ANALOG DEVICES

SHARC Processor

Silicon Anomaly List — ADSP-21467/21469

ABOUT ADSP-21467/21469 SILICON ANOMALIES

These anomalies represent the currently known differences between revisions of the SHARC® ADSP-21467/21469 product(s) and the functionality specified in the ADSP-21467/21469 data sheet(s) and the Hardware Reference book(s).

SILICON REVISIONS

A silicon revision number with the form "-x.x" is branded on all parts (see the data sheet for information on reading part branding). The silicon revision can also be electronically read by reading the REVPID register either via JTAG or DSP code.

The following DSP code can be used to read the register:
<UREG> = REVPID;

Silicon REVISION	REVPID[7:4]
0.0	0000

ANOMALY LIST REVISION HISTORY

The following revision history lists the anomaly list revisions and major changes for each anomaly list revision.

Date	Anomaly List Revision	Data Sheet Revision	Additions and Changes
04/18/2011	E	Rev 0	Added anomalies: 15000021,15000022.
09/29/2010	D	Rev 0	Added anomalies: 15000018,15000020.
03/11/2010	C	PrF	Added anomaly: 15000016.
11/03/2009	B	PrD	Added anomalies: 15000010, 15000011, 15000012, 15000013, 15000014, and 15000015.
03/01/2009	A	PrC	Initial release

NR003870E

One Technology Way, P.O.Box 9106, Norwood, MA 02062-9106 U.S.A.
Tel: 781.329.4700 **www.analog.com**
Fax: 781.461.3113

SUMMARY OF SILICON ANOMALIES

The following table provides a summary of ADSP-21467/21469 anomalies and the applicable silicon revision(s) for each anomaly.

No.	ID	Description	0.0
1	15000002	Incorrect Popping of stacks possible when exiting IRQx/Timer Interrupts with DB modifier	x
2	15000003	IOP Register access immediately following an External Memory access may not work	x
3	15000004	Effect latency of some System Registers may be 2 cycles instead of 1 for External data accesses	x
4	15000005	Writes to LCNTR, CURLCNTR and LADDR from Internal Memory may fail if there is a DMA block conflict	x
5	15000006	Specific Core Stall may not execute as expected	x
6	15000007	Frame Sync Edge Detection (FSED) feature in SPORT TDM mode does not work	x
7	15000010	Incorrect value when the results of Enhanced Modify/BITREV Instruction are used in the very next Instruction	x
8	15000011	Incorrect Execution of CALL(DB) Instructions in VISA operation, under specific conditions	x
9	15000012	Latency with external FLAG-based Conditional instructions involving DAG register post-modify operation	x
10	15000013	Input leakage current drawn by MediaLB pins may be higher than expected	x
11	15000014	Special PLL Initialization Sequence required if MediaLB interface is used in DMA-driven transfer mode	x
12	15000015	Data error when reading large (>4096 words) block of data over DDR2 interface	x
13	15000016	When PM accesses are used, some Instructions may get corrupted under specific conditions	x
14	15000018	SPORT DMA may not work as expected, when SPORTs from the same DMA group access both the external memory and internal memory for data and/or TCB and other peripheral DMAs (including SPORT DMAs in other groups) are also enabled in parallel	x
15	15000020	PLL Programming may not take effect under specific conditions	x
16	15000021	After an emulator halt at the instruction before ‘idle' instruction, the Core Timer stops decrementing even after code execution restarts.	x
17	15000022	Corruption of Link Port Receiver Data	x

Key: x = anomaly exists in revision
. = Not applicable

DETAILED LIST OF SILICON ANOMALIES

The following list details all known silicon anomalies for the ADSP-21467/21469 including a description, workaround, and identification of applicable silicon revisions.

1. 15000002 - Incorrect Popping of stacks possible when exiting IRQx/Timer Interrupts with DB modifier:

DESCRIPTION:
If a delayed branch modifier (DB) is used to return from the interrupt service routines of any of IRQx (hardware) or timer interrupts, the automatic popping of ASTATx/ASTATy/MODE1 registers from the status stack may not work correctly.

The specific instructions affected by this anomaly are **`"RTI(DB);"`** and **`"JUMP(CI)(DB);"`**.

This anomaly affects only IRQx and Timer Interrupts as these are the only interrupts that cause the sequencer to push an entry onto the status stack. This anomaly applies to both internal and external memory execution.

WORKAROUND:
Do not use (DB) modifiers in instructions exiting IRQ or Timer ISRs. Instructions in the delay slot should be moved to a location prior to the branch.

Note: This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and Operating Systems supported by ADI, such as VisualDSP++ and VDK please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):
0.0

2. 15000003 - IOP Register access immediately following an External Memory access may not work:

DESCRIPTION:
If an instruction making an access to an IOP register immediately follows another instruction that performs an access to external memory, the IOP register access may not occur correctly.

WORKAROUND:
Separate the two instructions by inserting another instruction in between them, such as a NOP.

APPLIES TO REVISION(S):
0.0

3. 15000004 - Effect latency of some System Registers may be 2 cycles instead of 1 for External data accesses:

DESCRIPTION:
The following registers that have an effect latency of 1 (the maximum number of instructions it takes for a write to these registers to take effect) will instead have an effect latency of 2 if any of their bits impact an instruction containing an external data access:

```
MODE1, MODE2, MMASK, SYSCTL, BRKCTL, ASTATx, ASTATy, STKYx, and STKYy.
```

For example, consider the following sequence of instructions:

```
bit set MODE1 BR8;
nop;          //Sufficient if not immediately followed by external memory access instruction
nop;          //Extra NOP needed if following instruction accesses external memory
pm(i8,m12)=f9; //i8 is pointing to an address in external memory
```

Registers other than the ones listed above are not affected by this anomaly.
Note that the anomaly is independent of whether the instruction itself resides in internal or external memory.
Rather, the anomaly is encountered if there are external memory data accesses within the two instructions immediately following the register modification.

WORKAROUND:
If any of the above registers with an effect latency of 1 is modified, it is recommended that no accesses involving external memory (over either PM or DM bus) are performed in the two instructions immediately following the register modification. It is recommended to insert two NOPs after such register modifications.

APPLIES TO REVISION(S):
0.0

4. 15000005 - Writes to LCNTR, CURLCNTR and LADDR from Internal Memory may fail if there is a DMA block conflict:

DESCRIPTION:
Writes to LCNTR, CURLCNTR and LADDR from internal memory (either as a DM access or as a PM access) may fail when a DMA transfer to/from the same block occurs in the same cycle.

For example, consider the following instruction:

```
CURLCNTR = dm(i0,m0);
```

Now consider any DMA access involving the same memory block as pointed to by address **`(i0+m0)`**. If the DMA and the above write align in such a way that DMA transfer happens in the same cycle as the above instruction, then the above write will fail.

Also note that the anomaly also occurs if **`(i0+m0)`** points to a memory-mapped I/O register.

WORKAROUND:
1) Change the DMA to source/target a different internal memory block thereby avoiding any DMA block conflict.

2) Instead of loading these registers directly from memory, they can be loaded indirectly as a 2-step process as shown below:

```
r0 = dm(i0,m0);
CURLCNTR = r0;
```

APPLIES TO REVISION(S):
0.0

5. 15000006 - Specific Core Stall may not execute as expected:

DESCRIPTION:
Under certain specific conditions outlined below, one type of core stall that is normally incurred does not get executed. In certain cases this can cause incorrect code operation.

When PCSTK is loaded, and an RTS/RTI is executed immediately afterward, there is a stall as the return waits for a writeback of PCSTK before the return.
Example1:

```
[1] PCSTK = dm(i0,m1);
[2] RTS;
```

Example2:

```
[1] PCSTK = dm(i0,m1);
[2] RTI;
```

If: The memory access in instruction 1 is to a memory-mapped IOP register.
-OR-
DMA is simultaneously accessing the same bank as the memory access in instruction 1.
Then: Stall does not occur, and this results in the RTS/RTI vectoring to some undefined location instead of the value from PCSTK.

WORKAROUND:
Add a nop between instructions 1 and 2.

Note: This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and Operating Systems supported by ADI, such as VisualDSP++ and VDK please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):
0.0

6. 15000007 - Frame Sync Edge Detection (FSED) feature in SPORT TDM mode does not work:

DESCRIPTION:
The newly defined Frame Sync Edge Detection (FSED) feature in SPORT TDM mode does not work. This feature was introduced to ensure that when a SPORT is operating in TDM mode with external clock and frame sync, it waits for an active edge of the external frame sync signal before initiating SPORT RX/TX transfers, regardless of whether it was enabled during the time of the frame sync signal being at an active level.

WORKAROUND:
Do not set the FSED bit (bit 2 within the SPCTLNn register). This will ensure that the SPORT functions identically as on prior products.

APPLIES TO REVISION(S):
0.0

7. 15000010 - Incorrect value when the results of Enhanced Modify/BITREV Instruction are used in the very next Instruction:

DESCRIPTION:
In the following specific sequence of instructions, the memory or the register load in **INSTR3** will not contain the correct updated value of the DAG register **Ia** from **INSTR2**, but rather its value from **INSTR1**:

```
INSTR1: Ia = <immediate load | register load | memory load>;
INSTR2: Ia = MODIFY|BITREV (Ib, Mc);
INSTR3: <memory load | register load>  = Ia;
```

Note that this anomaly is only applicable in the case where **Ia** and **Ib** are unique and different.

WORKAROUND:
The user must avoid the above exact sequence of instructions which might produce an incorrect result.

Note: This workaround may be built into the development tool chain and/or into the operating system source code. For tool chains and Operating Systems supported by ADI, such as VisualDSP++ and VDK please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):
0.0

8. 15000011 - Incorrect Execution of CALL(DB) Instructions in VISA operation, under specific conditions:

DESCRIPTION:
Direct and indirect CALL(DB) instructions might not be executed correctly if all of the following specific conditions are true:

1. The CALL(DB) instruction being executed belong to the Variable-Instruction-Size Architecture instruction set (i.e., VISA instructions).

2. An interrupt occurs simultaneously such that it causes the CALL(DB) instruction to be aborted.

3. A DMA block conflict causes a core pipeline stall to occur in the immediate cycle following step 2.

If all of the above conditions are true, the CALL(DB) instruction is never executed, and consequently the function call is missed. Upon returning from the interrupt, program execution resumes from the instruction immediately following the CALL(DB) instruction.

Note that a DMA block conflict occurs if instructions and DMA data buffers are stored in the same internal memory block. The DMA activity itself could either be a peripheral or external port DMA, or a DMA engine associated with one of the dedicated hardware accelerators.

Also note that this anomaly only affects conditional and unconditional CALL(DB) instructions in VISA and does NOT apply to traditional (i.e., non-VISA) CALL(DB) instructions. The anomaly also does NOT apply to any other VISA branch instructions (such as delayed/non-delayed CJUMP and JUMP instructions) or to non-delayed CALL instructions.

WORKAROUND:
The anomaly can be avoided if any one of the following workarounds is followed:

1. Ensure that there is no DMA block conflict. In other words, DMA data buffers must be placed in a different internal memory block than the block(s) containing instructions.

2. Avoid the use of CALL(DB) instructions in VISA functions.

3. When building an executable, modules or functions containing CALL(DB) instructions are built using traditional instruction set and not VISA instruction set.

APPLIES TO REVISION(S):
0.0

9. 15000012 - Latency with external FLAG-based Conditional instructions involving DAG register post-modify operation:

DESCRIPTION:
External FLAG-based Conditional instructions involving DAG register post-modify operation must not be followed immediately by an instruction that uses the same index register.

For example, in the following instruction sequence shown below:

```
INSTR1: IF COND dm(Ia,Mb); //any instruction that involves post-modify operation
INSTR2: dm(Ia,Mc); //any instruction that depends on updated Ia value
```

The value of the DAG index register in **INSTR2** will either be **Ia** or **(Ia+Mb)** depending on whether **INSTR1** was aborted or executed.

In the unique case where **COND** is an external **FLAG** condition (for example, say **FLAG2_IN**) which is set asynchronously by an external source or event, the necessary internal stalls which would result in the DAG index register getting the correct value do not take effect, and consequently the value of the DAG index register in **INSTR2** may not contain the correct and expected value.

WORKAROUND:
Separate the instructions in the above sequence by at least two NOPs.

APPLIES TO REVISION(S):
0.0

10. 15000013 - Input leakage current drawn by MediaLB pins may be higher than expected:

DESCRIPTION:
The maximum specified input leakage current for MediaLB interface in 3-pin mode is around 1uA. However, in the MediaLB interface present on ADSP-21462W, ADSP-21465W, and ADSP-21469W, this value may be significantly higher under worst-case conditions (for example, around 116uA). This will not damage any other devices on the MediaLB bus or limit the number of devices that can share the bus.

WORKAROUND:
Remove (or omit) the external 47Kohm pulldown resistor that are usually present on the board. The purpose of this resistor is to ensure that the signals are driven low when none of the devices on the bus is actively driving the bus, and is no longer needed as our processor's MediaLB pins have internal pulldowns of their own.

Note that the effect of not having this external pull-down resistor on the bus may have an influence on the Bus Hold Times of a signal. Therefore, customers are advised to measure their bus hold timing and ensure that they do not violate the values specified in the datasheet.

APPLIES TO REVISION(S):
0.0

11. 15000014 - Special PLL Initialization Sequence required if MediaLB interface is used in DMA-driven transfer mode:

DESCRIPTION:
The MediaLB interface's DMA clock may lose synchronization with the processor's internal clock if the procesor's PLL is placed in bypass mode as part of the initial initialization sequence.

WORKAROUND:
Use the following instruction sequence while initializing the PLL if using MediaLB interface in DMA-driven transfer mode:

1. Disable clock to the MediaLB interface.

```
ustat4 = dm(PMCTL1);
bit set ustat4 MLBOFF;
R2=dm(MLB_VCCR); // any "dummy" read of an IOP register outside of the core to force
                 // clock synchronization between core and peripheral clock domains
dm(PMCTL1) = ustat4;
```

2. Place PLL in bypass mode.
3. Proceed with programming PLL parameters as per default guidelines. Provide sufficient delay in order for the changes to take effect, again as per default guidelines.
4. Bring PLL out of bypass mode and re-enable the clock to the MediaLB interface:

```
ustat1 = dm(PMCTL);
bit clr ustat1 PLLBP;
ustat4 = dm(PMCTL1);
bit clr ustat4 MLBOFF;
dm(PMCTL) = ustat1;
dm(PMCTL1) = ustat4;
```

Ensure that the above instruction sequence is executed from within internal memory, is not interrupted, and that there is no background DMA activity.

Note that the above procedure does not need to be followed if the MediaLB interface is not used in a system, or if the MediaLB interface is expected to operate only with core-driven data transfers.

APPLIES TO REVISION(S):
0.0

12. 15000015 - Data error when reading large (>4096 words) block of data over DDR2 interface:

DESCRIPTION:
If there are more than 4096 uninterrupted continuous sequential reads from external DDR2 memory (they may be either core-driven accesses or DMA), there will be a data error observed at the 4097th memory location of the destination buffer, in the form of a replicated data word (the same word as was read into the 4096th location).

WORKAROUND:
Implementing any one of the following workarounds will avoid the problem:

1. Limit the size of uninterrupted reads from external DDR2 memory to less than or equal to 4096 32-bit words.

2. Disable DDR2 read optimization. But note that doing so will affect performance.

APPLIES TO REVISION(S):
0.0

13. 15000016 - When PM accesses are used, some Instructions may get corrupted under specific conditions:

DESCRIPTION:
When specific PM accesses are used, either the PM instruction or some Instructions which follow this instruction may get corrupted. The problem is seen when the PM accesses has any of the below conditions:

1. Conflicts with another core/DMA access to the same memory block
2. Accesses any of the memory mapped(IOP) registers
3. Accesses the external memory space

CASE 1 (applicable for both VISA and NON-VISA mode):
The single instruction loop which has the PM access instruction described above as part of the instruction in the loop, may not work as expected. The PM instruction in the loop while being fetched from the cache may get corrupted. This is applicable for both counter based and non-counter based loops. For counter based loops the corruption occurs only if the count value is greater than four.

Example1:

```
lcntr=x, do (pc,1) until lce;    // x > 4
dm(I0,M0)=R10, pm(I12,M10)=R10;  //PM access meets one of the conditions described
```

Example2:

```
lcntr=x, do (pc,1) until lce;    // x > 4
R10 = pm(I12,M10);               //PM access meets one of the conditions described
```

Example3:

```
do (pc,1) until forever;
R10 = pm(I12,M10);              //PM access meets one of the conditions described
```

CASE 2 (applicable only for VISA mode):
A code sequence which includes two successive PM instructions and the second PM access instruction is compressed and meets one of the conditions described above, may not work as expected. The instruction(s) following the second PM access (while getting fetched from the cache) may get corrupted. The particular instruction(s) which get corrupted is dependent on the size of the instructions following the above sequence and their alignment in the Instruction Alignment Buffer (IAB). This anomaly is not applicable for the pm sequence which is formed by the first and last instruction of a hardware loop containing pm accesses.

Example1:

```
 pm(I12,M10)= <imm_data>; //PM access 1, <imm_data> - can be 16-bit/32-bit immediate value
compressed or uncompressed
 dm(I0,M0)=R10, pm(I12,M10)=R10;  //PM access 2 compressed, meets one of the conditions
described
 ...
 ...
```

Instruction(s) following this sequence may get corrupted.

Example2:

```
pm(I12,M10)= <imm_data>; //PM access 1, <imm_data> - can be 16-bit/32-bit immediate value
R10 = pm(I12,M10);       //PM access 2, meets one of the conditions described
...
...
```

Instruction(s) following this sequence may get corrupted.

Note that this anomaly is seen especially at low operating temperatures, but it is not limited to any particular operating temperature range.

WORKAROUND:

Applicable for CASE 1:

1. Avoid memory block conflict stalls for these scenarios by moving either the core/DMA access or the PM access to different memory block.

2. Avoid single instruction loops having program memory access by unrolling the loop. This can be done by one of the following ways:
 a. Replicating the instruction and correspondingly reducing the loop count
 b. Adding another instruction or NOP instruction to the loop
 c. Breaking the instruction which contains pm access into two separate instructions.

3. Disable the cache for the problematic PM accesses.

```
BIT SET MODE2 CADIS;
nop;
nop;
lcntr=x, do (pc,1) until lce;   // x > 4
dm(I0,M0)=R10, pm(I12,M10)=R10; // PM access meets one of the conditions described
BIT CLR MODE2 CADIS;
nop;
nop;
```

Note that disabling the cache may affect the performance of the hardware loop. The PM instruction in the loop will always stall and the loop will take double the time than the cache enabled case.

Applicable only for CASE 2:

1. Avoid memory block conflict stalls for these scenarios by moving either the core/DMA access or the PM access to different memory block.

2. Add a "nop" or a non-PM access instruction between the two problematic PM accesses.

3. All sequences of PM accesses with length more than 1 should remain uncompressed from second instruction onwards. This can be done by using the assembler directive ".NOCOMPRESS".

```
pm(I12,M10)= <imm_data>; //PM access 1, <imm_data> - can be 16-bit/32-bit
                         //immediate value
.NOCOMPRESS;             //Disable compression
dm(I0,M0)=R10, pm(I12,M10)=R10; //PM access 2, meets one of the condition described
.COMPRESS;                      //Enable compression
```

4. Replace the pm access with the dm access.

5. Disable the cache for the problematic PM accesses.

```
BIT SET MODE2 CADIS;
nop;
nop;
pm(I12,M10)= <imm_data>; //PM access 1, <imm_data> - can be 16-bit/32-bit
                         //immediate value
dm(I0,M0)=R10, pm(I12,M10)=R10; //PM access 2, DM access points to the same
                                //memory block resulting in a block conflict
BIT CLR MODE2 CADIS;
nop;
nop;
```

Note: Some of these workarounds may be built into the development tool chain and/or into the operating system source code. For tool chains and Operating Systems supported by ADI, such as VisualDSP++ and VDK please consult the "Silicon Anomaly Tools Support" help page in the applicable documentation and release notes for details.

APPLIES TO REVISION(S):

0.0

14. 15000018 - SPORT DMA may not work as expected, when SPORTs from the same DMA group access both the external memory and internal memory for data and/or TCB and other peripheral DMAs (including SPORT DMAs in other groups) are also enabled in parallel:

DESCRIPTION:
The SPORT DMA channels for the SHARC processors are arranged in 4 groups as follows:

a. Group 1 : DMA channels for the SPORTs 0 and 1 (SP0A,SP0B, SP1A and SP1B)
b. Group 2 : DMA channels for the SPORTs 2 and 3 (SP2A,SP2B, SP3A and SP3B)
c. Group 3 : DMA channels for the SPORTs 4 and 5 (SP4A,SP4B, SP5A and SP5B)
d. Group 4 : DMA channels for the SPORTs 6 and 7 (SP6A,SP6B, SP7A and SP7B)

The SPORT DMA may not work as expected, when the SPORTs from the same group are enabled in DMA mode with read/write transactions to both internal and external memory and another DMA also enabled in parallel. This problem is applicable for both the normal DMA mode and DMA chaining mode of SPORT DMAs.

When the problem occurs, the data read from the memory may not be correct for the SPORT transmit DMA. For the chaining DMA operations, additionally the TCB may also not be loaded correctly. For the receive DMA case, the data written to the memory will be correct but the latency between two successive writes will increase.

WORKAROUND:
1. For normal DMA mode, place all the SPORT DMA buffers belong to the same group either in internal memory or in external memory,but not on both.

2. For DMA chaining mode, place both the SPORT TCBs and SPORT DMA buffers belong to the same group, either in internal memory or in the external memory, but not on both.

APPLIES TO REVISION(S):
0.0

15. 15000020 - PLL Programming may not take effect under specific conditions:

DESCRIPTION:

The PLL may not get programmed to the new value in software under specific conditions. This problem is seen under very specific conditions of temperature, frequency, and operating voltage. The reason for the failure is that the new PLL multiplier values configured using the PLLM bits are not updated correctly and the PLL continues to use the previous multiplier value only. The symptom of the anomaly will be that the PLL will continue to work at the old frequency (regardless of the new value programmed).

Note that while this anomaly is only observable in very specific and narrow ranges of voltage, temperature, and frequency, these ranges do fall within valid user operating conditions and therefore not provided as a workaround. This anomaly is also not seen for cases where any portion of the PLL divider circuitry alone are re-programmed (such as for example INDIV, PLLDx, LPCKRx, or SDCKRx/DDR2CKRx ratios).

WORKAROUND:

When programming the PLL to a new frequency in software, please follow the following sequence described below in order to ensure that the PLL is programmed correctly. The first time, the PLL is placed in bypass in three discrete steps:
1. The input clock divider (INDIV) bit in the PMCTL register is set,
2. The PLL multiplier value is not modified, but rather kept at the same value as before,
3. The PLL is placed in bypass mode for the appropriate number of cycles.

Note: In case of the first time that the PLL is programmed after emerging from reset, this value will be the same as represented by the hardware CLK_CFGx pins. For subsequent re-programmings of the PLL, the multiplier value will be the same as what had been previously programmed.

The PLL is brought out of bypass after the appropriate number of cycles, and the second time around, is now re-programmed with the desired PLL multiplier and divider values as usual.

The following code sequence is provided below as an example only. Note that programming of other PLL-derived clocks such as peripheral clocks remains unaffected by this anomaly and can be programmed in the normal manner.

Example Assembly Code Sequence

```
#include <def21469.h>    // for ADSP-2146x family of processors
// #include <def21479.h>  // for ADSP-2147x family of processors
// #include <def21489.h>  // for ADSP-2148x family of processors

// Set INDIV bit in PMCTL register
USTAT1 = dm(PMCTL);
bit set USTAT1  INDIV;
dm(PMCTL) = USTAT1;

//Program PLL to same value as CLK_CFGx pins/previously programmed value in software……
bit set USTAT1  PLLM_prev;
dm(PMCTL) = USTAT1;

//…then place PLL in bypass mode
bit set USTAT1  PLLBP;
dm(PMCTL) = USTAT1;

// Wait recommended number of cycles
lcntr=4096, do (PC,1) until lce;
nop;

// Bring PLL out of bypass mode by clearing PLLBP bit
USTAT1 = dm(PMCTL);
bit clr USTAT1 PLLBP;
dm(PMCTL) = USTAT1;

//Wait for PLL bypass exit
lcntr=16, do (PC,1) until lce;
nop;

// Read the PMCTL register contents and clear the previous  PLL bit fields
USTAT1 = dm(PMCTL);
// Clears all the Multiplier, Divider, INDIV, DDR2CKR and LPCKR bits of PMCTL
// for ADSP-2146x processors.
bit clr USTAT1 0x7C01FF;
```

```
// Clears all the Multiplier, Divider, INDIV and SDCKR bits of PMCTL for ADSP-
// 2147x/ADSP-2148x processors.
//bit clr USTAT1 0x1C01FF;

// If INDIV is needed for new programming parameters, set INDIV bit in PMCTL register…
#ifdef INDIV_required
bit set USTAT1   INDIV;
dm(PMCTL) = USTAT1;

//…program PLL with new parameters: PLLM32,PLLD2,and DIVEN in this example
// Also program any PLL derived peripheral clocks here...
bit set USTAT1  PLLM32|PLLD2|DIVEN|DDR2CKR2|LPCKR3;
dm(PMCTL) = USTAT1;

// and then place PLL in bypass mode for second time
bit set USTAT1  PLLBP;
bit clr USTAT1 DIVEN;
dm(PMCTL) = USTAT1;
#endif

// If INDIV is not needed, clear INDIV bit in PMCTL register…
#ifdef INDIV_not_required
bit clr USTAT1   INDIV;
dm(PMCTL) = USTAT1;

//…program PLL with new parameters: PLLM16,PLLD2,and DIVEN in this example
// Also program any PLL derived peripheral clocks here...
bit set USTAT1  PLLM16|PLLD2|DIVEN|DDR2CKR2|LPCKR3;
dm(PMCTL) = USTAT1;

// and then place PLL in bypass mode for second time
bit set USTAT1  PLLBP ;
bit clr USTAT1 DIVEN;
dm(PMCTL) = USTAT1;
#endif

// Wait recommended number of cycles
lcntr=4096, do (PC,1) until lce;
nop;

// Bring PLL out of bypass mode by clearing PLLBP bit
USTAT1 = dm(PMCTL);
bit clr USTAT1 PLLBP;
dm(PMCTL) = USTAT1;

//Wait for PLL bypass exit
lcntr=16, do (PC,1) until lce;
nop;
// PLL will be programmed correctly at this point
```

Example C Code sequence

```
#include <def21469.h>    // for ADSP-2146x family of processors
#include <cdef21469.h>
// #include <def21479.h>  // for ADSP-2147x family of processors
// #include <cdef21479.h>
// #include <def21489.h>  // for ADSP-2148x family of processors
// #include <cdef21489.h>

int i, pmctlsetting;

// Set INDIV bit in PMCTL register
pmctlsetting= *pPMCTL;
```

```
pmctlsetting|= INDIV;
*pPMCTL= pmctlsetting;

//Program PLL to same value as CLK_CFGx pins/previously programmed value in software……
pmctlsetting|= PLLM_prev;
*pPMCTL= pmctlsetting;

//…then place PLL in bypass mode
pmctlsetting|= PLLBP;
*pPMCTL= pmctlsetting;


//Wait for recommended number of cycles
for (i=0; i<4096; i++)
    asm("nop;");

// Bring PLL out of bypass mode by clearing PLLBP bit
*pPMCTL ^= PLLBP;

for (i=0; i<16; i++)
    asm("nop;");

// Read the PMCTL register contents  and clear the previous  PLL bit fields
pmctlsetting= *pPMCTL;
// Clears all the Multiplier, Divider, INDIV, DDR2CKR and LPCKR bits of PMCTL
// for ADSP-2146x processors.
pmctlsetting&= ~(0x7C01FF);

// Clears all the Multiplier, Divider, INDIV and SDCKR bits of PMCTL for ADSP-
// 2147x/ADSP-2148x processors.
//pmctlsetting&= ~(0x1C01FF);

// If INDIV is needed for new programming parameters, set INDIV bit in PMCTL register…
#ifdef INDIV_required
pmctlsetting|= INDIV;
*pPMCTL= pmctlsetting;

//…program PLL with new parameters: PLLM32,PLLD2,and DIVEN in this example
// Also program any PLL derived peripheral clocks here...
pmctlsetting|= PLLM32|PLLD2|DIVEN|DDR2CKR2|LPCKR3;
*pPMCTL= pmctlsetting;

// and then place PLL in bypass mode for second time
pmctlsetting|= PLLBP;
pmctlsetting^=DIVEN;
*pPMCTL= pmctlsetting;
#endif

// If INDIV is not needed, clear INDIV bit in PMCTL register…
#ifdef INDIV_not_required
pmctlsetting^= INDIV;
*pPMCTL= pmctlsetting;

//…program PLL with new parameters: PLLM16,PLLD2,and DIVEN in this example
// Also program any PLL derived peripheral clocks here...
pmctlsetting|= PLLM16|PLLD2|DIVEN|DDR2CKR2|LPCKR3;
*pPMCTL= pmctlsetting;

// and then place PLL in bypass mode for second time
pmctlsetting|= PLLBP;
pmctlsetting^=DIVEN;
*pPMCTL= pmctlsetting;
#endif

// Wait recommended number of cycles
for (i=0; i<4096; i++)
          asm("nop;");
```

```
// Bring PLL out of bypass mode by clearing PLLBP bit
    *pPMCTL ^= PLLBP;

for (i=0; i<16; i++)
         asm("nop;");
// PLL will be programmed correctly at this point
```

APPLIES TO REVISION(S):
0.0

16. 15000021 - After an emulator halt at the instruction before ‘idle' instruction, the Core Timer stops decrementing even after code execution restarts.:

DESCRIPTION:
When the processor is halted at a breakpoint in an emulator session, the core timer correctly stops decrementing and restarts when code execution is resumed. However, if the emulator breakpoint is placed at an instruction just before an idle instruction, the core timer remains halted even after the code execution is resumed.

In the example code below, the core timer remains halted after code execution is resumed following the core halt at Instruction1. The same behavior is seen if Instruction1 is executed by ‘single stepping'.

```
Enable_Core_timer:
……
Bit set MODE1 TIMEN;
……
……
INSTR1;                // Placing breakpoint here causes the anomaly
Idle;
INSTR2;
INSTR3;
```

Note: This issue impacts only Emulator session debug.

WORKAROUND:
None

APPLIES TO REVISION(S):
0.0

17. 15000022 - Corruption of Link Port Receiver Data:

DESCRIPTION:
Under rare conditions, the asynchronous assertion (going high) of LACKx signal by the Link Port receiver (to resume a halted transfer from the transmitter) may not be detected properly by the transmitter. As a result of this, the transmit data changes as expected but the Link Port Clock stays high, thereby resulting in data corruption at the receiver end.

The symptoms of this anomaly are observed as the receiver missing the least significant byte (the first byte) at the reception of a particular word. Consequently, the corruption will be propagated as a byte shift for the rest of the transfer.

The occurrence of this anomaly is observed to be independent of Link Port frequency as well as to the ratio of Core Clock to Link Port Clock frequencies, although the probability of error over time will change based on different ratios.

The probability of error can be reduced by reducing the core frequency, and setting the ratio of core-to-link-port-clock divide ratio (LCLKR bit field in the PMCTL register) to the slowest possible value of 1:4.

Error recovery from a data corruption can be achieved by implementing a simple re-transmit protocol in conjunction with the above steps.The link port transmitter embeds a simple checksum as part of the transmitted data payload. This checksum is calculated by the link port receiver and if the checksums don't match, the receiver requests the transmitter to re-transmit the previous data block (via a simple communication protocol, such as a Flag).

WORKAROUND:
The anomaly can be avoided if the receiver does not de-assert (low) its LACKx signal during the course of the data transfer.

1. In the case where the link port receiver is an FPGA, this can be achieved by increasing the depth of the receiver FIFO sufficiently so that it can completely accommodate all transmitted data without having to de-assert the LACKx signal.

2. In the case where the link port receiver is another ADSP-21469, the anomaly can be avoided by using the following link port protocol:

2a. For DMA-based transfers, set up the DMA block transfer size to two 32-bit words. Disable the transmitter inside the "DMA external transfer done interrupt". Use the LTRQ interrupt to wake up the link port transmitter again when LACKx goes high and re-enable the transmitter inside the ISR to send the next two words, and so on.

2b. For core-driven transfers, load the 2-deep TX FIFO with two 32-bit words. The link port transmitter will then poll for its LPBS bit and disable the transmitter when the transfer is complete. Use the LTRQ interrupt to wake up the transmitter when LACKx goes high and re-enable the transmitter inside the ISR to send the next two words, and so on.

APPLIES TO REVISION(S):
0.0