
*pDdr2Data_temp0 = tempData0;     /* DDR2 memory write */
tempData1 = *pDdr2Data_temp0;     /* DDR2 memory read */

/* Perform two more writes with a 0x55555555 and 0xAAAAAAAA pattern to complete
the compensation cycle. */
tempData0 = 0x55555555;
*pDdr2Data_temp1 = tempData0;     /* DDR2 memory write */
tempData0 = 0xAAAAAAAA;
*pDdr2Data_temp0=tempData0;        /* DDR2 memory write */
```

Advisory 21   *SRIO Port 0 Reset Affects Other Ports*

Revision(s) Affected: 2.1, 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 22   *SRIO OUTBOUND_ACKID Field Not Read Correctly*

Revision(s) Affected: 2.1, 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 25   ***DMA Corruption of External Data Buffer***

Revision(s) Affected: 2.1, 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 TCI6486 device, see Table 8.

Table 8. TCI6486 UMAP1 Allocation

UMAP1	ADDRESS RANGE	AFFECTED
N/SMC	0x00200000 - 0x002BFFFF	Yes

- 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 TCI6486 device, 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

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

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 2.
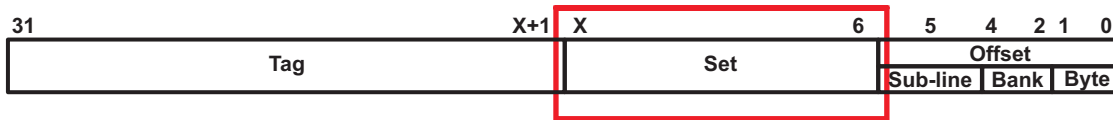
| 31 | X+1 | X | 6 | 5 | 4 | 2 | 1 | 0 |

| Tag | Set | Offset |
| | | Sub-line | Bank | Byte |

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

Figure 2. 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 3 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.
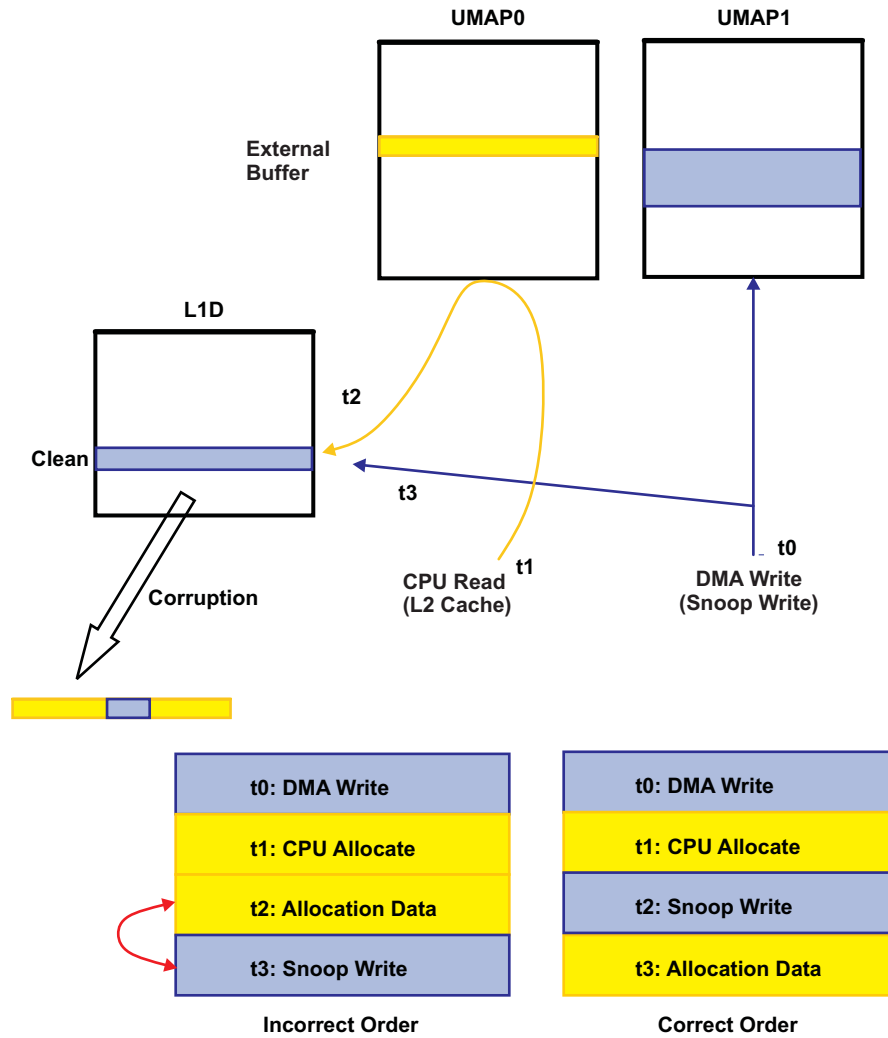
Figure 3. Cache Line Operations Flow

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

We propose 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

```
;; =================================================================== ;;

        .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:** 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.

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 (c) 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 28	***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 29   ***DMA Corruption of L1D$ Allocation***

Revision(s) Affected: 2.1

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 4.
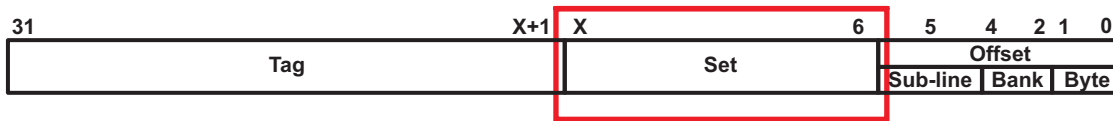
31	X+1	X		6	5	4	2	1	0
Tag		Set			Offset				
					Sub-line	Bank		Byte	

The value X is determined by how large the L1D cache is in the particular application (see Table 10).

Figure 4. L1D Cache Address Mapping

Table 10. 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.
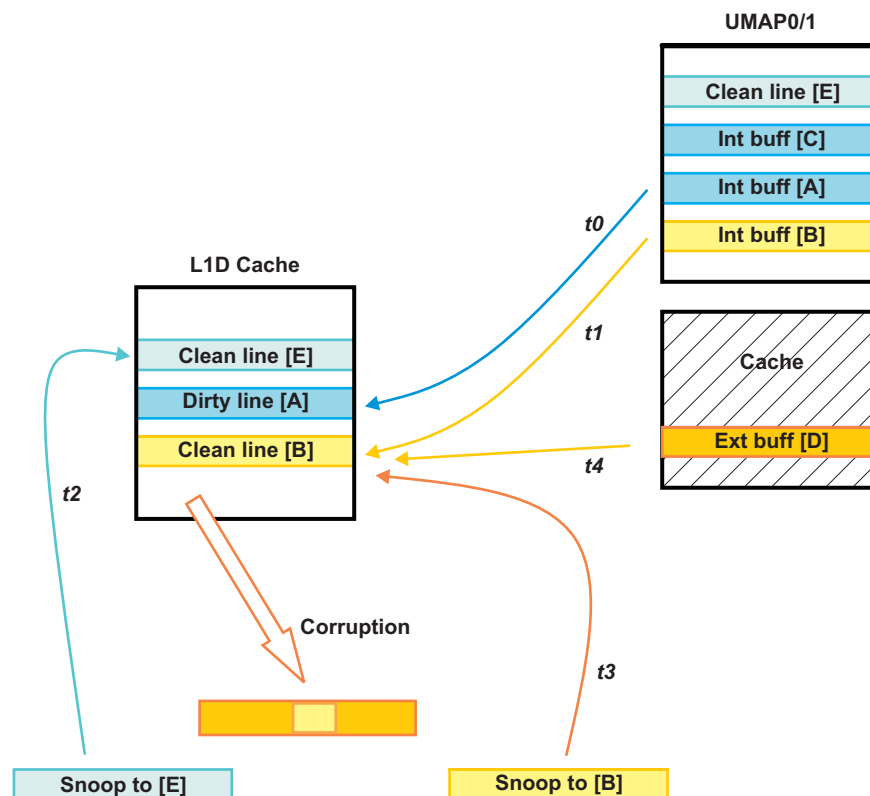
Figure 5 shows the sequence of events.

Figure 5. 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 30	***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 31   *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 6. 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.
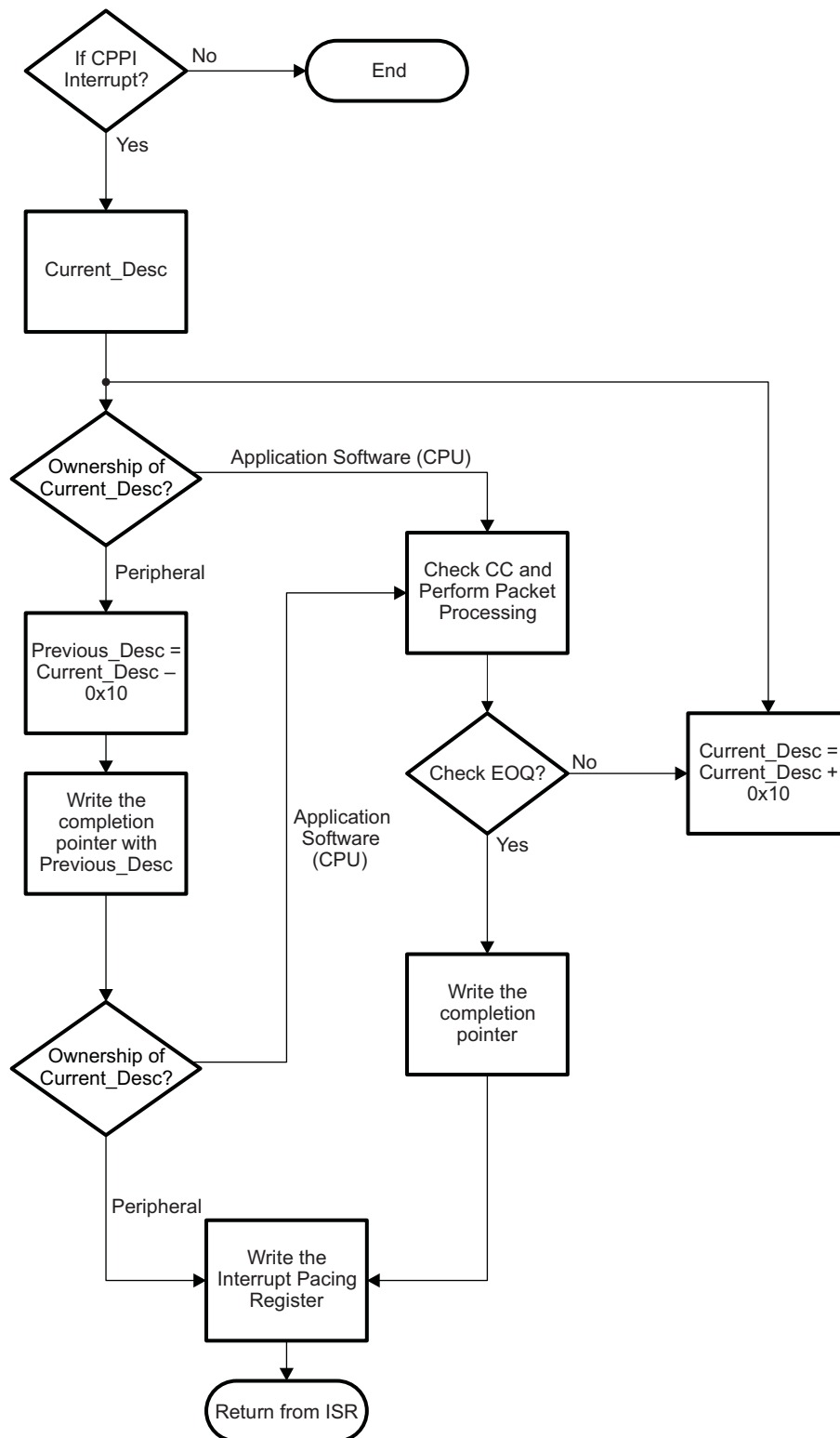8. Write the interrupt pacing register to enable the next interrupt.

Figure 6. ISR Workaround Flowchart

Copyright © 2009–2011, Texas Instruments Incorporated

Advisory 32   *Serial RapidIO Internal Digital Loopback is Not Always Stable*

Revision(s) Affected: 2.1 and earlier

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 1.2 Usage Notes and Known Design Exceptions to Functional Specifications

3.1 Silicon Revision 1.2 Usage Notes

Silicon revision 1.2 applicable usage notes have been found on a later silicon revision; for more detail, see Section 2.1, *Silicon Revision 2.1 Usage Notes*, of this document.

3.2 Silicon Revision 1.2 Known Design Exceptions to Functional Specifications

Table 11 lists the silicon revision 1.2 known design exceptions to functional specifications. Advisories are numbered in the order in which they were added to this document. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. If the design exceptions are no longer applicable or if the information has been documented elsewhere, those advisories are removed. 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 to Section 2.2, *Silicon Revision 2.1 Known Design Exceptions to Functional Specifications*, of this document.

Table 11. Silicon Revision 1.2 Advisory List

Advisory 6   *DSP SDMA/IDMA: Unexpected Stalling of SDMA/IDMA Access to L2 SRAM*

Revision(s) Affected: 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 that are allowed to access these
memories are GEM megamodules, DMA controllers, TSIPs, EMAC, UTOPIA, HPI,
SRIO, and SRIO wrapper. 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 cache, 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 or 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 7) and SDMA/IDMA paths (the orange lines
in Figure 7) 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 7 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
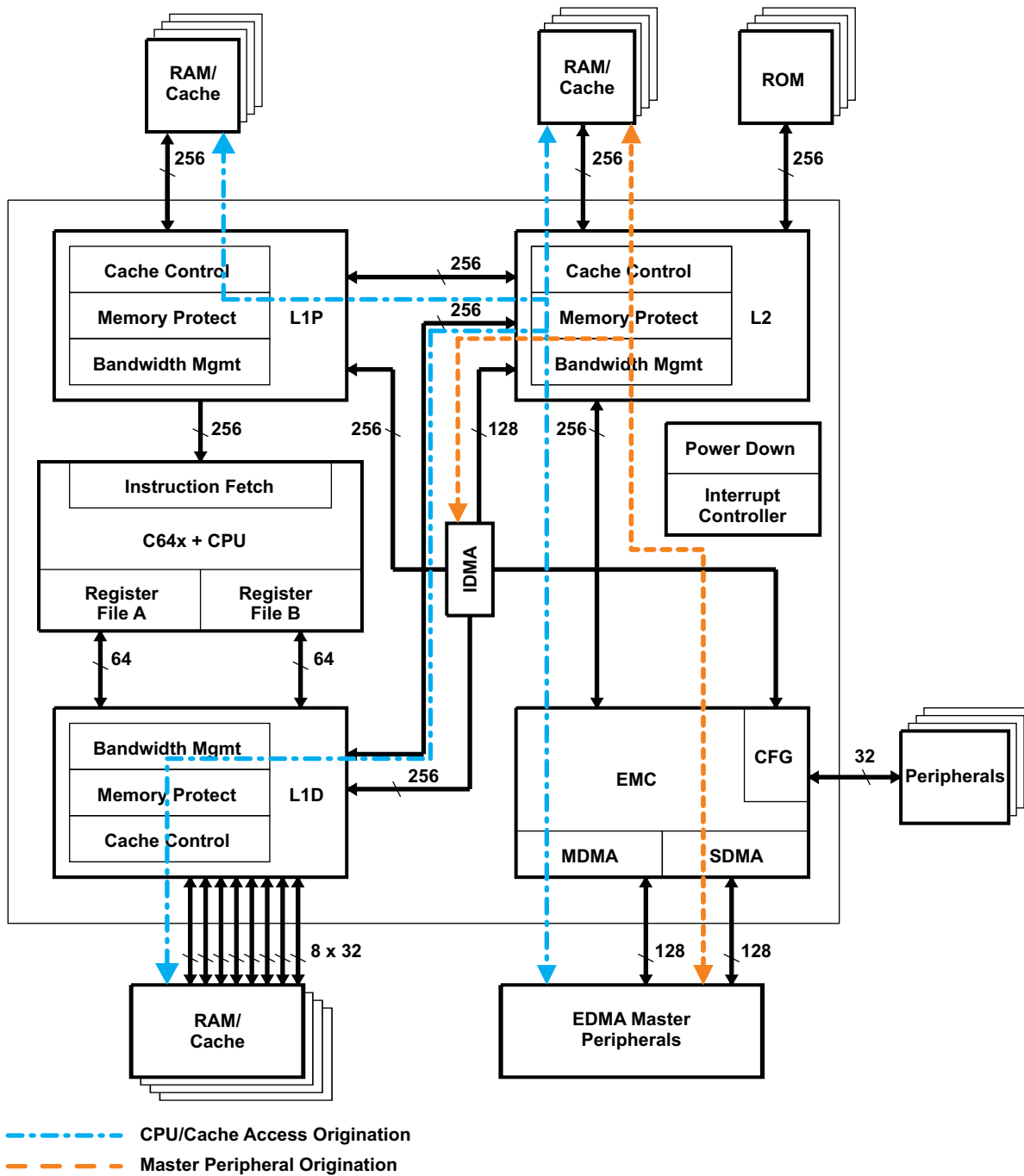controller or directly from the DSP.

Figure 7. 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 occurring include lower-than-expected peripheral throughput or loss of I/O data.

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 external memory or non-local L2, TSIP Tx/Rx data transfer or an EMAC Tx/Rx/CPPI transfer or UTOPIA Tx/Rx transfers, SRIO Tx/RX/CPPI message transfers, SRIO direct IO accesses, or HPI accesses). 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 writing 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: 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 consequence of the IDMA/SDMA stall issue is the C64x+ MDMA-SDMA deadlock that requires a device reset or power cycle 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 to slaveX
- Any one of the following (see Figure 8):
  - 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 shared cross-connect bridge that is shared with either TC0 or TC1 and GEM MDMA traffic from two other GEMs. 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 cross-connect bridge, then the bridge may fill due to the stalled SDMA before the read command can propagate through the bridge and complete. The C6472 device has two cross-connect bridges (Xconn1 and Xconn2). GEMs 0, 1, 2, and TC0 use cross-connect bridge 1. GEMs 3, 4, 5, and TC1 use cross-connect bridge 2. In summary, a deadlock can occur if the following is true:

- GEMx issues a read to any of other GEM's L1D or L2 SRAM through cross-connect bridge 1 or 2.
- Any of the other GEMs or TCx on the same cross-connect bridge issue commands to GEMx L2.
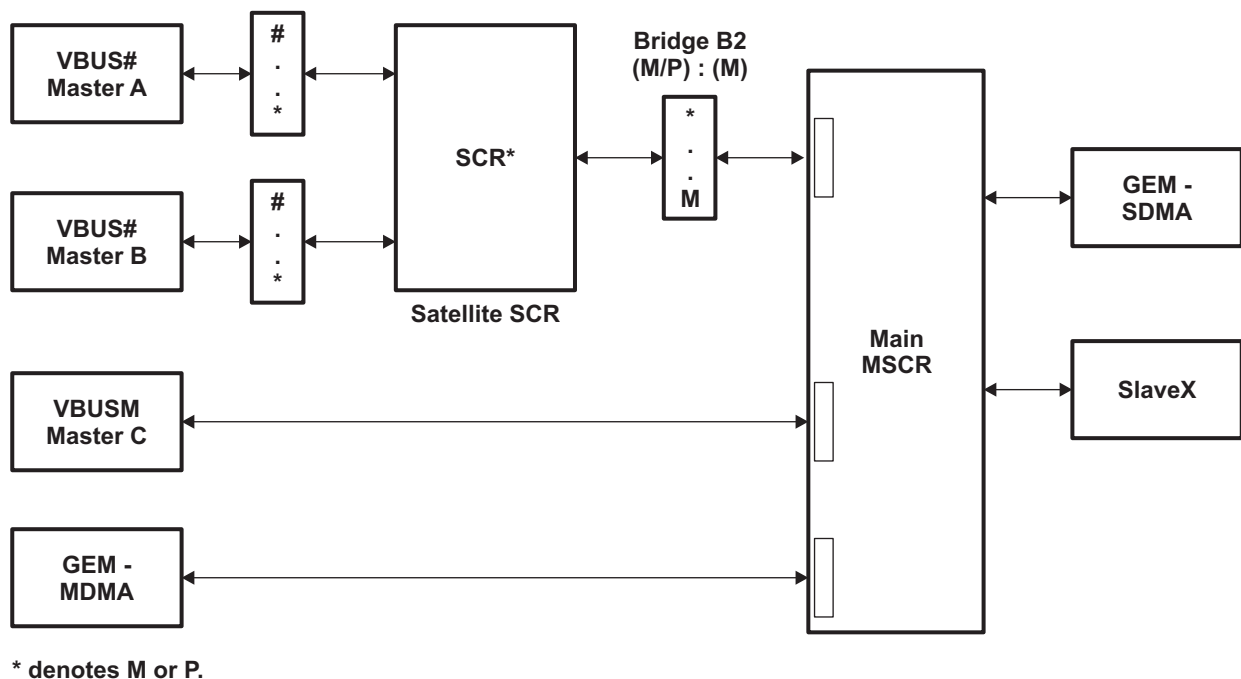

Figure 8. Data Pipelined SCR

Workarounds:　　**Method 1**

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 C6472 multi-core 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 2

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.

Dataflow Requirements

To avoid the issue due to a C64x+ deadlock, workarounds depend on the CPU/peripheral bus master that is accessing the GEM internal or external memory:

CPU/PERIPHERAL BUS MASTER	WORKAROUND
GEM	GEMs should not write to the memory of any other GEM. GEMs must not directly read from the memory of any other GEMs without providing the L1D cache disable workaround, mentioned in Method 1, 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). 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 the DDR.
TSIP0,1,2	All TSIPs (0, 1, 2) should either write to the GEM's memory or the DDR, but not both.
SRIO, SRIO CPPI	SRIO should transfer payload data to only the GEM memories or to the DDR2 SDRAM, but not both. This includes any direct I/O writes as well as any inbound receive messaging transfers. 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.
EMAC0, EMAC1, UTOPIA, HPI	All four masters should write to the GEM's memory or the DDR, but not both. This includes buffers and buffer descriptors of EMAC and any writes issued by the host. EMAC CPPI descriptors should be placed wholly in local wrapper memory, any combination of wrapper and L2 memory, or any combination of wrapper and DDR2 SDRAM.

Advisory 7   *Potential SerDes Clocking Issue*

Revision(s) Affected: 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).

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.

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

Advisory 8   *Potential Insertion or Deletion of 2 Bits in SerDes Data Stream*

Revision(s) Affected: 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 initializes 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.

Workaround: SRIO has an auto-recovery in silicon rev. 1.0 and 1.1. 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.

Advisory 10	*Atomic Operations Fail to Complete*

Revision(s) Affected: 1.2, 1.1, 1.0

Details:

In the C6472 device atomic access monitors are located only in the shared-memory controller. These monitors assure that only the correct sequence of accesses are successful. Successful completion of atomic operations requires the successful completion of a sequence of load-linked (LL), store-linked (SL), and commit-linked (CMTL) instructions to a single address by a single CPU. The atomic operation instructions were intended to temporarily freeze the cache for execution of those instructions, in all respects. However, the implicit cache freezing functionality is incomplete due to an internal exception. Therefore, for the accesses to be successful, the L1D cache must be explicitly temporarily frozen around the execution of these instructions to ensure that the accesses generated by these instructions are made all the way to the shared-memory controller. Since the L1D memory controller (DMC) does not completely disable the cache functionality for these instructions resulting in a failure to complete, explicit disables and, possibly, re-enables of the L1D cache around these instructions must be provided by the software.

Workaround:

Reference implementations of *atomic add* and *atomic exchange* that incorporate workarounds for atomic operations and L1D freeze mode are provided below. Since the L1D cache lines must be explicitly frozen, the memory objects of the atomic access instructions should be the only thing stored in a 64B L1D cache line (i.e., normal reads and writes should not be performed to the remaining locations in the same L1D cache line).

```
;Atomically add A4 to the value pointed to by B4. Returns the sum.
;This is a C-callable function as written.
; int atomic_add(int value, volatile int *sema)
;
      .asg 0x01840044, L1DCC
      .global _atomic_add
_atomic_add:

      MV A4, B5 ; Copy value to exchange to B side.
||    MVKL L1DCC, A5 ; \_ L1D cache control for freezing
      MVKH L1DCC, A5 ; / and unfreezing L1D.
||    MVK 1, A0 ; Value to put into L1DCC.OPER for freeze.

loop:
      STW .D1T1 A0, *A5 ; Freeze L1D.
||    DINT ; Disable interrupts.
      LDW .D1T1 *A5, A2 ; Get previous freeze. Stall until frozen.

[!A0] MVK .D2 0, B1 ; (block EDI)
      LL .D2 *B4, B6 ; Establish monitor and get old value.
[!A0] MVK .D2 0, B1 ; (block EDI)
      NOP 3 ; wait for LL to complete.
      ADD .D2 B5, B6, B7 ; Add new value to old *and* block EDI.
      SL .D2 B7, *B4 ; Store new value to monitor.
[!A0] MVK .D2 0, B1 ; (block EDI)
      CMTL.D2 *B4, B0 ; Attempt to commit.
[!A0] MVK .D2 0, B1 ; (block EDI)

      SHRU A2, 16, A2 ; Shift L1DCC.POPER down to L1DCC.OPER.
      STW .D1T1 A2, *A5 ; Restore previous frozen/non-frozen state.
      RINT ; Restore interrupts

;; ==== Interrupt may occur here prior to issuing branch.

[!B0] BNOP.S2 loop, 0 ; Loop if it didn't commit.
||    MV B7, A4 ; Copy sum to return register.
[ B0] BNOP.S2 B3, 5 ; Return if it did commit.
;; ==== Interrupt may occur here prior to reaching branch target.
;; ==== Branch occurs.

; Atomically exchange the value pointed to by B4 with the value in A4.
; This is a C-callable function as written.
```

```
;
; int atomic_exhange(int value, volatile int *sema)
;
        .asg 0x01840044, L1DCC
        .global _atomic_exchange
_atomic_exchange:

        MV A4, B5 ; Copy value to exchange to B side.
||      MVKL L1DCC, A5 ; \_ L1D cache control for freezing
        MVKH L1DCC, A5 ; / and unfreezing L1D.
||      MVK 1, A0 ; Value to put into L1DCC.OPER for freeze.

loop:
        STW .D1T1 A0, *A5 ; Freeze L1D.
||      DINT ; Disable interrupts.
        LDW .D1T1 *A5, A2 ; Get previous freeze. Stall until frozen.

[!A0] MVK .D2 0, B1 ; (block EDI)
        LL .D2 *B4, B6 ; Establish monitor and get old value.
[!A0] MVK .D2 0, B1 ; (block EDI)
        SL .D2 B5, *B4 ; Store new value to monitor.
[!A0] MVK .D2 0, B1 ; (block EDI)
        CMTL.D2 *B4, B0 ; Attempt to commit.
[!A0] MVK .D2 0, B1 ; (block EDI)

        SHRU A2, 16, A2 ; Shift L1DCC.POPER down to L1DCC.OPER.
        STW .D1T1 A2, *A5 ; Restore previous frozen/non-frozen state.
        RINT ; Restore interrupts.

;; ==== Interrupt may occur here prior to issuing branch.

[!B0] BNOP.S2 loop, 0 ; Loop if it didn't commit.
||    MV B6, A4 ; Copy exchanged value to return register.
[ B0] BNOP.S2 B3, 5 ; Return if it did commit.
;; ==== Interrupt may occur here prior to reaching branch target.
;; ==== Branch occurs.
```

Advisory 12   *PMC: Local Reset (Ireset) Followed By Block Invalidate Hangs*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: The PMC never goes into idle when Ireset is followed by a programmed block invalidate without sufficient delay. The L1PINV command issued to the PMC is not acknowledged until the CFG request is read in. When combined with an Ireset, the PMC treats it as a new L1PINV command every cycle and the invalidation counter never increments. The result is a deadlock.

Workaround: Ensure that 2Kb CPU cycles have elapsed after beginning execution at the reset vector before an L1PINV is issued. The minimum number of CPU cycles is decided by the time duration for the global cache invalidation to finish before a new L1PINV starts.

Note: Advisories 12 and 13 must be used together.

Advisory 13   *PMC: L1P Cache Not Invalidated During Ireset*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: The PMC is supposed to perform cache invalidation on Ireset. However, due to an internal exception, the PMC may drop cache invalidate on Ireset and the remaining addresses are skipped from invalidation.

Workaround: The code that is located at the Ireset start address, beginning of the local L2 memory by default, must perform a complete invalidation of the L1P before proceeding.

Note: Advisories 12 and 13 must be used together.

Advisory 14   *UMC: L2MPFAR Fails to Log CPU Protection Faults Under Certain Conditions*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: When a CPU memory protection fault on a UMAP0 access occurs in the same cycle as a DMA memory protection fault on a UMAP1 access, the DMA fault information, rather than the CPU fault information, is logged in L2MPFAR and L2MPFSR.

Workaround: There is no workaround for this advisory.

Advisory 18　　　　*PrivID For Non-CPU Masters Is Same as GEM0 CPU*

Revision(s) Affected:　1.2, 1.1, 1.0

Details:　The GEM0 CPU and all non-EDMA system masters on the device are assigned the same privilege ID. (The EDMA inherits the privilege ID from the programming GEM.) This requires the software to take corrective actions to differentiate memory accesses from the GEM0 CPU and the non-CPU peripheral masters when a memory access violation occurs. **Note:** Within GEM0, the CPU generated accesses are identified as local, although IDMA accesses are recognized as PrivID=0 accesses. This provides some distinction between true CPU accesses and non-EDMA system masters for GEM0.

Workaround:　A partial workaround is to keep the non-GEM traffic always as *user* access and the GEM0 traffic always as *supervisor* access. In that case, AID0 supervisor transactions are known to be GEM0 CPU and AID0 user transactions are known to be non-CPU and non-EDMA peripheral masters. **Note:** If the HPI/SRIO are configured as supervisors for host accesses to on-chip memory and certain MMRs, then it is not possible to distinguish between GEM0 supervisor traffic and HPI/SRIO traffic.

Note: This issue was fixed in later silicon revisions. GEM0 CPU privilege ID was changed from 0 to 6.

PID	SILICON REVISION ≤1.2	SILICON REVISION >1.2
0	Non-EDMA System Masters; GEM0 CPU 0	Non-EDMA System Masters
1	GEM1 CPU	GEM1 CPU
2	GEM2 CPU	GEM2 CPU
3	GEM3 CPU	GEM3 CPU
4	GEM4 CPU	GEM4 CPU
5	GEM5 CPU	GEM5 CPU
6	-	GEM0 CPU

Advisory 19   *UTOPIA Lock-Up Issue*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: A PDMA issue could cause the UTOPIA receive cell buffer in memory to become corrupted and UTOPIA and PDMA to stop transmitting and receiving cells. UTOPIA and PDMA must be reset through the PSC and reconfigured to restart operation.

Workaround(s):   A partial workaround is to perform a software recovery. A lock-up detection and UTOPIA reset routine on the C6472 device can recover the lockup. Lock-up detection can be performed by a polling routine running at 1-kHz frequency. When the lockup occurs, no calls are lost; no calls must be torn down and setup again. The detection and recovery procedure is:

1. Core y detects a cell with a bad header and signals Core A through the IPC.
2. Core A disables the UTOPIA, closes all DMA channels (on all cores), and executes a reset of UTOPIA through the PSC.
3. Core A signals all other cores to perform a local UTOPIA restart through the IPC.
4. Core A waits for all other cores to acknowledge the restart.
5. Core A performs a global UTOPIA enable.

In addition to the software recovery, the following three optional workarounds could reduce the risk of UTOPIA lock-up occurrence.

1. One cell per block for transmits.
   (a) One transmit block with multi-cells could block a PDMA receive channel from winning arbitration to copy data from the UTOPIA receive FIFO to the PDMA receive channel buffer in time to avoid a *super-sized* cell. The maximum block time depends on the number of cells in the block due to the PDMA arbitration mechanism.
   (b) One-cell transmit reduces the time until the PDMA re-arbitrates between the receive and transmit channels. The re-arbitration enables the blocked receive channel to win the grant.
   (c) The implementation of one-cell transmit is purely software based and has minimal impact on UTOPIA throughput.
2. Any two cells that arrive in the UTOPIA are sent to different PDMA channels. This cuts the available bandwidth in half, but it prevents the problem.
3. Set the UCLK at the lowest possible speed without impacting the system performance. The lower the UCLK speed, the higher the tolerance for stalls. The chance for lockup is significantly reduced.

Advisory 23   *DMA Access to L2 SRAM May Stall When the DMA Has Lower Priority Than the CPU*

Revision(s) Affected: 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 12.

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

Table 12. C6472 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)
EMAC	7	PRI_ALLOC.EMAC
HPI	7	PRI_ALLOC.HOST
UTOPIA - PDMA	1	PRI_ALLOC.UTOPIAPDMA
TSIP	7	DMACTL (TSIP 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)

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.

This issue has been fixed on a later silicon revision.

Advisory 24   ***DMA Access to L2 SRAM May Stall When the DMA and the CPU Command Priority is Equal***

Revision(s) Affected: 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 13.

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

Table 13. C6472 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)
EMAC	7	PRI_ALLOC.EMAC
HPI	7	PRI_ALLOC.HOST
UTOPIA - PDMA	1	PRI_ALLOC.UTOPIAPDMA
TSIP	7	DMACTL (TSIP 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)

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: Set the CPU and the DMA commands to L2 on different priorities.

Advisory 26   ***SDMA/IDMA Blocking Issue Update: L2 Victim Traffic Due To L2 Block Writeback During Any Pending CPU Request***

Revision(s) Affected: 1.2, 1.1, 1.0

Details: This advisory is an update to Advisory 6 in this document. Advisory 6 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 below are actually labeled as 1, 2a, 2b, and 2c in Advisory 6.

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, and 1.2 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 6.

No additional deadlock risk potential is created by the addition of the new condition to silicon revisions 1.0, 1.1, and 1.2 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.

Table 14. 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

Under certain conditions, L2 victim traffic due to a block writeback can block SDMA/IDMA accesses to UMAP0 during CPU requests. On the C6472 device, UMAP0 (L2 RAM0) is mapped at 0x00800000 - 0x00897FFF and UMAP1 (SL2 RAM) is mapped at 0x00200000 - 0x002BFFFF in the local memory map of each core. There are four transactions that must occur 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 or instruction fetch 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 that were already affected with the other four conditions of the SDMA/IDMA stall issue from Advisory 6, there is no additional workaround needed. If all of the deadlock avoidance steps listed in Advisory 6 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 6.

Workaround 2: Do not use L2$

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

Advisory 27   *L1P$ Miss May Block SDMA Accesses*

Revision(s) Affected: 1.2, 1.1, 1.0

Details: This advisory is an update to Advisory 6 and Advisory 26 in this document. Advisory 6 and Advisory 26 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 below are actually labeled as 1, 2a, 2b, and 2c in Advisory 6. Item 5 is described in Advisory 26.

This advisory covers one more blocking condition:

6. L1P$ miss may stall SDMA accesses.

For silicon revisions 1.0, 1.1, and 1.2 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 6 and Advisory 26.

No additional deadlock risk potential is created by the addition of the new condition to silicon revisions 1.0, 1.1, and 1.2 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.

Table 15. 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

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 C6472 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 also 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.

    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. 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. shows the timing relationship.
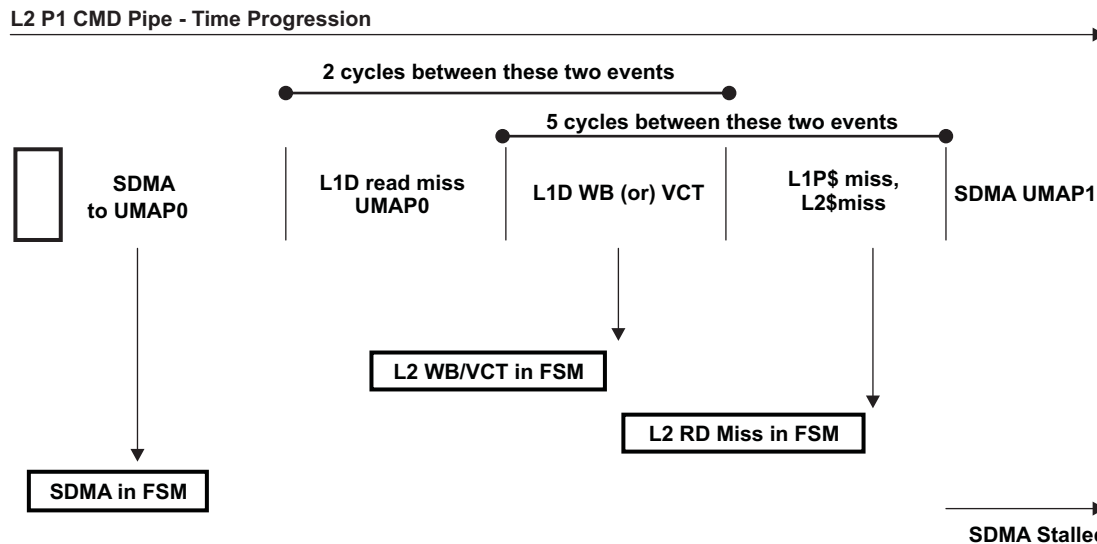
L2 P1 CMD Pipe - Time Progression

Figure 9. 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 that were already affected with the other four conditions of the SDMA/IDMA stall issue from Advisory 6, there is no additional workaround needed. If all of the deadlock avoidance steps listed in Advisory 6 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 6.

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.

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

4.1 Silicon Revision 1.1 Usage Notes

Silicon revision 1.1 applicable usage notes have been found on a later silicon revision; for more detail, see Section 2.1, *Silicon Revision 2.1 Usage Notes*, of this document.

4.2 Silicon Revision 1.1 Known Design Exceptions to Functional Specifications

Table 16 lists the silicon revision 1.1 known design exceptions to functional specifications. Advisories are numbered in the order in which they were added to this document. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. If the design exceptions are no longer applicable or if the information has been documented elsewhere, those advisories are removed. Therefore, advisory numbering may not be sequential.

All other known design exceptions to functional specifications for silicon revision 1.1 still apply and have been moved up to Section 2.2, *Silicon Revision 2.1 Known Design Exceptions to Functional Specifications*, or Section 3.2, *Silicon Revision 1.2 Known Design Exceptions to Functional Specifications*, of this document.

Table 16. Silicon Revision 1.1 Advisory List

Advisory 9   *I2C Slave Boot Does Not Work*

Revision(s) Affected: 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.

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

5.1 Silicon Revision 1.0 Usage Notes

Silicon revision 1.0 applicable usage notes have been found on a later silicon revision; for more detail, see Section 2.1, *Silicon Revision 2.1 Usage Notes*, of this document.

5.2 Silicon Revision 1.0 Known Design Exceptions to Functional Specifications

Table 17 lists the silicon revision 1.0 known design exceptions to functional specifications. Advisories are numbered in the order in which they were added to this document. If the design exceptions are still applicable, the advisories move up to the latest silicon revision section. If the design exceptions are no longer applicable or if the information has been documented elsewhere, those advisories are removed. 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 to Section 2.2, *Silicon Revision 2.1 Known Design Exceptions to Functional Specifications*, Section 3.2, *Silicon Revision 1.2 Known Design Exceptions to Functional Specifications*, or Section 4.2, *Silicon Revision 1.1 Known Design Exceptions to Functional Specifications*, of this document.

Table 17. Silicon Revision 1.0 Advisory List

Title	Page
Advisory 1 — SRIO: Packet-Forwarding NREAD Operations Larger Than 16 Bytes Fail	65
Advisory 2 — RGMII EMAC: Boot Start-Up Issue	66
Advisory 3 — DDR2 EMIF: Clock Synchronization Issue	67
Advisory 4 — TSIP: Receive Channel 4 Bitmap Corruption Issue	69
Advisory 5 — Device Configuration: HOUT is Not Generated When HPI is Disabled	72

Advisory 1   SRIO: Packet-Forwarding NREAD Operations Larger Than 16 Bytes Fail

Revision(s) Affected: 1.0

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.
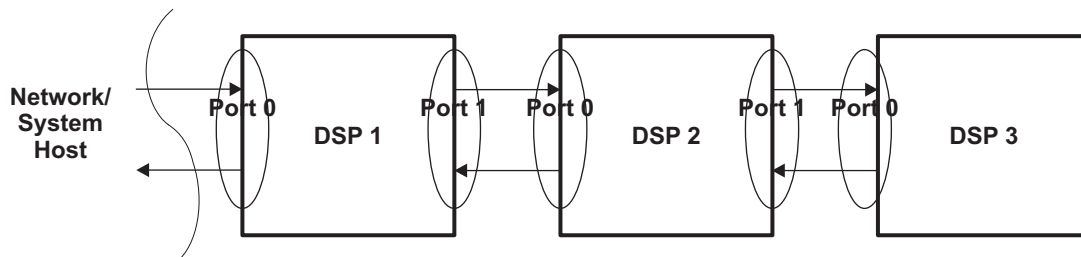
The second option is illustrated in Figure 10.

Figure 10. 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 2   *RGMII EMAC: Boot Start-Up Issue*

Revision(s) Affected: 1.0

Details: The RGMII EMAC boot mode in the C6472 will fail to transmit an Ethernet-ready frame when that boot mode is selected by the BOOTMODE[3:0] pins at device reset. In most systems, the host that is responsible for sending the boot table will not send the table until it sees the Ethernet-ready frame.

Workarounds: Three workarounds could be used to address this issue:

1. Select one of the I2C master boots. Program the selected parameter table in the I2C device to select EMAC boot as the extended boot mode type. Table 18 is a parameter table that selects the EMAC boot mode. The minimum set of parameters also includes the chip PLL multiplier, which allows this mode to automatically set the multiplier value as required by the DDR clock issue alert for use of external memory. More detailed information on programming the I2C parameter tables for additional boot configuration requirements can be found in the *TMS320C6472 Bootloader User's Guide* (literature number SPRUTBD).

Table 18. I2C Parameter Table for EMAC Boot

OFFSET	VALUE	COMMENTS
0x00	0x000A	Parameter table is 10 bytes
0x02	0xFED2	1's complement checksum
0x04	0x0105	EMAC boot
0x06	0x0000	Port 0
0x08	0x001E	PLL multiplier is x30 (Assumes CLKIN1=16.667MHz and PG1.0 silicon)

2. Have the host that is responsible for sending the boot table broadcast a small boot table with the program that is shown in the example below. This causes any C6472 devices that have not sent an Ethernet-ready frame and have not begun to receive boot table packets to restart the EMAC boot procedure and transmit the Ethernet-ready frame.

   Boot Restart Code

   ```
   warm_restart .equ  0x00100110
                mvkl  #warm_restart, b0
                mvkh  #warm_restart, b0
                bnop  b0, 5
                nop
   ```

3. If the host responsible for sending the boot table already knows the identity of all devices it is responsible to boot, then the host can begin sending the boot table packets after some customer TBD delay following a device reset. Under typical conditions, the device will be ready to receive EMAC boot packets within 2 ms following the deassertion of reset.

Advisory 3   *DDR2 EMIF: Clock Synchronization Issue*

Revision(s) Affected: 1.0

Details: The DDR2 EMIF in the C6472 device requires that specific consideration be given to the DDR2 clock with respect to the CPU clock, in order to successfully complete all writes to external memory. When independent clock sources are used for CLKIN1 (CHIP clock) and CLKIN3 (DDR2 clock), writes to the DDR2 external memory may be corrupted in certain situations.

Workaround: Provide a single clock source to both CLKIN1 and CLKIN3 and set the PLL1 multiplier to multiply CLKIN1 by x30.

The selected frequency should be 16.67 MHz and the PLL1 multiplier should be set to multiply CLKIN1 by x30. This will create a 500-MHz clock to each CPU. The DDR2 data rate, which has a fixed multiply by x20 of CLKIN3, will be 333 MHz.

The following three figures help illustrate the likely hardware changes. Figure 11 shows the clock source diagram that TI expects most customers have implemented. Figure 12 shows the requested implementation change. Figure 13 shows the workaround implementation requirement.
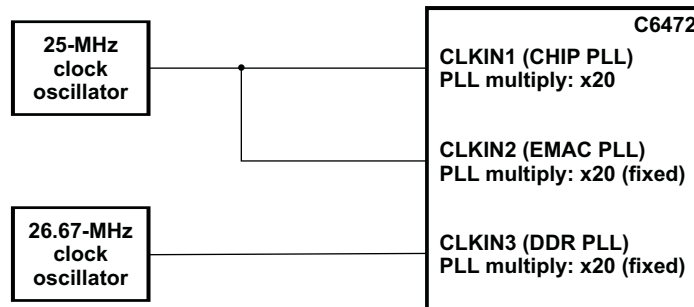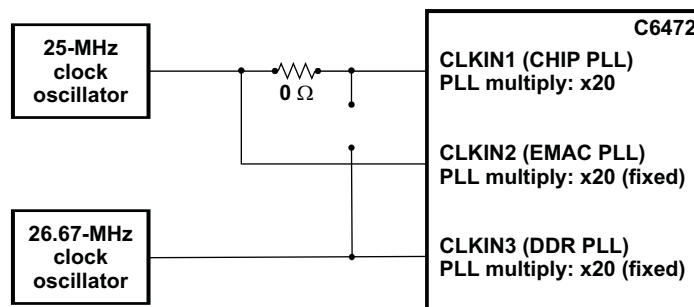

Figure 11. Expected Customer Implementation

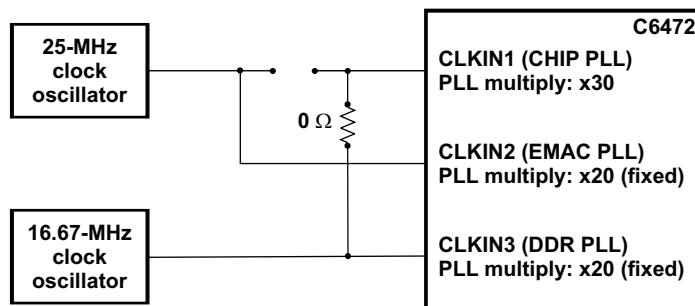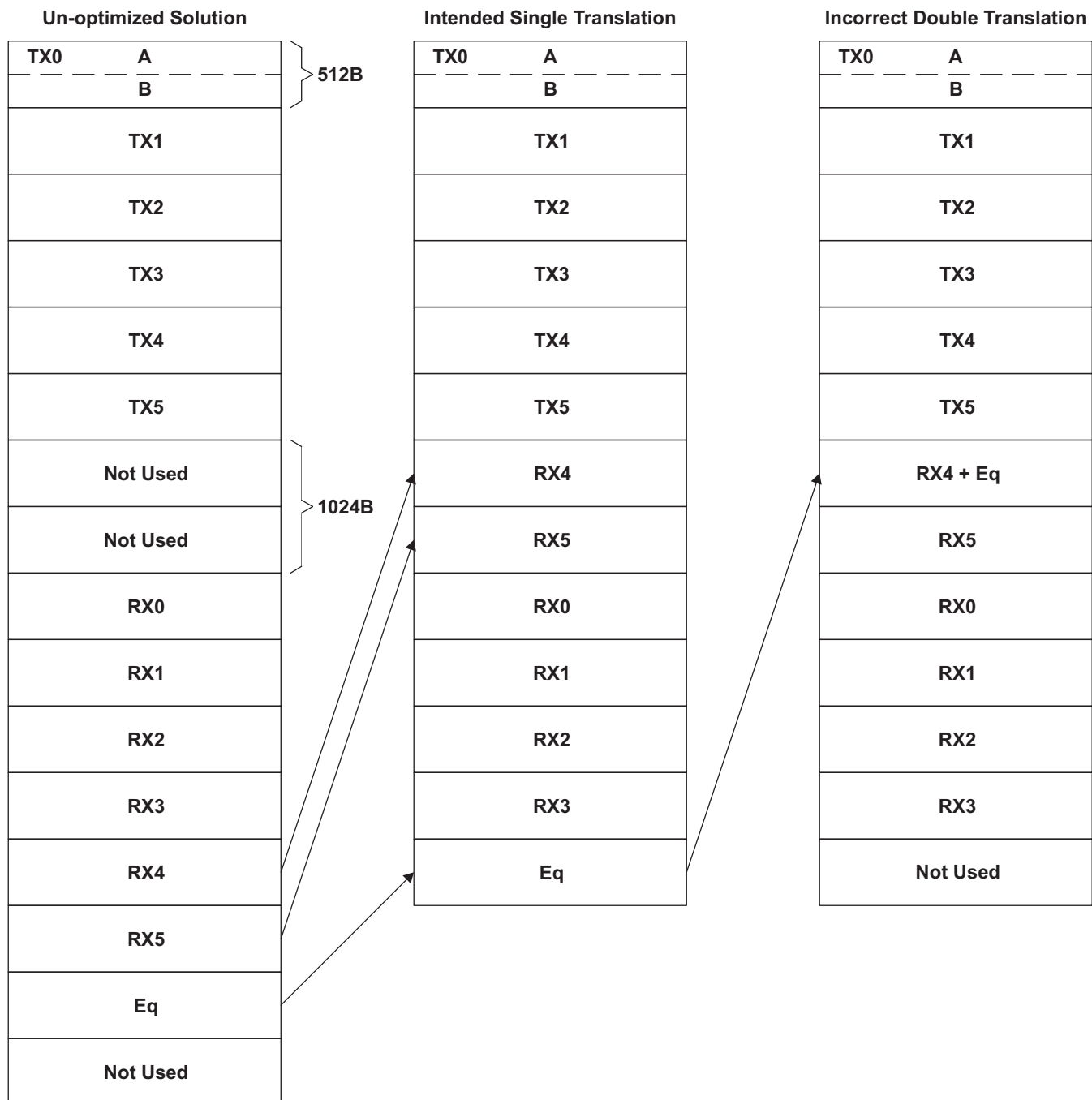**Figure 12. Requested Implementation Change**

Figure 13. Interim Workaround for Silicon Revision 1.0

Advisory 4   *TSIP: Receive Channel 4 Bitmap Corruption Issue*

Revision(s) Affected: 1.0

Details: The bitmap memory for receive channel 4 will be corrupted if an error occurs on any TSIP channel. This results in the receive data for channel 4 (associated with GEM4) being lost for frame intervals beginning with the first error until the issue is addressed and the receive channel 4 bitmap has been reinitialized and activated. Corruption is due to a double translation of the address going to the memory used for bitmaps and error queues (see Figure 14). Figure 15 shows a bitmap memory snapshot where the TSIP receive channel 4 bitmap has been corrupted.

If an error occurs during execution of the normal application, then a TSIP error interrupt is asserted for every channel that experiences an error. This error interrupt is asserted to the corresponding DSP subsystem. The DSP subsystem that receives the error interrupt must correct the error condition to avoid prolonged loss of data on its own channel as well as on channel 4. When DSP subsystem 4 receives an error it must determine whether the error is due to its own failure and, if it is, that error must be corrected first.

Un-optimized Solution	Intended Single Translation	Incorrect Double Translation
TX0 A / B }512B	TX0 A / B	TX0 A / B
TX1	TX1	TX1
TX2	TX2	TX2
TX3	TX3	TX3
TX4	TX4	TX4
TX5	TX5	TX5
Not Used }1024B	RX4	RX4 + Eq
Not Used	RX5	RX5
RX0	RX0	RX0
RX1	RX1	RX1
RX2	RX2	RX2
RX3	RX3	RX3
RX4	Eq	Not Used
RX5		
Eq		
Not Used		

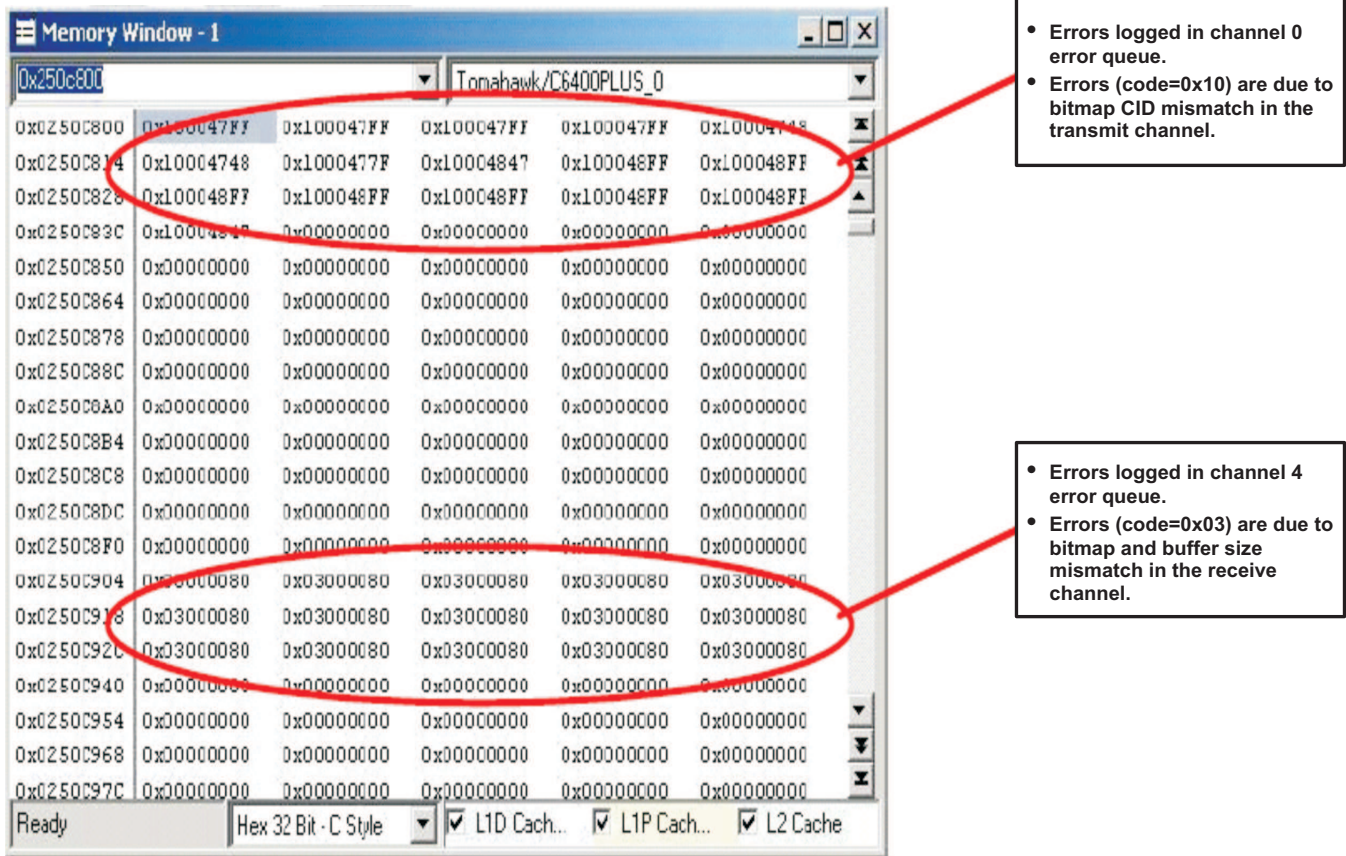**Figure 14. Bitmap Memory Address Translation**

Figure 15. Bitmap Memory Corruption Example

Workaround: The following describes the two-part Workaround

1. Fix the application code or use condition that is responsible for generating the TSIP error. This requires examination of the TSIP error registers for all the channels.
2. Re-initialize the receive channel 4 bitmap after the error condition is rectified.

Regardless of the cause of the failure, DSP subsystem 4 must re-initialize the content of its bitmap memory. The re-initialization of its bitmap memory must start with the A set.

Advisory 5   *Device Configuration: HOUT is Not Generated When HPI is Disabled*

Revision(s) Affected: 1.0

Details: HOUT is a host output signal that is pulsed by a write to IPCGR15, which is a register in the CRLF. HOUT is placed in a high-impendence state when HPI is disabled in the system. This occurs even though HOUT is functionally independent of HPI.

Workaround: HPI must remain enabled if HOUT is used, even if there is no plan to use HPI. Since the I/O buffers for HPI are on when HPI is enabled, board resistors should be used to pull all HPI I/O buffers to the 3.3-V supply rail or to ground. The preferred direction of the resistors is the same as the internal pull resistors that are present when HPI is disabled.

Appendix A Revision History

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

Scope: Applicable updates relating to the C6472 device have been incorporated.

Table 19. C6472 Revision History

SEE	ADDITIONS/MODIFICATIONS/DELETIONS
Section 1.2	Package Symbolization and Revision Identification:
	Modified Figure 1 to include CTZ package and changed figure caption to: Lot Trace Code Examples for TMS320C6472 (CTZ/ZTZ Packages)

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