Tiva™ C Series TM4C129x Microcontrollers Silicon Revisions 1, 2, and 3

1 Introduction

This document describes known exceptions to the functional specifications for all of the Tiva™ C Series TM4C129x microcontrollers. Note that some features are not available on all devices in the series, so not all errata may apply to your device. See your device-specific data sheet for more details.

For details on ARM® Cortex™-M4F CPU advisories, see the *ARM Core Cortex™-M4 (AT520) and Cortex-M4F (AT521) Errata Notice* (SPMZ637).

2 Device Nomenclature

To designate the stages in the product development cycle, TI assigns prefixes to the part numbers of all microcontroller (MCU) devices. Each Tiva C series family member has one of two prefixes: XM4C or TM4C (for example, **XM4C**129XNCPDTI). These prefixes represent evolutionary stages of product development from engineering prototypes (XM4C) through fully qualified production devices (TM4C).

Device development evolutionary flow:

XM4C — Experimental device that is not necessarily representative of the final device's electrical specifications and may not use production assembly flow.

TM4C — Production version of the silicon die that is fully qualified.

XM4C devices are shipped against the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

TM4C devices 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 (XM4C) 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.

3 Device Markings

Figure 1 shows an example of the Tiva™ C Series TM4C129x microcontroller package symbolization.

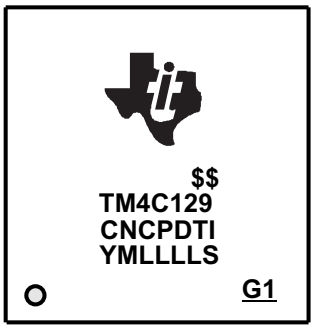


Figure 1. Example of Device Part Markings

This identifying number contains the following information:

- **Lines 1 and 5:** Internal tracking numbers
- **Lines 2 and 3:** Part number

For example, TM4C129C on the second line followed by NCPDTI on the third line indicates orderable part number TM4C129CNCPDTI. Note that the first letter in the part number indicates the product status. A T indicates the part is fully qualified and released to production; an X indicates the part is experimental (pre-production) and requires a waiver. Pre-production parts also include a revision number in the part number, for example, **X**M4C129C followed by NCPDTI**3** indicates revision 3. The **DID0** register identifies the version of the microcontroller, as shown in Table 1. The MAJOR and MINOR bit fields indicate the die revision number. Combined, the MAJOR and MINOR bit fields indicate the TM4C129x microcontroller silicon revision number.

Table 1. Tiva™ C Series TM4C129x Silicon Revision Codes

MAJOR Bit Field Value	MINOR Bit Field Value	Die Revision	Silicon Revision
0x0	0x0	A0	1
0x0	0x1	A1	2
0x0	0x2	A2	3

- **Line 4:** Date code The first two characters on the fourth line indicate the date code, followed by internal tracking numbers. The two-digit date code YM indicates the last digit of the year, then the month. For example, a 34 for the first two digits of the fourth line indicates a date code of April 2013.

4 Advisory to Silicon Revision Correlation

Table 2. Advisory to Silicon Revision Matrix

Advisory Number	Advisory Title	Silicon Revision(s) Affected		
		1	2	3
ADC				
ADC#09	First two ADC Samples From the Internal Temperature Sensor Must be Ignored	X	X	X
ADC#13	A Glitch can Occur on pin PE3 When Using any ADC Analog Input Channel to Sample	X	X	X
ADC#14	The First two ADC Samples may be Incorrect	X	X	X
ADC#15	ADC Global Synchronization does not function	X		
ADC#16	Phase Offset does not Delay as Expected if Sample Sequencers are not Triggered at the Same Time	X	X	
CRC				
CRC#01	Any Data Read From a Non-CRC Register After Accessing the CRCRSLTPP Register is Incorrect	X		
DMA				
DMA#02	µDMA Data may be Corrupted if Transferred or Received While Entering or Exiting Deep Sleep Mode	X	X	X
ELEC				
ELEC#02	V_{BAT} Supply pin may be Damaged if the pin Voltage Ramps Faster Than 0.7 V/µs	X	X	X
ELEC#03	PIOSC Frequency Variance does not Meet +/- 4.5% Across Voltage and Temperature		X	
ELEC#04	Equation for C_L for HIBXOSC is not Correct	X	X	X
ELEC#06	PLL VCO Minimum is not Specified	X	X	X
ELEC#07	TFlash and TFlashlpds Parameter Incorrect	X	X	X
EPI				
EPI#01	Data Reads can be Corrupted when the Code Address Space in the EPI Module is Used	X	X	X
Ethernet				
ETH#01	The LED Polarity Bit in the Ethernet MAC Clock Configuration Register Does Not Function	X		
ETH#02	Ethernet May not Initialize After System Reset	X	X	X
ETH#03	RBIAS Resistor Required Independent of the use of On-Chip Ethernet PHY	X	X	X
ETH#04	SMI Electrical Section Update (1)	X	X	X
ETH#05	SMI Electrical Section Update (2)	X	X	X
GPIO				
GPIO#03	GPIO Pins may Glitch on Power up	X		
GPIO#09	In Some Cases, Noise Injected Into GPIO pins PB0 and PB1 can Cause High Current Draw	X	X	X
General-Purpose Timers				
GPTM#09	General-Purpose Timers do not Synchronize When Configured for RTC Mode	X	X	X
GPTM#12	The GPTMPP Register Does not Correctly Indicate Alternate Clock Capability	X		
GPTM#13	Reading the GPTMTnV, the GPTMTnR, or the GPTMRTCPD Registers may Return Incorrect Values When Using an Alternate Clock Source	X	X	X
GPTM#14	General-Purpose Timer ADC and µDMA Triggers may not be Captured in Certain Modes When Using PIOSC	X		
GPTM#15	Counter Does not Immediately Clear to 0 When MATCH is Reached In Edge Count Up Mode	X	X	X
GPTM#16	Registers Cannot be Written to When Operating the GPTM in One-Shot or Periodic 32-bit Mode with the Alternate Clock Source	X	X	X
GPTM#17	The GPTMSYNC Register Bits Must be Manually Cleared when Using an Alternate Clock Source	X		

Table 2. Advisory to Silicon Revision Matrix (continued)

Advisory Number	Advisory Title	Silicon Revision(s) Affected		
		1	2	3
Hibernation				
HIB#10	If MEMCLR is set to a Non-Zero Value, a Tamper Event may not Clear all of the Bits in the HIBDATA Register	X	X	X
HIB#12	Tamper Logging Failure on XOSC Fail Event	X		
HIB#13	Tamper Events may be Missed in log	X		
HIB#15	The VDDFAIL Interrupt bit in the HIBIC Register is not Properly Cleared	X		
HIB#16	Application Code May Miss New Tamper Event During Clear	X	X	X
HIB#17	$\overline{\text{WAKE}}$ Cannot be Used to Wake From Hibernate Mode	X(1)		
HIB#18	Can get two Matches per day in Calendar Mode	X	X	X
HIB#19	The First Write to the HIBCTL Register may not Complete Successfully After a Hibernation Module Reset	X	X	X
I2C				
I2C#05	Register Read of I2CMCS may Cause an ADDR or DATA NAK to be Missed	X	X	X
I2C#08	I2C Master BUSY Status bit does not get set Immediately	X	X	X
LCD				
LCD#01	LIDD-Mode DMA Transactions in the LCD Controller Cause the Microcontroller to Become Unresponsive	X	X	X
LCD#02	LCD DMA FIFO Underflow Interrupt Occurs When EPI is Mapped to an External SDRAM With an Address That is not 0x1000.0000	X	X	X
LCD#03	LCD Module Does not Restart if an Underflow Occurs	X	X	X
	Memory			
MEM#03	EEPROM Data May be Corrupted if an EEPROM Write or Erase is Interrupted	X	X	
MEM#07	Soft Resets should not be Asserted During EEPROM Operations			X
MEM#09	ROM_SysCtlClockFreqSet() does not Properly Configure MOSC	X		
MEM#11	The ROM Version if the TivaWare EEPROMInit API Does not Correctly Initialize the EEPROM			X
MEM#12	Code Jumps from Flash to ROM when EEPROM is Active may Never Return	X	X	
MEM#13	Concurrent µDMA Reads from Flash Memory and EEPROM Accesses May Fetch Incorrect µDMA Data	X	X	
MEM#15	Specific Flash Locations in any Sector do not get Erased	X	X	X
MEM#16	JTAG Unlock Issue when BOOTCFG is Committed with Debug Disabled	X	X	X
MEM#17	Clearing Reserved Bits of BOOTCFG Causes ROM Boot Loader to Fail	X	X	X
MEM#18	Only Lower 512KB Flash can be Protected on 1MB Devices	X	X	X
MEM#19	Certain GPIOs Cannot be Configured as boot pins	X	X	X
MEM#20	Setting Mirror mode bit Causes Bus Faults on 512KB Flash	X	X	X
ONEWIRE				
ONEWIRE#01	A Delay is Needed for the 1-Wire Receive Configuration	X		
PWM				
PWM#01	Under Certain Circumstances, the PWM Load Interrupt is Triggered as Soon as the PWM is Enabled	X		
PWM#02	Setting the PWMSYNC Bits May Not Synchronize the PWM Counters if PWMDIV is Used	X		
PWM#03	The PWM Generators May Not Generate Interrupts or ADC Triggers	X		
PWM#04	PWM Generator Interrupts can only be Cleared 1 PWM Clock Cycle After the Interrupt Occurs	X	X	X
PWM#05	Generator Load with Global Sync May Lead to Erroneous Pulse Width	X	X	X
PWM#06	PWM Output May Generate a Continuous High Instead of a Low or a Continuous Low Instead of High	X	X	X

(1) Applicable to devices with date codes earlier than 0x38 (August 2013).

Table 2. Advisory to Silicon Revision Matrix (continued)

Advisory Number	Advisory Title	Silicon Revision(s) Affected		
		1	2	3
QEI				
QEI#01	When Using the Index Pulse to Reset the Counter, a Specific Initial Condition in the QEI Module Causes the Direction for the First Count to be Misread	X	X	X
SSI				
SSI#03	SSI1 can Only be Used in Legacy Mode	X	X	X
SSI#04	The First Byte Sent by the QSSI in Master Mode is Incorrect when Using the Alternate Clock	X		
SSI#05	Bus Contention in Bi- and Quad-Mode of SSI	X	X	X
SSI#06	SSI Receive FIFO Time-out Interrupt may Assert Sooner than Expected in Slave Mode	X	X	X
SSI#07	SSI Transmit Interrupt Status Bit is not Latched	X	X	X
SSI#08	SSI Slave in Bi and Quad Mode swaps XDAT0 and XDAT1	X	X	X
System Control				
SYSCTL#03	The MOSC Verification Circuit Does not Detect a Loss of Clock After the Clock has been Successfully Operating	X	X	X
SYSCTL#09	Some Devices may not Start Properly During Power Up if V_{DDC} is Decaying Between 200 and 100 mV When Power is Reapplied	X		
SYSCTL#12	MOSC Does not Power Down in Deep Sleep when it is not the Deep Sleep Clock Source	X		
SYSCTL#13	The NMIC register does not indicate NMI sources when read	X		
SYSCTL#14	Power Consumption is Higher When MOSC is Used in Single-Ended Mode	X		
SYSCTL#15	Watchdog Reset Improperly Updates the NMIC Register	X		
SYSCTL#16	On-Chip LDO may not Start Properly During Power Up	X	X	
SYSCTL#18	DIVSCLK Outputs a Different Clock Frequency than Expected when DIV = 0x0	X	X	X
SYSCTL#19	Extra Steps Required to Unlock the Device if NW = 0	X	X	X
SYSCTL#22	Change to the PLL Clock Divider may Cause System Clock to be out of Specification	X	X	X
SYSCTL#23	MOSC as the Source to OSCCLK may Cause a bus Fault on Reset	X	X	X
UART				
UART#01	When UART SIR Mode is Enabled, µDMA Burst Transfer Does not Occur	X	X	X
USB				
USB#02	USB Controller Sends EOP at end of Device Remote Wake-Up	X		
USB#03	Any Data Read From a Non-USB Register After Accessing the USBPP, USBPC, or the USBCC Register is Incorrect	X		
USB#04	Device Sends SE0 in Response to a USB Bus Reset	X	X	X
USB#05	USB Resume Occasionally does not Wake Device from Deep Sleep	X	X	X
Watchdog Timers				
WDT#08	Reading the WDTVALUE Register may Return Incorrect Values When Using Watchdog Timer 1	X	X	X

5 Known Design Exceptions to Functional Specifications

Table 3. Advisory List

Table 3. Advisory List (continued)

ADC#09 *First two ADC Samples From the Internal Temperature Sensor Must be Ignored*

Revision(s) Affected: 1, 2, and 3.

Description: The analog source resistance (R_S) to the ADC from the internal temperature sensor exceeds the specified amount of 500Ω. This causes a settling time requirement that is longer than the sampling interval to the converter.

Workaround(s): A small delay must be inserted between samples to allow enough time for the voltage to settle. Configure the respective ADC Sample Sequence n Sample and Hold Time (ADCSSTSH) register for a sample and hold width of at least 16 ADC clocks (TSHx = 0x4).

Alternatively, three consecutive samples from the same channel must be taken to accurately sample the internal temperature sensor using the ADC. The first two consecutive samples should be discarded and the third sample can be kept. These consecutive samples cannot be interrupted by sampling another channel.

ADC#13 *A Glitch can Occur on pin PE3 When Using any ADC Analog Input Channel to Sample*

Revision(s) Affected: 1, 2, and 3.

Description A glitch may occur on PE3 when using any ADC analog input channel (AINx) to sample. This glitch can occur when PE3 is configured as an analog input channel (AINx) and happens at the end of the ADC conversion. These glitches will not affect analog measurements on PE3 when configured as AIN0 as long as the specified source resistance is met.

Workaround(s) A 1kΩ external pull-up or pull-down on PE3 will help to minimize the magnitude of the glitch to 200 mV or less.

ADC#14 *The First two ADC Samples may be Incorrect*

Revision(s) Affected: 1, 2, and 3.

Description The first two ADC samples taken after the ADC clock is enabled in the xCGCADC register may be incorrect.

Workaround(s) Reset the ADC peripheral using the SRADC register after the ADC peripheral clock is enabled and before initializing the ADC and enabling the sample sequencer.

ADC#15 *ADC Global Synchronization Does not Function*

Revision(s) Affected: 1 only.

Description: The SYNCWAIT and GSYNC bits in the ADC Processor Sample Sequence Initiate (ADCPSSI) register are set to allow sample sequencers in ADC0 and ADC1 to be synchronized. These bits do not function.

Workaround(s): None.

ADC#16 *Phase Offset does not Delay as Expected if Sample Sequencers are not Triggered at the Same Time*

Revision(s) Affected: 1 and 2.

Description: The phase difference set in the ADC Sample Phase Control (ADCSPC) register does not reference the same starting point in time if the sequencers are configured for a phase offset and are not triggered at the same time.

Workaround(s): Use the same trigger to ensure that the sample sequencers will trigger at the same time. If using processor trigger and both ADC modules with phase offset, use the GSYNC and SYNCWAIT bits in the ADC Processor Sample Sequence Initiate (ADCPSSI) register to ensure that the trigger occurs simultaneously. The phase offsets will not align if triggering using trigger always mode.

CRC#01 *Any Data Read From a Non-CRC Register After Accessing the CRCRSLTPP Register is Incorrect*

Revision(s) Affected: 1 only.

Description: A CRC Post Processing Result (CRCRSLTPP) register read followed by a read from any other non-CRC register on the AHB results in incorrect data in the non-CRC register.

Workaround(s): Read any CRC register after reading the CRCRSLTPP register and before reading any other register on the AHB. To determine which modules are on the AHB, refer to Figure 1-1 in the data sheet.

DMA#02 *µDMA Data may be Corrupted if Transferred or Received While Entering or Exiting Deep Sleep Mode*

Revision(s) Affected: 1, 2, and 3.

Description: Transferred or received data using the µDMA from either the UART or the SSI peripherals may get corrupted when entering Deep Sleep mode from Run mode or exiting Deep Sleep mode to Run mode if the Run mode clock configuration is not the same as the Deep Sleep mode clock configuration.

Workaround(s): Program the Run mode clock configuration to match the Deep Sleep mode clock configuration right before entering Deep Sleep mode.

ELEC#02 *V_{BAT} Supply pin may be Damaged if the pin Voltage Ramps Faster Than 0.7 V/µs*

Revision(s) Affected: 1, 2, and 3.

Description The V_{BAT} supply pin may be damaged if the pin voltage ramps faster than 0.7 V/µs. Fast V_{BAT} ramps are a concern when a battery is being connected or the V_{BAT} supply is hard switched.

Workaround(s) An RC circuit as shown should be added to the V_{BAT} pin to prevent the damage. The R_1 and C_1 should be placed close to the microcontroller for best protection. In systems that do not require Hibernate when the VDD supply is off, the V_{BAT} pin should be tied to the VDD supply, which typically ramps at a rate slower than 0.7 V/µs. The R1 and C1 components are not required for a V_{BAT} supply ramp less than 0.7 V/µs.

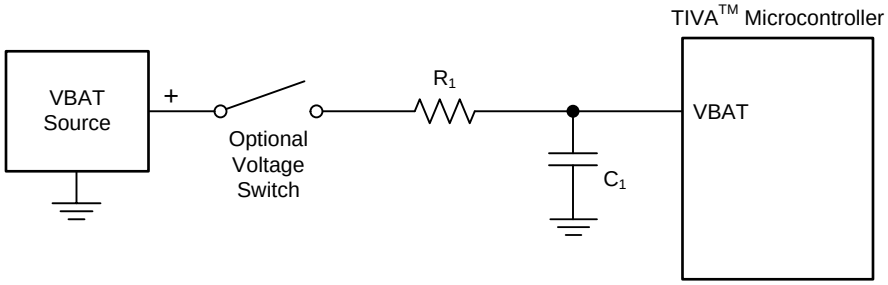


Figure 2. RC Circuit

ELEC#03 *PIOSC Frequency Variance does not Meet +/- 4.5% Across Voltage and Temperature*

Revision(s) Affected: 2 only.

Description: The measured internal 16-MHz precision oscillator (PIOSC) frequency variance across the specified voltage and temperature range when factory calibration is used is +/-6%. This is greater than the +/-4.5% that is specified in the data sheet.

Workaround(s): The PIOSC can be recalibrated at a specific voltage and temperature to get maximum frequency variance of 1% at those conditions.

ELEC#04 *Equation for C_L for HIBXOSC is not Correct*

Revision(s) Affected: 1, 2, and 3.

Description: The Electrical Specification for C_L for HIBXOSC is a function $C_L = (C_1*C_2)/(C_1+C_2) + C_{PKG} + C_{PCB}$ and does not include the C_0 Shunt Capacitance Specified by Crystal Manufactured.

Workaround(s): The correct equation is as follows:

$C_L = (C_1*C_2)/(C_1+C_2) + C_{SHUNT}$

$C_{SHUNT} = C_0 + C_{PKG} + C_{PCB}$

ELEC#06 *PLL VCO Minimum is not Specified*

Revision(s) Affected: 1, 2, and 3.

Description: The PLL VCO minimum value is not specified in the datasheet.

Workaround(s): VCO min value is 100MHz.

ELEC#07 T_{FLASH} and $T_{FLASHLPDS}$ Parameter Incorrect

Revision(s) Affected: 1, 2, and 3.

Description: The data sheet parameters for T_{FLASH} and $T_{FLASHLPDS}$, which is the time to restore the flash to active state from low power state is given to be 5 µs. The correct value is 96 µs.

Workaround(s): None

EPI#01 *Data Reads can be Corrupted when the Code Address Space in the EPI Module is Used*

Revision(s) Affected: 1, 2, and 3.

Description: The external code address space at address 0x1000.0000 is specified for the EPI module using the ECSZ and ECADR fields in the EPI Address Map (EPIADDRMAP). However, data reads can be corrupted when using this address space.

Workaround(s): Code cannot be executed from the 0x1000.0000 address space. The EPI address spaces at 0x6000.0000 and 0x8000.0000 can be used instead.

In addition, when reading from EPI memory mapped to the code address space at 0x1000.0000, replace direct EPI memory reads via pointers with calls to the EPIWorkaroundWordRead(), EPIWorkaroundHWordRead() or EPIWorkaroundByteRead() functions depending on the data size for the read operation. Similarly, when writing to the EPI code address space, replace direct writes with calls to the EPIWorkaroundWordWrite(), EPIWorkaroundHWordWrite() or EPIWorkaroundByteWrite() functions. These APIs are new and can be found in Appendix 2. For Keil, IAR, GCC, and Code Bench, these functions are defined as inline functions in the epi.h file in *C:\ti\TivaWare_C_Series-2.0\driverlib*. For CCS, which doesn't support this structure, these should be added to a new file placed in the *\driverlib* directory called epi_workaround_ccs.s, and this file should be added to the project. Note that the new DriverLib APIs and the CCS file mentioned in Appendix 2 are included in TivaWare release 2.0.1 and later releases.

ETH#01 *The LED Polarity Bit in the Ethernet MAC Clock Configuration Register Does Not Function*

Revision(s) Affected: 1 only.

Description: The LED Polarity bit (POL) in the Ethernet MAC Clock Configuration Register (EMACCC) does not function. The LED outputs coming from the Ethernet PHY are active high and their polarity cannot be configured.

Workaround(s): None.

ETH#02 *Ethernet May not Initialize After System Reset*

Revision(s) Affected: 1, 2, and 3.

Description: When using Ethernet, the Ethernet initialization may complete but would not start correctly after Ethernet initialization.

Workaround(s): Before Ethernet Initialization the Flash Prefetch must be turned OFF by writing 1 to FLASHCONF.FPFOFF. After completing Ethernet Initialization, the user code must turn ON the Flash Prefetch by clearing the FLASHCONF.FPFOFF bit to restore system performance.

ETH#03 *RBIAS Resistor Required Independent of the use of On-Chip Ethernet PHY*

Revision(s) Affected: 1, 2, and 3.

Description: As per the data sheet the acceptable practice for Ethernet PHY enabled parts is a no connect (NC) to the RBIAS pin if on-chip PHY is not used. However the ROM Boot Loaders enable the Ethernet PHY when the Flash is erased for Ethernet PHY parts which causes the ROM Boot Loader to fail. Also if a 25MHz crystal is used without RBIAS resistor, then JTAG may not work.

Workaround(s): A RBIAS resistor is required even if the application does not require the Ethernet PHY. In this case, a 4.7KOhm 10% tolerance resistor can be used in place of 4.87KOhm 1% tolerance resistor between the RBIAS pin and GND.

ETH#04 *SMI Electrical Section Update (1)*

Revision(s) Affected: 1, 2, and 3.

Description: The following table in the Electrical Characteristics for Ethernet Controller AC Characteristics for MII Serial Management the table is incorrect for parameters N20 and N21.

Parameter No.	Parameter	Parameter Name	Min	Nom	Max	Unit
N20	F_{MDC}	EN0MDC frequency	-	-	2.5	MHz
N21	T_{MDIO_DLY}	EN0MDC to EN0MDIO (output) delay time	0	-	150	ns
N22	T_{MDIO_SU}	EN0MDIO (input) to EN0MDC setup time	10	-	-	ns
N23	T_{MDIO_HLD}	EN0MDIO (input) to EN0MDC hold time	10	-	-	ns

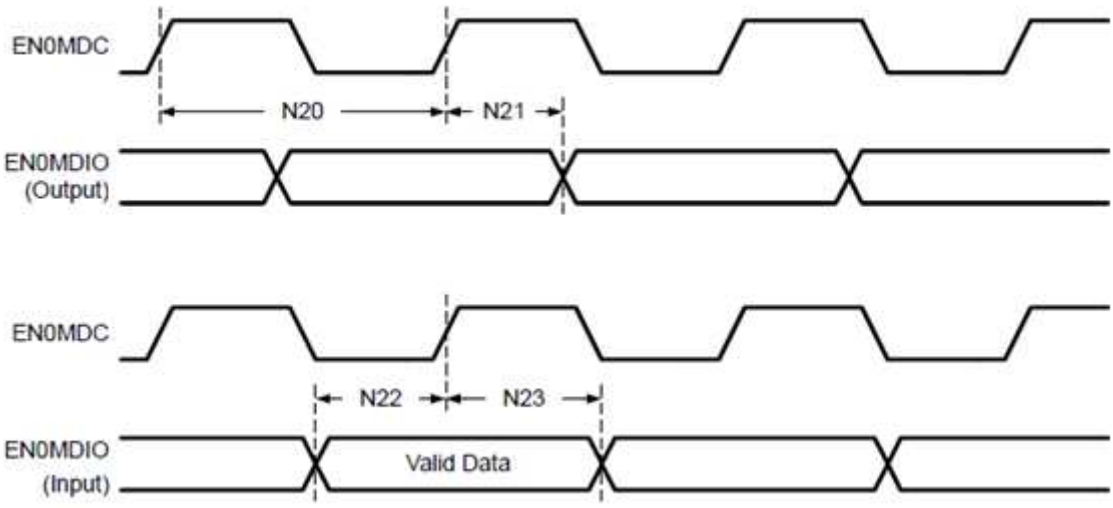
The MDO Timing = (1/(2*Fmdc)) + 1 System Clock Period + 10ns

Workaround(s): None.

ETH#05 *SMI Electrical Section Update (2)*

Revision(s) Affected: 1, 2, and 3.

Description: The following timing diagram in the Electrical Characteristics for Ethernet Controller AC Characteristics for MII Serial Management is the corrected diagram.



Workaround(s): None.

GPIO#03 *GPIO Pins may Glitch on Power up*

Revision(s) Affected: 1 only.

Description: Some devices may drive the GPIOs to ground during power up for less than 5 μs when V_{DDC} is ~ 400-500 mV.

Workaround(s): None.

GPIO#09 *In Some Cases, Noise Injected Into GPIO pins PB0 and PB1 can Cause High Current Draw*

Revision(s) Affected: 1, 2, and 3.

Description: A fast transition on either PB0 or PB1 can switch on a low resistance path between one or both pins and ground potentially causing a high current draw.

This condition has been observed when the signal at the device pin has a rise time or fall time (measured from 10% to 90% of VDD) that is faster than 2 ns. The condition is more likely to occur at high temperatures or in noisy environments. It can occur when the pin is in input or output mode or with any pin multiplexing options.

If the condition is induced while the pin is configured as an output GPIO, then changing the pin state to low and then returning it to a high state at a lower temperature will resolve the condition.

Workaround(s):

1. Do not use PB0 and PB1. Connect both to GND through a 1 kΩ resistor and configure them as GPIO inputs.
2. If PB0 and PB1 are used as USB0ID and USB0VBUS, refer to the USB section of SPMA056, *System Design Guidelines for the TM4C129x Family of Tiva™ C Series Microcontrollers*, and confirm all guidelines contained in the document are followed.

GPTM#09 *General-Purpose Timers do not Synchronize When Configured for RTC Mode*

Revision(s) Affected: 1, 2, and 3.

Description: When attempting to synchronize the General-Purpose Timers using the GPTM Synchronize (GPTMSYNC) register, they do not synchronize if any of the timers are configured for RTC mode. This applies even if the timers are using the alternate clock.

Workaround(s): None.

GPTM#12 *The GPTMPP Register Does not Correctly Indicate Alternate Clock Capability*

Revision(s) Affected: 1 only.

Description The ALTCLK bit in the GPTM Peripheral Properties (GPTMPP) register reads as 0x0, the ALTCLK source is available to the timer. When the ALTCLK bit reads as 0x1, the ALTCLK source is not available to the timer. This is opposite from the intended implementation of these bits. The ALTCLK bit should read as 0x0 when the ALTCLK source is not available for use by the timer and the ALTCLK bit should read as 0x1 when the ALTCLK source is available for use by the time.

Workaround(s) None.

GPTM#13 *Reading the GPTMTnV, the GPTMTnR, or the GPTMRTCPD Registers may Return Incorrect Values When Using an Alternate Clock Source*

Revision(s) Affected: 1, 2, and 3.

Description Incorrect values may be read from the GPTM Timer n Value (GPTMTnV), the GPTM Timer n (GPTMTnR), and the GPTM RTC Predivide (GPTMRTCPD) registers when using an alternate clock source.

Workaround(s) None.

GPTM#14 *General-Purpose Timer ADC and µDMA Triggers may not be Captured in Certain Modes When Using PIOSC*

Revision(s) Affected: 1 only.

Description An ADC or µDMA event that is triggered by the general-purpose timer may not be triggered when the timer is configured to use the alternate clock and the system clock frequency is greater than 17 MHz. This affects the following counting modes:

Mode	Direction	Trigger Affected
Edge Count	Down	Capture Match
One-shot	Up	Time Out
One-shot	Down	Time Out

Workaround(s) None.

GPTM#15 *Counter Does not Immediately Reset to 0 When MATCH is Reached In Edge Count Up Mode*

Revision(s) Affected: 1, 2, and 3.

Description When configured for input edge count mode and count up mode, after counting to the match value, the counter uses one additional edge to reset the timer to 0. As a result, after the first match event, all subsequent match events occur after the programmed number of edge events plus one.

Workaround(s) In software, account for one additional edge in the programmed edge count after the first match interrupt is received.

GPTM#16 *Special Configuration is Required when Operating the GPTM in 32-bit Mode with the Alternate Clock Source*

Revision(s) Affected: 1, 2, and 3.

Description: If the alternate clock source is selected and the timer is configured for 32-bit mode and either one-shot or periodic mode, the GPTM Timer n Match (GPTMTnMATCHR) and GPTM Timer n Interval Load (GPTMTnILR) registers are not writable. As a result, the alternate clock source should not be used when operating the general-purpose timer in these configurations.

Workaround(s): The system clock should be used as the clock source for the general-purpose timer in one-shot or periodic 32-bit modes.

GPTM#17 *The GPTMSYNC Register Bits Must be Manually Cleared when Using an Alternate Clock Source*

Revision(s) Affected: 1 only.

Description: If an alternate clock source is configured in the Alternate Clock Configuration (ALTCLKCFG) register and enabled using the GPTM Clock Configuration (GPTMCC) register, the bits in the GPTM Synchronize (GPTMSYNC) register are not cleared automatically after being set.

Workaround(s): When using the bits in the GPTMSYNC register, software must clear the bits prior to setting them for a subsequent update.

HIB#10 *If MEMCLR is set to a Non-Zero Value, a Tamper Event may not Clear all of the Bits in the HIBDATA Register*

Revision(s) Affected: 1, 2, and 3.

Description: If the MEMCLR bit field in the HIB Tamper Control (HIBTPCTL) register is set to a non-zero value, the Hibernation Data (HIBDATA) register may not clear the specified bits. The MEMCLR bit field provides the option to clear all, the upper half, lower half, or none of the Hibernate memory on a tamper event.

Workaround(s): After clearing the tamper event by setting the TPCLR bit, the application should clear the data in the Hibernate memory in the HIBDATA register (write either the upper half, the lower half, or all of the bits to all zeros).

HIB#12 *Tamper Logging Failure on XOSC Fail Event*

Revision(s) Affected: 1 only.

Description: When XOSC failure is created in the Tamper module, the XOSC bit does not get set in HIBTPLOG1, HIBTPLOG3, HIBTPLOG5, or HIBTPLOG7. The XOSC fail can be a result of shorting or grounding one of the XOSC pins or by removing the 32.768-kHz oscillator source. The user will know the XOSC fail has occurred by reading the XOSCFAIL bit in the HIBTPSTAT register. The user may not be able to correlate with an RTC time stamp.

Workaround(s): None.

HIB#13 *Tamper Events may be Missed in log*

Revision(s) Affected: 1 only.

Description: The HIB Tamper module log captures up to four events. Events 1, 2, 3 and the last event are captured. However, events 4 through the last event minus one are overwritten. If an event occurs after the 3rd and then de-asserts before the last entry, the event will be lost.

Workaround(s): Systems that have VDD available can be configured to wake up on a tamper event and clear the log.

HIB#15 *The VDDFAIL Interrupt bit in the HIBIC Register is not Properly Cleared*

Revision(s) Affected: 1 only.

Description: The VDDFAIL bit in the Hibernation Interrupt Clear (HIBIC) register should be cleared by writing a 1 to bit 7 of the register, but it is not cleared.

Workaround(s): To clear the VDDFAIL bit, write a 1 to both bit 7 and the reserved bit 1 of the HIBIC register.

HIB#16 *Application Code May Miss New Tamper Event During Clear*

Revision(s) Affected: 1, 2, and 3.

Description: During the clear of a tamper event, a new tamper event could be missed or the tamper log could be corrupted. The clear of a tamper event starts with the Tamper Clear (TPCLR) bit in the HIB Tamper Control register (HIBTPCTL) being written. The write takes 3 rising edges of the 32.768-kHz clock to complete the clear.

Workaround(s): To prevent missing a tamper event during these three Hibernate clock cycles and restoring the tamper log to its reset state, workaround code must be implemented in the NMI handler as shown in Appendix 1. The new DriverLib APIs mentioned in Appendix 1 are included in TivaWare release 2.0.1 and later releases.

HIB#17 *$\overline{WAKE}$ Cannot be Used to Wake From Hibernate Mode*

Revision(s) Affected: 1 only. Applicable to devices with date codes before 0x38 (August, 2013). See Section 3 on how to read the date code.

Description: The external $\overline{WAKE}$ pin cannot be used to wake from Hibernate mode. In addition, the WAKENC bit in the Hibernation Peripheral Properties (HIBPP) register that indicates the presence of the $\overline{WAKE}$ pin is clear.

Workaround(s): Use any of the following methods of waking from Hibernate mode:

- RTC match wake event
- Low battery wake event
- External $\overline{RST}$
- GPIO K[7:4]
- Tamper TMPR[3:0]

HIB#18 *Can get two Matches per day in Calendar Mode*

Revision(s) Affected: 1, 2, and 3.

Description: When the CAL24 bit in the Hibernation Calendar Control (HIBCALCTL) register is clear, the RTC counts in 12 hour, AM/PM mode. The AM/PM bit in the Hibernation Calendar Match 0 (HIBCALM0) specifies whether the match should occur in the AM or the PM. However, this bit is ignored when determining if a match is occurring. As a result, an RTC match could occur twice in one day.

Workaround(s): Adjust the match time to 24 hour mode before configuring the HIBCALM0 register and set the CAL24 bit. Alternatively, when the match occurs, check the AM/PM bit in the Hibernation Calendar (HICAL0) register to determine if the match is correct.

HIB#19 *The First Write to the HIBCTL Register may not Complete Successfully After a Hibernation Module Reset*

Revision(s) Affected: 1, 2, and 3.

Description: The initial write to the HIBCTL register may not occur if the Hibernation module is reset. The WRC bit in the HIBCTL register may not be set.

Workaround(s): After a Hibernation module reset, check to see if the WRC bit is set and perform the following:

- If the WRC bit is not set within the maximum oscillator startup time, perform a software reset of the Hibernation module and retry the HIBCTL write. The maximum oscillator startup time is given by the Hibernation XOSC startup time parameter, TSTART, in the Hibernation External Oscillator (XOSC) Input Characteristics table in the Electrical Characteristics chapter of the data sheet.
- If the WRC bit is set but the HIBCTL write was not successful, retry the HIBCTL write.

I2C#05 *Register Read of I2CMCS may Cause an ADDR or DATA NAK to be Missed*

Revision(s) Affected: 1, 2, and 3.

Description: A read of the I2CMCS returns the Status of the I2C Master including the error bits for ADR and DATA NAK. If the bits are updated for a NAK on the same system clock as the read of the I2CMCS register, then the read will clear the bits and return an ACK status to the CPU. Also the interrupt bit for NAK would not be set.

Workaround(s): Instead of polling the I2CMCS the application code must use only the Interrupt method for communicating with external slaves.

I2C#08 *I2C Master BUSY Status bit does not get set Immediately*

Revision(s) Affected: 1, 2, and 3.

Description: The I2C master register I2CMCS provides the status information on the I2C transaction. In polling mode the application checks if the BUSY status bit is cleared before issuing the next I2C transaction command. However, it takes approximately 60% of the I2C clock period for the BUSY status bit to be set after the command is issued.

Workaround(s):

- Use interrupt to perform data transfer instead of polling the BUSY status bit in I2CMCS register.

  OR

- Poll bit 0 of I2CMRIS register to be set to indicate the completion of the last transaction command, then clear it by writing 1 to I2CMICR before sending the next transaction command.

  OR

- Add a delay loop of at least 60% of the I2C clock period before polling the I2CMCS register BUSY status bit.

LCD#01 *LIDD-Mode DMA Transactions in the LCD Controller Cause the Microcontroller to Become Unresponsive*

Revision(s) Affected: 1, 2, and 3.

Description: Whenever a LIDD-mode DMA transaction is requested by setting the DMAEN bit of the LCD LIDD Control (LCDLIDDCTL) register, after setting the LCD DMA frame buffer n base and ceiling addresses in the LCDDMABAFB and LCDDMACAFB registers, respectively, the next attempt to access any LCD controller register causes the microcontroller and the JTAG connection to become unresponsive. A power-on reset recovers the device.

Workaround(s): The following code must be inserted prior to each new LCD DMA transaction. The code resets the LCD DMA engine. Additional code around the LCDClockReset() function call is required when using LIDD mode. This code temporarily configures the CS signal to the display as a GPIO output to ensure that it remains high during the LCD DMA reset operation. Without this GPIO code, a low glitch will likely occur on the CS signal, which may cause the attached display to operate incorrectly.

```
//
// If using LIDD mode, revert the CS pin to GPIO control.
// Configure the CS pin as a GPIO output, drive the pin high, and
// clear the respective alternate function register bit. This is
// done to prevent glitches on CS during the time the LCD
// controller is reset after each transaction. This step is not
// needed for Raster mode.
//
GPIOPinTypeGPIOOutput(GPIO_PORTJ_BASE, 0x40);
GPIOPinWrite(GPIO_PORTJ_BASE, 0x40, 0x40);
HWREG(GPIO_PORTJ_BASE + GPIO_O_AFSEL) &= ~0x40;
//
// Reset the module (but preserve all register values)
//
LCDClockReset(LCD0_BASE, LCD_CLOCK_MAIN);
//
// If using LIDD mode, set the CS pin back to hardware control.
// This step is not needed for Raster mode.
//
HWREG(GPIO_PORTJ_BASE + GPIO_O_AFSEL) |= 0x40;
```

LCD#02 *LCD DMA FIFO Underflow Interrupt Occurs When EPI is Mapped to an External SDRAM With an Address That is not 0x1000.0000*

Revision(s) Affected: 1, 2, and 3.

Description: If the EPI controller is mapped to allow indirect access to SDRAM where the ECADR bit field is 0x0 and the ERADR bit field is not 0x0 in the EPI Address Map (EPIADDRMAP) register, a DMA FIFO underflow interrupt occurs. This is a result of the LCD being a lower priority than the CPU when they both try to access the SDRAM.

Workaround(s): Map the EPI to use the external code area 0x1000.0000 (ECADR = 0x1). This makes the LCD higher priority than the CPU and prevents this FIFO underflow condition.

LCD#03 *LCD Module Does not Restart if an Underflow Occurs*

Revision(s) Affected: 1, 2, and 3.

Description: The LCD module does not restart if a DMA FIFO underflow interrupt occurs. Writes to the UFLOWRST bit in the LCD Control (LCDCTL) register have no effect.

Workaround(s): If a DMA FIFO underflow interrupt occurs (the FIFOU interrupt bit is set), restart the LCD Raster controller by calling LCDRasterEnable().

MEM#03 *EEPROM Data May be Corrupted if an EEPROM Write is Interrupted*

Revision(s) Affected: 1 and 2.

Description: Corrupted EEPROM data can occur if an EEPROM write is interrupted with any of the following power events:

- Power failure
- External reset ($\overline{\text{RST}}$) (if configured for a simulated POR sequence in the RESBEHAVCTL register (EXTRES = 0x3))
- Brown-out (BOR) event (if configured for a simulated POR sequence in the RESBEHAVCTL register (BOR = 0x3))
- Watchdog reset (if configured for a simulated POR sequence in the RESBEHAVCTL register (WDOGn = 0x3))

The corrupted EEPROM data that can result from this sequence is not limited to the current word being written. If these events do not apply to your system, then normal EEPROM operation is expected. If a failure occurs, there will not be any indication of the failed erase or corrupted data (for example in the PRETRY and the ERETRY bits in the EEPROM Support Control and Status (EESUPP) register.

Workaround(s): Depending on the system, there are a few potential workarounds:

1. Program the EEPROM only when the device is guaranteed to not have power removed and when a brown-out reset and an external reset will not occur. There are no restrictions on EEPROM reads.
2. Use the Flash memory with application software to store data instead of the EEPROM controller.
3. Limit the number of lifetime EEPROM writes to 7 writes per word.
4. Use an external EEPROM.

MEM#07 *Soft Resets Should not be Asserted During EEPROM Operations*

Revision(s) Affected: 3 only.

Description: EEPROM data may be corrupted if any of the following soft resets are asserted during an EEPROM program or erase operation:

- Software reset (SYSRESREQ)
- Software peripheral reset of the EEPROM module
- Watchdog reset (if configured as a system reset in the RESBEHAVCTL register)
- MOSC failure reset
- BOR reset (if configured as a system reset in the RESBEHAVCTL register)
- External reset (if configured as a system reset in the RESBEHAVCTL register)
- Writes to the HSSR register

Workaround(s): Ensure that any of the above soft resets are not asserted during an EEPROM program or erase operation. The WORKING bit of the EEDONE register can be checked before the reset is asserted to see if an EEPROM program or erase operation is occurring. Soft resets may occur when using a debugger and should be avoided during an EEPROM operation. A reset such as the Watchdog reset can be mapped to an external reset using a GPIO or Hibernate can be entered, if time is not a concern.

MEM#09 *ROM_SysCtlClockFreqSet() Does not Properly Configure MOSC*

Revision(s) Affected: 1 only.

Description: The ROM_SysCtlClockFreqSet() function does not properly configure the MOSC. Due to this erratum, any ROM functions that rely on the proper operation of the MOSC, such as the Ethernet boot loader, will not function correctly.

Workaround(s): Use the TivaWare function of SysCtlClockFreqSet() in Flash memory.

MEM#11 *The ROM Version of the TivaWare EEPROMInit API Does not Correctly Initialize the EEPROM*

Revision(s) Affected: 3 only.

Description: The ROM_EEPROMInit API in TivaWare does not correctly initialize the EEPROM module as described in the data sheet. It should not be used to initialize the EEPROM.

Workaround(s): Use the Flash version of the EEPROMInit API in TivaWare version 2.1 or later.

MEM#12 *Code Jumps from Flash to ROM when EEPROM is Active may Never Return*

Revision(s) Affected: 1 and 2.

Description: ROM function calls may never return when the EEPROM is active (when the WORKING bit in the EEPROM Done Status (EEDONE) register is set or when an EEPROM register is read) and the user application is executing from Flash memory.

Workaround(s): Replace ROM functions with the Flash-based equivalent functions.

MEM#13 *Concurrent µDMA Reads from Flash Memory and EEPROM Accesses May Fetch Incorrect µDMA Data*

Revision(s) Affected: 1 and 2.

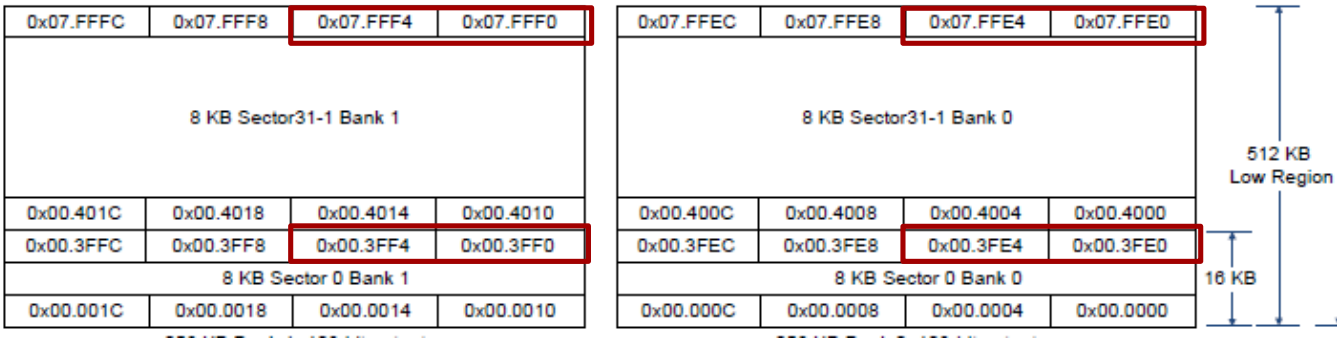
Description: The µDMA data fetched from the Flash memory may be incorrect if EEPROM is active while µDMA is reading the Flash memory.

Workaround(s): Disable the µDMA for Flash memory operations before enabling EEPROM or accessing EEPROM registers.

MEM#15 *Specific Flash Locations in any Sector do not get Erased*

Revision(s) Affected: 1, 2, and 3.

Description: If only one or both of the first two words of the last line of a sector in a bank are programmed, an Erase of entire Sector, Mass Erase of device, or toggle mass erase does not erase the flash back to all 1's. The diagram below shows the affected words in Sector 0 and Sector 31 of the lower 512KB of Flash.



Workaround(s): Program any other word in the Sector-Bank to allow Sector Erase to work.

NOTE: Toggle Mass Erase or Mass Erase will still not work.

MEM#16 *JTAG Unlock Issue when BOOTCFG is Committed with Debug Disabled*

Revision(s) Affected: 1, 2, and 3.

Description: After configuring the BOOTCFG register with NW=0, DBG1=0 and DBG=1, the JTAG Debugger access is locked and cannot be unlocked by one Unlock Sequence Run.

Workaround(s): The Unlock Sequence has to be run twice for the device to be unlocked.

MEM#17 *Clearing Reserved Bits of BOOTCFG Causes ROM Boot Loader to Fail*

Revision(s) Affected: 1, 2, and 3.

Description: After clearing the BOOTCFG register reserved bits the ROM Boot Loader will load the application code but not execute.

Workaround(s): The User Code has to ensure that the Reserved bits of BOOTCFG are always set to ‘1’.

MEM#18 *Only Lower 512KB Flash can be Protected on 1MB Devices*

Revision(s) Affected: 1, 2, and 3.

Description: The TM4C129 device provides user committable registers for 16 flash memory protection registers out of which FMPRE8-15, FMPPE8-15 do not get committed after a Power Cycle. As a result only the lower 512KB can be protected on 1MB devices.

Workaround(s): The Application Code has to re-commit the registers FMPRE8-15 and FMPPE8-15 every time.

MEM#19 *Certain GPIOs Cannot be Configured as boot pins*

Revision(s) Affected: 1, 2, and 3.

Description: Pins PC0-3, PD7 and PE7 cannot be selected as boot pins. These pins are locked on the TM4C129x devices, and the ROM does not unlock them before checking for polarity. As a result, the devices may not go into ROM boot loader or may remain in ROM boot loader on a POR reset.

Workaround(s): Use other GPIOs as boot pin. Refer to the BOOTCFG register description for details.

MEM#20 *Setting Mirror mode bit Causes Bus Faults on 512KB Flash*

Revision(s) Affected: 1, 2, and 3.

Description: The mirror mode feature allows the swap of lower half of the flash address space with the upper half. On the devices with 512KB flash, when the user sets the FLASHCONF.FMME bit to enable the mirror mode, bus fault will be generated for any access to the upper half of the flash address space.

Workaround(s): None.

ONEWIRE#01 *A Delay is Needed for the 1-Wire µDMA Receive Configuration*

Revision(s) Affected: 1 only.

Description: A 1-Wire µDMA receive configuration includes a write to the DMAOP field of the ONEWIREDMA register followed by a write to the ONEWIREDATW register with the value 0xFFFF.FFFF to prime the read operations. If a sufficient delay is not inserted between these two instructions, the write to the ONEWIREDATW register is not recognized and the read does not start.

Workaround(s): After writing to the DMAOP field in the ONEWIREDMA register, insert a delay of at least 187.5 ns (3 PIOSC clock cycles) before writing the ONEWIREDATW register.

PWM#01 *Under Certain Circumstances, the PWM Load Interrupt is Triggered as Soon as the PWM is Enabled*

Revision(s) Affected: 1 only.

Description: A spurious PWM interrupt occurs immediately when the PWM is enabled under the following conditions:

- The PWM Load register contains a nonzero value and
- Either of the PWM Compare registers contains a value less than the value in the PWM Load register and
- PWM interrupts are enabled.

Workaround(s): None.

PWM#02 *Setting the PWMSYNC Bits May Not Synchronize the PWM Counters if PWMDIV is Used*

Revision(s) Affected: 1 only.

Description: The bits in the PWM Time Base Sync (PWMSYNC) register are used to synchronize the counters in the PWM generators. The PWMDIV field in the PWM Clock Configuration (PWMCC) register is used to specify a fractional version of the system clock to use for the counters. If the PWMSYNC bits are set when the PWMDIV field is configured to anything other than 0x0, the counters may not be synchronized.

Workaround(s): None.

PWM#03 *The PWM Generators May Not Generate Interrupts or ADC Triggers*

Revision(s) Affected: 1 only.

The PWM Generators May Not Generate Interrupts or ADC Triggers

Description: The PWM generators do not reliably generate interrupts or ADC triggers.

Workaround(s): None.

PWM#04 *PWM Generator Interrupts can only be Cleared 1 PWM Clock Cycle After the Interrupt Occurs*

Revision(s) Affected: 1, 2, and 3.

Description: A write of 1 to the PWMxISC register is expected to clear the corresponding generator interrupt status on the next system clock. However, the write will clear the generator interrupt status on the next PWM clock. Any write to the PWMxISC to clear the interrupt before the next PWM clock will be ignored and the interrupt will be re-asserted.

Workaround(s): After the interrupt is asserted, the CPU must wait for one PWM clock cycle before writing 1 to the PWMxISC to clear the corresponding generator interrupt status. The larger the PWM clock divider value, the longer the system delay to clear the interrupt.

PWM#05 *Generator Load with Global Sync May Lead to Erroneous Pulse Width*

Revision(s) Affected 1, 2 and 3.

Description: There is a condition where the generator timer may still count with the old value, but the new comparator value gets loaded causing an erroneous high pulse width.

Workaround(s): The application must use the interrupt status to write the value of the new load and comparator values. When using the down count mode, clear the raw interrupt status bit for the respective comparator down count match interrupt status bit, wait for it to be set again and then update the new value for the load and comparator match.

PWM#06 *PWM Output May Generate a Continuous High Instead of a Low or a Continuous Low Instead of High*

Revision(s) Affected

1, 2 and 3.

Description

When using PWM in UP-DOWN count mode, the PWM generator may generate a continuous High instead of Low or continuous Low instead of High under the following condition:

- The PWM generator is used with action for comparator A or B UP and DOWN to control the PWM cycle.
- The PWM comparator A or B is changed from a count of 2 or more to PWM Load Count value.
- A continuous High is generated if the invert option is enabled.
- A continuous Low is generated if the invert option is disabled.

Workaround(s)

- Use DOWN count mode of PWM
- When changing from any value (other than 1) to the PWM load count value in the Comparator Load Register, always go to value of 1 before going to the PWM load count value.

QEI#01 *When Using the Index Pulse to Reset the Counter, a Specific Initial Condition in the QEI Module Causes the Direction for the First Count to be Misread*

Revision(s) Affected: 1, 2, and 3.

Description: When using the index pulse to reset the counter with the following configuration in the QEI Control (QEICTL) register:

- SIGMODE is 0 indicating quadrature mode
- CAPMODE is 1 indicating both PhA and PhB edges are counted

and the following initial conditions:

- Both PhA and PhB are 0
- The next quadrature state is in the counterclockwise direction

the QEI interprets the state change as an update in the clockwise direction, which results in a position mismatch of 2.

Workaround(s): None.

SSI#03 *SSI1 can Only be Used in Legacy Mode*

Revision(s) Affected: 1 (all modules). 2 and 3 (only SSI1).

Description: Bi-, quad-, and advance-modes of operation do not function correctly on the specified SSI module(s). As a result, any affected module can only be used for legacy operation.

Workaround(s): On revision 2 devices, use SSI0, SSI2, or SSI3 for bi-, quad-, and advance-mode operation. Only use SSI1 for legacy-mode operation.

SSI#04 *The First Byte Sent by the SSI in Master Mode is Incorrect when Using the Alternate Clock*

Revision(s) Affected: 1 only.

Description: When the alternate clock source is selected by setting the ALTCLK bit in the SSI Clock Configuration (SSICC) register and the SSI is in master mode, the first byte transmitted is incorrect.

Workaround(s): Use the system clock for the QSSI when in master mode.

SSI#05 *Bus Contention in Bi- and Quad-Mode of SSI*

Revision(s) Affected: 1, 2, and 3.

Description: When the SSI is configured in Bi- or Quad-mode, and a read from external memory is performed after the SSI is configured for Receive Mode, bus contention can occur on the SSI data pins on that first data read.

Workaround(s): Perform a dummy read from memory before the first valid read operation after configuring the SSI for Receive Mode (setting the DIR bit in the QSSI Control (SSICR1) register). For example:

```
SSIConfigSetExpClk(SSI0_BASE,SysCtlClockFreqSet (),
SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER, 1000000, 8);
SSIAdvModeSet(SSI0_BASE, SSI_ADV_BI_READ); //Receive Mode set
SSIDataPut(SSI0_BASE, &pui32DataRx[ui32Index]); //intentional dummy write
SSIDataGetNonBlocking(SSI-_BASE, ui32Dummy); //dummy read
SSIDataPut(SSI0_BASE, &pui32DataRx[ui32Index]); //intentional dummy write
SSIDataGetNonBlocking(SSI0_BASE, &pui32DataRx[0])); //first intentional read
SSIDataPut(SSI0_BASE, &pui32DataRx[ui32Index]); //intentional dummy write
SSIDataGetNonBlocking(SSI0_BASE, &pui32DataRx[1])); //second intentional read
```

If the transfer normally requires any dummy operations, such as the intentional dummy writes shown above, the dummy read should occur before the normal dummy operations.

Note that if your application is sensitive to the SSIClk, the dummy read outputs a clock cycle. Reconfigure the SSIClk pin to a GPIO input while performing the dummy read to prevent this from affecting your clock-sensitive application.

SSI#06 *SSI Receive FIFO Time-out Interrupt may Assert Sooner than Expected in Slave Mode*

Revision(s) Affected: 1, 2, and 3.

Description: The SSI receive FIFO time-out interrupt may assert sooner than 32 system clock periods in slave mode if the CPSDVSR field in the SSI Clock Prescale (SSICPSR) register is set to a value greater than 0x2. Master mode is not affected by this behavior.

Workaround(s): In some cases, software can use the SCR field in the SSI Control 0 (SSICR0) register in combination with a CPSDVSR field value of 0x2 to attain the same SSI clock frequency. For example, if the desired serial clock rate is SysClk/48, then CPSDVSR = 0x2 and SCR = 0x17 can be used instead of CPSDVSR = 0x18 and SCR = 0x1 to achieve the same clock rate, using the equation SSInCLK = SysClk / (CPSDVSR * (1 + SCR)). If there is not a value of SCR that can be used with CPSDVSR = 0x2 to attain the required serial clock rate, then the receive FIFO time-out feature cannot be used.

SSI#07 *SSI Transmit Interrupt Status Bit is not Latched*

Revision(s) Affected: 1, 2, and 3.

Description: SSI Transmit Interrupt Status Bit does not work correctly.

- Condition-1: Master Mode with interrupts when transmit FIFO is half full or less (SSICR1.EOT bit is clear). SSIMIS will be asserted every time the TXFIFO drops below half FIFO threshold causing the interrupt to be asserted all the time even if SSIICR register is used to clear the Interrupt condition
- Condition-2: Master mode with interrupts when transmit FIFO is empty. (SSICR1.EOT bit is set) SSITXRIS Is not latched when the transmit buffer is empty. SSIMIS and SSIRIS registers may be read as 0 by the CPU.

Workaround(s): SSI must be initialized with SSICR1.EOT bit clear (Condition-1). In the interrupt handler on completion of the transfer from the buffer to SSIDR clear the SSIIM.TXIM to stop any further interrupts. When the next set of interrupts are required for transmission then set the SSIIM.TXIM bit.

SSI#08 *SSI Slave in Bi and Quad Mode swaps XDAT0 and XDAT1*

Revision(s) Affected: 1, 2, and 3.

Description: The SSI Slave when in Bi or Quad Mode sends the data on XDAT0 to XDAT1 line and XDAT1 to XDAT0 line.

Workaround(s): When transmitting the data from SSI Slave in Bi and Quad Mode the CPU must swap the bits when writing to the Data Register SSIDR for the correct bit to be transmitted. When using uDMA CPU must write the swapped bits in the uDMA's source buffer.

SYSCTL#03 *The MOSC Verification Circuit Does not Detect a Loss of Clock After the Clock has been Successfully Operating*

Revision(s) Affected: 1, 2, and 3.

Description: If the MOSC clock source has been powered up and operating correctly and is subsequently removed or flatlines, the MOSC verification circuit does not indicate an error condition.

Workaround(s): Use Watchdog module 1, which runs off of PIOSC, to reset the system if the MOSC fails.

SYSCTL#09 *Some Devices may not Start Properly During Power up*

Revision(s) Affected: 1 only.

Description: In very rare cases, the internal LDOs may not start properly during power up. If the LDOs do not start properly, the device may not begin operating, and VDDC may not reach its specified levels.

Workaround(s): Power cycle the device until the device starts up correctly. This issue has not been seen on devices when the VDD rise time from 0 V to 3.0 V is less than 100 us. However, meeting this condition does not guarantee that the issue will not occur.

SYSCTL#12 *MOSC Does not Power Down in Deep Sleep when it is not the Deep Sleep Clock Source*

Revision(s) Affected: 1 only.

Description: The main oscillator (MOSC) continues to run in Deep Sleep mode if it was used as the system clock source for Run mode. The Deep Sleep clock source programmed in the DSCLKCFG register is still applied in Deep Sleep, though the current consumption is about 350 µA higher in Deep Sleep mode than it would be if the MOSC was disabled.

Workaround(s): Turn off the main oscillator before entering Deep Sleep mode by switching to an alternate clock source and then setting the PWRDN bit in the Main Oscillator Control (MOSCCTL) register.

SYSCTL#13 *The NMIC Register Does not Indicate NMI Sources when Read*

Revision(s) Affected: 1 only.

Description: The NMI Cause Register (NMIC) register should indicate which possible NMI source caused an NMI - MOSCFAIL, Tamper, WDT, BOR or external NMI signal. However, this register does not function correctly.

Workaround(s): Use alternate sources to determine which possible event caused the NMI:

- MOSCFAIL – check the MOFRIS bit in the Raw Interrupt Status (RIS) register
- Tamper – check the STATE field in the HIB Tamper Status (HIBTPSTAT) register
- WDT – check the WDTRIS bit in the Watchdog Raw Interrupt Status (WDTRIS) register
- BOR – check the BORRIS bit in the Raw Interrupt Status (RIS) register
- External signal – configure the GPIO interrupt registers to trigger an interrupt when the NMI pin is asserted and check the GPIO Raw Interrupt Status (GPIORIS) register

SYSCTL#14 *Power Consumption is Higher When MOSC is Used in Single-Ended Mode*

Revision(s) Affected: 1 only.

Description: The MOSC internal oscillator continues to run, even when a single-ended clock source is attached to OSC0. This issue does not affect proper operation but does result in additional power consumption of up to 3.5 mA.

Workaround(s): None

SYSCTL#15 *Watchdog Reset Improperly Updates the NMIC Register*

Revision(s) Affected: 1 only.

Description: When the watchdog expires and causes a reset, the WDT bit in the NMI Cause (NMIC) register is set, and it should not be. Note that the Reset Cause Register (RESC) is appropriately updated to indicate the watchdog reset.

Workaround(s): After a watchdog reset occurs, manually clear the WDT bit in the NMIC register to remove the erroneous NMI indication.

SYSCTL#16 *On-Chip LDO may not Start Properly During Power Up*

Revision(s) Affected: 1 and 2.

Description: In very rare cases, a non-monotonic voltage rise of VDDA between the minimum and maximum Power-On Reset Threshold (V_{POR}) voltage range, 2.0 V and 2.55 V, can cause the on-chip LDO to not start up. Because the LDO controls the core voltage (VDDC), the device cannot start up correctly in this situation. If the LDO fails to start, power cycle the device until a successful power up occurs. A software or hardware reset cannot restart the LDO.

Workaround(s): A monotonic voltage rise of VDDA prevents this issue from occurring; however, a perfect monotonic ramp is difficult to achieve, particularly during LDO inrush. The risk of encountering this issue can be minimized by performing one of the following:

- If the VDD and VDDA pins are connected directly to the same power source, at every possible point A and point B along the VDDA waveform between 2.0 V and 2.55 V, point B must never fall below point A after 15 µs, as shown in Figure 3.
- Use a separate power supply for VDDA to reduce noise and isolate it from the effects of LDO inrush. At every possible point A and point B along the VDDA waveform between 2.0 V and 2.55 V, point B must never fall below point A after 15 µs, as shown in Figure 3. The separate power supply will make it easier to avoid this condition.

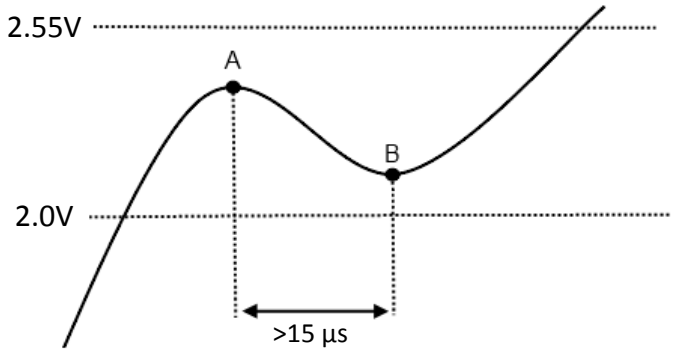


Figure 3. VDDA Waveform to Avoid Between V_{POR} min and V_{POR} max

SYSCTL#18 *DIVSCLK Outputs a Different Clock Frequency than Expected when DIV = 0x0*

Revision(s) Affected: 1, 2, and 3.

Description: In the Divisor and Source Clock Configuration (DIVSCLK) register, if the DIV bit field is 0x0 (divided by 1), the clock output to the GPIO is not what is expected:

- If the DIV bit field has not yet been adjusted in code, the clock frequency will be the clock source defined in the SRC bit field divided by 32. For example, if SRC = 0x1 (PIOSC) and DIV = 0x0, the clock output to the GPIO will have a frequency of 500 kHz, derived from the PIOSC.
- If the DIV bit field has already been adjusted in code, the clock frequency will be the clock source defined in the SRC bit field divided by the previous DIV bit field value. For example, if SRC = 0x1 (PIOSC) and DIV was previously 0x1 (divided by 2) then written to 0x0, the clock output to the GPIO will have a frequency of 8 MHz, derived from the PIOSC.

Workaround(s): If clock accuracy of the source is not a factor, certain frequencies can be achieved using a non-zero DIV value and a different SRC value. For example, to achieve a 16 MHz clock, instead of SRC = 0x1 (PIOSC) and DIV = 0x0, use SRC = 0x0 (System Clock) and the respective DIV value (DIV = 0x4 (divided by 5) for a system clock of 80 MHz).

SYSCTL#19 *Extra Steps Required to Unlock the Device if NW = 0*

Revision(s) Affected: 1, 2, and 3.

Description: If the NW bit in the Boot Configuration (BOOTCFG) register is clear when the microcontroller gets “locked”, the debug port unlock sequence, as described in the data sheet, does not recover the microcontroller.

Workaround(s): The debug port unlock sequence, needs to be performed twice to unlock the device.

SYSCTL#22 *Change to the PLL Clock Divider may Cause System Clock to be out of Specification*

Revision(s) Affected: 1, 2, and 3.

Description: When generating system clock from the PLL, the user programs the system control register bits RSCLKCFG.PSYSDIV to divide the PLL output. This register value may not be loaded into the physical divider causing the system clock to be divided by 2. This condition will cause the system clock to be out of specification.

The issue will be caused if ROM_SysCtlClockFreqSet or SysCtlClockFreqSet API provided in TivaWare 2.1.2 or earlier is used.

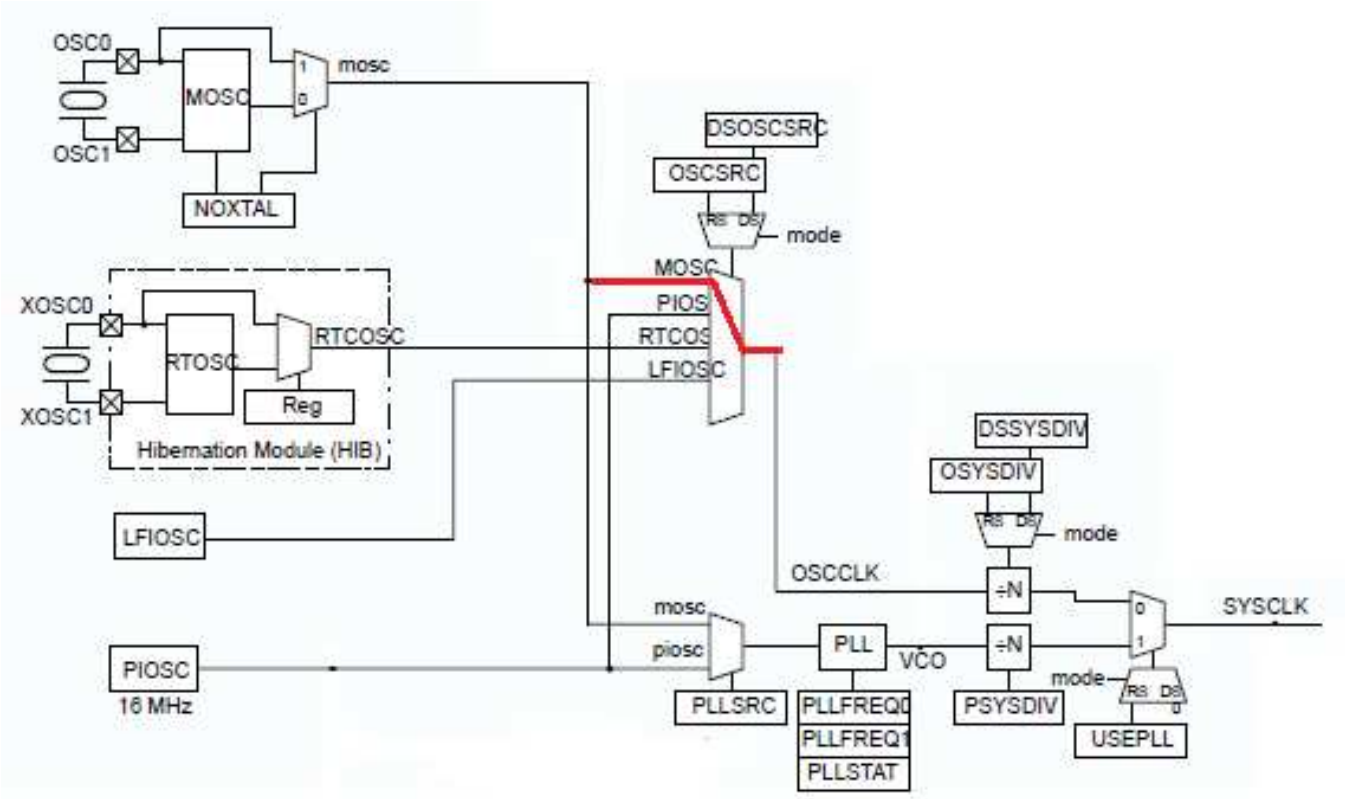
Workaround(s): There are two workarounds:

- Update the SysCtlClockFreqSet API as shown in Appendix 3
- Use the SysCtlClockFreqSet API in flash from TivaWare 2.1.3 release onwards.

SYSCTL#23 *MOSC as the Source to OSCCLK may Cause a bus Fault on Reset*

Revision(s) Affected: 1, 2, and 3.

Description: MOSC as the source to the OSCCLK may cause a bus fault on reset. This may happen even when using the PLL as SYSCLK and configuring the MOSC as the source for OSCCLK. Please see the following image on the affected clock path.



In TivaWare 2.1.2 or earlier, the SysCtlClockFreqSet sets the PLL source as MOSC when the API is called with the parameter SYSCTL_OSC_MAIN. It sets the OSCCLK as MOSC which is not required and may cause this issue. This issue may occur when using the ROM version of the API.

Workaround(s): Do not use MOSC as the OSCCLK source. There are two workarounds for application using SysCtlClockFreqSet API.

- Update the SysCtlClockFreqSet API as shown in Appendix 3
- Use the SysCtlClockFreqSet API in flash from TivaWare 2.1.3 release onwards.

UART#01 *When UART SIR Mode is Enabled, µDMA Burst Transfer Does not Occur*

Revision(s) Affected: 1, 2, and 3.

Description: If the IrDA Serial Infrared (SIR) mode is enabled in the UART peripheral and the µDMA is mapped to either UARTn RX or UARTn TX and is configured to do a burst transfer, the burst data transfer does not occur.

Workaround(s): Clear the respective SETn bit in the DMA Channel Useburst Set (DMAUSEBURSTSET) register to have the µDMA channel mapped to the UART to respond to single or burst requests to ensure that the data transfer occurs.

USB#02 *USB Controller Sends EOP at end of Device Remote Wake-Up*

Revision(s) Affected: 1 only.

Description: When the USB controller is operating as a Device and is suspended by the Host, and the USB controller issues a remote wake-up, an end of packet (EOP) is sent to the Host at the end of the Device's remote wake-up signal. Although this EOP is not expected, issues related to remote wake-up have not been observed. This does not affect USB certification.

Workaround(s): None.

USB#03 *Any Data Read From a Non-USB Register After Accessing the USBPP, USBPC, or the USBCC Register is Incorrect*

Revision(s) Affected: 1 only.

Description: A read from any of the following registers followed by a read from any other non-USB register on the AHB results in incorrect data in the non-USB register:

- USB Peripheral Properties (USBPP)
- USB Peripheral Configuration (USBPC)
- USB Clock Configuration (USBCC)

Workaround(s): Read any USB register after reading any of the registers in the above list and before reading any other non-USB register on the AHB. To determine which modules are on the AHB, refer to Figure 1-1 in the data sheet.

USB#04 *Device Sends SE0 in Response to a USB Bus Reset*

Revision(s) Affected: 1, 2, and 3.

Description: The USB Device (Tiva C MCU) will send an Single Ended Zero (SE0) bus state (USB0DP and USB0DM driven low) in response to a USB bus reset from the Host. Per USB specification, the Device should not drive these pins in the event of a USB bus reset. This does not affect USB certification.

Workaround(s): None.

USB#05 *USB Resume Occasionally does not Wake Device from Deep Sleep*

Revision(s) Affected: 1, 2, and 3.

Description: If configured to wake from Deep Sleep mode using a USB Resume signal, the device may remain in Deep Sleep mode if the Host tries to resume the Device from Suspend with a USB bus reset before it can enter Deep Sleep. There is a finite window of time where the RESUME interrupt is not realized. This window is from when the RESUME bit in the USB Device RESUME Interrupt Status and Clear (USBDRISC) register is cleared (write a 1) to before the Device enters Deep Sleep. During this time, if a bus reset or wake-up signal is issued by the Host, then the USBDRISC status bit clearing causes the valid USB bus operation to be lost.

Workaround(s): To prevent this from occurring, perform one of the two options:

- Ensure that the USB Suspend handler is exited only after a WFI instruction is processed by the core.
- Use Sleep mode instead of Deep Sleep mode and keep the USB module enabled in Sleep mode.

To minimize the window of time when the RESUME interrupt can be lost and reduce the risk of this issue occurring, clear the RESUME bit as close as possible to entering Deep Sleep.

NOTE: If using Sleep mode with the USB module enabled (second workaround), MOSC must be the clock source, using the PLL, and the system clock must be at least 30 MHz. As a result of the higher system clock and using Sleep mode instead of Deep Sleep mode, the current consumption will be higher with this workaround.

WDT#08 *Reading the WDTVALUE Register may Return Incorrect Values When Using Watchdog Timer 1*

Revision(s) Affected: 1, 2, and 3.

Description Incorrect values may be read from the Watchdog Value (WDTVALUE) register at the Watchdog Timer 1 base address when using Watchdog Timer 1.

Workaround(s) None.

6 Appendix 1

To address the erratum HIB#16, “Application Code May Miss New Tamper Event During Clear,” the HibernateTamperEventsClear() API must be replaced with the following APIs:

```
HibernateTamperEventsClearNoLock();
HibernateTamperUnLock();
HibernateTamperLock();
```

The API definitions are as follows:

```
//*****************************************************************************
//
//! Clears the tamper feature events without Unlock and Lock.
//!
//! This function is used to clear all tamper events without unlock/locking
//! the tamper control registers, so API HibernateTamperUnLock() should be
//! called before this function, and API HibernateTamperLock() should be
//! called after to ensure that tamper control registers are locked.
//!
//! This function doesn't block until the write is complete.
//! Therefore, care must be taken to ensure the next immediate write will
//! occure only after the write complete bit is set.
//!
//! This function is used to implement a software workaround in NMI interrupt
//! handler to fix an issue when a new tamper event could be missed during
//! the clear of current tamper event.
//!
//! \note The hibernate tamper feature is not available on all Tiva
//! devices.  Please consult the data sheet for the Tiva device that you
//! are using to determine if this feature is available.
//!
//! \return None.
//
//*****************************************************************************
void
HibernateTamperEventsClearNoLock(void)
{
    //
    // Wait for write completion.
    //
    _HibernateWriteComplete();

    //
    //
    // Set the tamper event clear bit.
    //
    HWREG(HIB_TPCTL) |= HIB_TPCTL_TPCLR;

}

//*****************************************************************************
//
//! Unlock temper registers.
//!
//! This function is used to unlock the temper control registers.  This
//! function should be only used before calling API
//! HibernateTamperEventsClearNoLock().
//!
//! \note The hibernate tamper feature is not available on all Tiva
//! devices.  Please consult the data sheet for the Tiva device that you
//! are using to determine if this feature is available.
//!
//! \return None.
//
//*****************************************************************************
void
HibernateTamperUnLock(void)
```

```
{
    //
    // Unlock the tamper registers.
    //
    HWREG(HIB_LOCK) = HIB_LOCK_HIBLOCK_KEY;
    _HibernateWriteComplete();
}
//*****************************************************************************
//
//! Lock temper registers.
//!
//! This function is used to lock the temper control registers.  This
//! function should be used after calling API
//! HibernateTamperEventsClearNoLock().
//!
//! \note The hibernate tamper feature is not available on all Tiva
//! devices.  Please consult the data sheet for the Tiva device that you
//! are using to determine if this feature is available.
//!
//! \return None.
//
//*****************************************************************************
void
HibernateTamperLock(void)
{
    //
    // Wait for write completion.
    //
    _HibernateWriteComplete();

    //
    // Lock the tamper registers.
    //
    HWREG(HIB_LOCK) = 0;
    _HibernateWriteComplete();
}
```

The software workaround must be added in the NMI Handler. The code mainly polls the tamper log entries during the tamper clear synchronization.

An example of an NMI handler with this workaround is shown below.

```
static uint32_t g_ui32RTCLog[4];
static uint32_t g_ui32EventLog[4];
//*****************************************************************************
//
// Handles an NMI interrupt generated by a Tamper event.
//
//*****************************************************************************
void
NMITamperEventHandler(void)
{
    uint32_t ui32NMIStatus, ui32TamperStatus;
    uint32_t pui32Buf[3];
    uint8_t ui8Idx, ui8StartIdx;
    bool     bDetectedEventsDuringClear;

    //
    // Get the cause of the NMI event.
    //
    ui32NMIStatus = SysCtlNMIStatus();

    //
    // We should have got the cause of the NMI event from the above function.
    // But in Snowflake RA0 the NMIC register is not set correctly when an
    // event occurs.  So as a work around check if the NMI event is caused by a
    // tamper event and append this to the return value from SysCtlNMIStatus().
```

```
// This way only this section can be removed once the bug is fixed in next
// silicon rev.
//
ui32TamperStatus = HibernateTamperStatusGet();
if(ui32TamperStatus & (HIBERNATE_TAMPER_STATUS_EVENT |
                       HIBERNATE_TAMPER_STATUS_EXT_OSC_FAILED))
{
    ui32NMIStatus |= SYSCTL_NMI_TAMPER;
}

//
// Check if SysCtlNMIStatus() returned a valid value.
//
if(ui32NMIStatus)
{

    //
    // Check if the NMI Interrupt is due to a Tamper event.
    //
    if(ui32NMIStatus & SYSCTL_NMI_TAMPER)
    {
        //
        // If the previous NMI event has not been processed by main
        // thread, we need to OR the new event along with the old ones.
        //
        if(g_ui32NMIEvent == 0)
        {
            //
            // Reset variables that used for tamper event.
            //
            g_ui32TamperEventFlag = 0;
            g_ui32TamperRTCLog = 0;

            //
            // Clean the log data for debugging purpose.
            //
            memset(g_ui32RTCLog, 0, (sizeof(g_ui32RTCLog))<<2);
            memset(g_ui32EventLog, 0, (sizeof(g_ui32EventLog))<<2);
        }


        //
        // Log the tamper event data before clearing tamper events.
        //
        for(ui8Idx = 0; ui8Idx< 4; ui8Idx++)
        {
            if(HibernateTamperEventsGet(ui8Idx,
                                        &g_ui32RTCLog[ui8Idx],
                                        &g_ui32EventLog[ui8Idx]))
            {
                //
                // Event in this log entry, store it.
                //
                g_ui32TamperEventFlag |= g_ui32EventLog[ui8Idx];
                g_ui32TamperRTCLog = g_ui32RTCLog[ui8Idx];
            }
            else
            {
                //
                // No event in this log entry. Done checking the logs.
                //
                break;
            }
        }

        //
```

```
// Process external oscillator failed event.
//
if(ui32TamperStatus & HIBERNATE_TAMPER_STATUS_EXT_OSC_FAILED)
{
    g_ui32TamperXOSCFailEvent++;
    g_ui32TamperEventFlag |= HIBERNATE_TAMPER_EVENT_EXT_OSC;
    g_ui32TamperRTCLog = HWREG(HIB_TPLOG0);
}
```

NOTE: This is the beginning of the block of workaround code.

```
// The following block of code is to workaround hardware defect
// which results in missing new tamper events during tamper clear
// synchronization.
//
// There is a window after the application code writes the tamper
// clear where a new tamper event can be missed if the application
// requires more than one tamper event pins detection.
//
// The tamper Clear is synchronized to the hibernate 32kHz clock
// domain.  The clear takes 3 rising edges of the 32KHz clock.
// During this window, new tamper events could be missed.
// A software workaround is to poll the tamper log during the
// tamper event clear synchronization.
//

//
// Clear the flag for the case there are events triggered
// during clear execution.
//
bDetectedEventsDuringClear = false;

//
// Unlock the Tamper Control register. This is required before
// calling HibernateTamperEventsClearNoLock().
//
HibernateTamperUnLock();
do
{
    //
    // We will start to poll the log registers at index 1 for
    // any new events.
    //
    ui8StartIdx = 1;
    //
    // Clear the Tamper event.
    // Note this API doesn't wait for synchronization, which
    // allows us to check the tamper log during
    // synchronization.
    //
    HibernateTamperEventsClearNoLock();

    //
    // Check new tamper event during tamper event clear
    // synchronization.
    // This will take about 92us(three clock cycles) at most.
    //
    while(HibernateTamperStatusGet() & HIBERNATE_TAMPER_STATUS_EVENT)
    {

        //
        // Clear execution isn't done yet , poll for new events.
        // If there were any new event, it will be logged in log 1
        // registers and so on.
```

```
//
for(ui8Idx = ui8StartIdx; ui8Idx< 4; ui8Idx++)
{
    if(HibernateTamperEventsGet(ui8Idx,
                                &g_ui32RTCLog[ui8Idx],
                                &g_ui32EventLog[ui8Idx]))
    {
        //
        // detected new event, store it.
        //
        g_ui32TamperEventFlag |= g_ui32EventLog[ui8Idx];

        //
        // check for more event.
        //
        continue;
    }
    else
    {
        //
        // no new event in this log, update the log index
        // to be checked next, and break out of loop.
        //
        ui8StartIdx = ui8Idx;
        break;
    }
}

//
// all last three logs have info. Check if all 4 logs
// have the same info. This is to detect the case that
// events happen during clear execution.
//
if(ui8Idx == 4)
{
    //
    // If events happens during clear
    // execution, all four log registers will be
    // logged with the same event, to detect this
    // condition, we will compare with all four log data.
    //
    if(HibernateTamperEventsGet(0, &g_ui32RTCLog[0], &g_ui32EventLog[0]))
    {
        if((g_ui32RTCLog[0] == g_ui32RTCLog[1])     &&
           (g_ui32EventLog[0] == g_ui32EventLog[1]) &&
          (g_ui32RTCLog[0] == g_ui32RTCLog[2])     &&
           (g_ui32EventLog[0] == g_ui32EventLog[2]) &&
           (g_ui32RTCLog[0] == g_ui32RTCLog[3])     &&
           (g_ui32EventLog[0] == g_ui32EventLog[3]))
        {
            //
            // Detected events during clear execution.
            // Event logging takes priority, the clear
            // will not be done in this case. We will need
            // to go back to the beginning of the loop and
            // clear the events.
            //
            if(bDetectedEventsDuringClear)
            {
                //
                // This condition has already detected,
                // we have cleared the event,
                // clear the flag.
                //
                bDetectedEventsDuringClear = false;
            }
```

```
                                else
                                {
                                    // This is the first time it has been
                                    // detected, set the flag.
                                    //
                                    bDetectedEventsDuringClear = true;
                                }

                                //
                                // Break out of while loop so that we can
                                // clear the events, and start the
                                // workaround all over again.
                                //
                                break;
                            }
                        }
                        else
                        {
                            //
                            // Log 0 didn't detect any events. So this is not
                            // the case of missing events during clear
                            // execution.
                            // Update the log index at which we will poll next.
                            // It should be the last log entry that OR all the
                            // new events.
                            //
                            ui8StartIdx = 3;
                        }
                    }
                }
            }
            while(bDetectedEventsDuringClear);
            //
            // Lock the Tamper Control register.
            //
            HibernateTamperLock();
```

NOTE: This is the end of the block of workaround code.

```
            //
            // Save the tamper event and RTC log info in the Hibernate Memory
            //
            HibernateDataGet(pui32Buf, 3);
            pui32Buf[1] = g_ui32TamperEventFlag;
            pui32Buf[2] = g_ui32TamperRTCLog;
            HibernateDataSet(pui32Buf, 3);

            //
            // Signal the main loop that an NMI event occurred.
            //
            g_ui32NMIEvent++;
        }

        //
        // Clear NMI events
        //
        SysCtlNMIClear(ui32NMIStatus);
    }

}
```

7 Appendix 2

To address the erratum EPI#01, "Data Reads can be Corrupted when the Code Address Space in the EPI Module is Used", the following code should be added to the epi.h file in the *C:\ti\TivaWare_C_Series-2.0\driverlib*:

```
#ifdef rvmdk
//*****************************************************************************
//
// Keil case.
//
//*****************************************************************************
inline void
EPIWorkaroundWordWrite(uint32_t *pui32Addr, uint32_t ui32Value)
{
  uint32_t ui32Scratch;

    __asm
    {
        //
        // Add a NOP to ensure we don’t have a flash read immediately before
        // the EPI read.
        //
        NOP

        //
        // Perform the write we're actually interested in.
        //
        STR ui32Value, [pui32Addr]

        //
        // Read from SRAM to ensure that we don't have an EPI write followed by
        // a flash read.
        //
        LDR ui32Scratch, [__current_sp()]
    }
}

inline uint32_t
EPIWorkaroundWordRead(uint32_t *pui32Addr)
{
  uint32_t ui32Value, ui32Scratch;

    __asm
    {
        //
        // Add a NOP to ensure we don’t have a flash read immediately before
        // the EPI read.
        //
        NOP

        //
        // Perform the read we're actually interested in.
        //
        LDR ui32Value, [pui32Addr]

        //
        // Read from SRAM to ensure that we don't have an EPI read followed by
        // a flash read.
        //
        LDR ui32Scratch, [__current_sp()]
    }

    return(ui32Value);
}

inline void
```

```
EPIWorkaroundHWordWrite(uint16_t *pui16Addr, uint16_t ui16Value)
{
    uint32_t ui32Scratch;

    __asm
    {
        //
        // Add a NOP to ensure we don’t have a flash read immediately before
        // the EPI read.
        //
        NOP

        //
        // Perform the write we're actually interested in.
        //
        STRH ui16Value, [pui16Addr]

        //
        // Read from SRAM to ensure that we don't have an EPI write followed by
        // a flash read.
        //
        LDR ui32Scratch, [__current_sp()]
    }
}

inline uint16_t
EPIWorkaroundHWordRead(uint16_t *pui16Addr)
{
    uint32_t ui32Scratch;
    uint16_t ui16Value;

    __asm
    {
        //
        // Add a NOP to ensure we don’t have a flash read immediately before
        // the EPI read.
        //
        NOP

        //
        // Perform the read we're actually interested in.
        //
        LDRH ui16Value, [pui16Addr]

        //
        // Read from SRAM to ensure that we don't have an EPI read followed by
        // a flash read.
        //
        LDR ui32Scratch, [__current_sp()]
    }

    return(ui16Value);
}

inline void
EPIWorkaroundByteWrite(uint8_t *pui8Addr, uint8_t ui8Value)
{
  uint32_t ui32Scratch;

    __asm
    {
        //
        // Add a NOP to ensure we don’t have a flash read immediately before
        // the EPI read.
        //
        NOP
```

```
        //
        // Perform the write we're actually interested in.
        //
        STRB ui8Value, [pui8Addr]

        //
        // Read from SRAM to ensure that we don't have an EPI write followed by
        // a flash read.
        //
        LDR ui32Scratch, [__current_sp()]
    }
}

inline uint8_t
EPIWorkaroundByteRead(uint8_t *pui8Addr)
{
    uint32_t ui32Scratch;
    uint8_t ui8Value;

    __asm
    {
        //
        // Add a NOP to ensure we don’t have a flash read immediately before
        // the EPI read.
        //
        NOP

        //
        // Perform the read we're actually interested in.
        //
        LDRB ui8Value, [pui8Addr]

        //
        // Read from SRAM to ensure that we don't have an EPI read followed by
        // a flash read.
        //
        LDR ui32Scratch, [__current_sp()]
    }

    return(ui8Value);
}

#else
#ifdef ccs

//*****************************************************************************
//
// Code Composer Studio versions of these functions can be found in separate
// source file epi_workaround_ccs.s.
//
//*****************************************************************************
extern void EPIWorkaroundWordWrite(uint32_t *pui32Addr, uint32_t ui32Value);
extern uint32_t EPIWorkaroundWordRead(uint32_t *pui32Addr);
extern void EPIWorkaroundHWordWrite(uint16_t *pui16Addr, uint16_t ui16Value);
extern uint16_t EPIWorkaroundHWordRead(uint16_t *pui16Addr);
extern void EPIWorkaroundByteWrite(uint8_t *pui8Addr, uint8_t ui8Value);
extern uint8_t EPIWorkaroundByteRead(uint8_t *pui8Addr);

#else
//*****************************************************************************
//
// GCC and IAR case.
//
//*****************************************************************************
inline void
```

```
EPIWorkaroundWordWrite(uint32_t *pui32Addr, uint32_t ui32Value)
{
    volatile register uint32_t ui32Scratch;

    __asm volatile (
        //
        // Add a NOP to ensure we don't have a flash read immediately before
        // the EPI read.
        //
        "    NOP\n"
        "    STR %[value],[%[addr]]\n"
        "    LDR %[scratch],[sp]\n"
         :  [scratch] "=r" (ui32Scratch)
         :  [addr] "r" (pui32Addr), [value] "r" (ui32Value)
     );

    //
    // Keep the compiler from generating a warning.
    //
    ui32Scratch = ui32Scratch;
}

inline uint32_t
EPIWorkaroundWordRead(uint32_t *pui32Addr)
{
    volatile register uint32_t ui32Data, ui32Scratch;

    //
    // ui32Scratch is not used other than to add a padding read following the
    // "real" read.
    //

    __asm volatile(
        //
        // Add a NOP to ensure we don't have a flash read immediately before
        // the EPI read.
        //
        "    NOP\n"
        "    LDR %[ret],[%[addr]]\n"
        "    LDR %[scratch],[sp]\n"
         : [ret] "=r" (ui32Data),
           [scratch] "=r" (ui32Scratch)
         : [addr] "r" (pui32Addr)
    );


    //
    // Keep the compiler from generating a warning.
    //
    ui32Scratch = ui32Scratch;

    return(ui32Data);
}

inline void
EPIWorkaroundHWordWrite(uint16_t *pui16Addr, uint16_t ui16Value)
{
    volatile register uint32_t ui32Scratch;

    __asm volatile (
        //
        // Add a NOP to ensure we don't have a flash read immediately before
        // the EPI read.
        //
        "    NOP\n"
        "    STRH %[value],[%[addr]]\n"
```

```
            "    LDR %[scratch],[sp]\n"
             :  [scratch] "=r" (ui32Scratch)
             :  [addr] "r" (pui16Addr), [value] "r" (ui16Value)
         );


    //
    // Keep the compiler from generating a warning.
    //
    ui32Scratch = ui32Scratch;
}

inline uint16_t
EPIWorkaroundHWordRead(uint16_t *pui16Addr)
{
    register uint16_t ui16Data;
    register uint32_t ui32Scratch;

    //
    // ui32Scratch is not used other than to add a padding read following the
    // "real" read.
    //

    __asm volatile(
        //
        // Add a NOP to ensure we don't have a flash read immediately before
        // the EPI read.
        //
        "    NOP\n"
        "    LDRH %[ret],[%[addr]]\n"
        "    LDR %[scratch],[sp]\n"
         : [ret] "=r" (ui16Data),
           [scratch] "=r" (ui32Scratch)
         : [addr] "r" (pui16Addr)
    );

    //
    // Keep the compiler from generating a warning.
    //
    ui32Scratch = ui32Scratch;

    return(ui16Data);
}

inline void
EPIWorkaroundByteWrite(uint8_t *pui8Addr, uint8_t ui8Value)
{
    volatile register uint32_t ui32Scratch;

    __asm volatile (
        //
        // Add a NOP to ensure we don't have a flash read immediately before
        // the EPI read.
        //
        "    NOP\n"
        "    STRB %[value],[%[addr]]\n"
        "    LDR %[scratch],[sp]\n"
         :  [scratch] "=r" (ui32Scratch)
         :  [addr] "r" (pui8Addr), [value] "r" (ui8Value)
     );

    //
    // Keep the compiler from generating a warning.
    //
    ui32Scratch = ui32Scratch;
}
```

```
inline uint8_t
EPIWorkaroundByteRead(uint8_t *pui8Addr)
{
    register uint8_t ui8Data;
    register uint32_t ui32Scratch;

    //
    // ui32Scratch is not used other than to add a padding read following the
    // "real" read.
    //

    __asm volatile(
        //
        // Add a NOP to ensure we don't have a flash read immediately before
        // the EPI read.
        //
        "    NOP\n"
        "    LDRB %[ret],[%[addr]]\n"
        "    LDR %[scratch],[sp]\n"
         : [ret] "=r" (ui8Data),
           [scratch] "=r" (ui32Scratch)
         : [addr] "r" (pui8Addr)
    );

    //
    // Keep the compiler from generating a warning.
    //
    ui32Scratch = ui32Scratch;

    return(ui8Data);
}

#endif
```

In addition, if using CCS, the following code should be saved as a file entitled epi_workaround_ccs.s in the C:\ti\TivaWare_C_Series-2.0\driverlib directory and included in the project:

```
;*******************************************************************************
;
; epi_workaround_ccs.s - EPI memory access functions.
;
; Copyright (c) 2013 Texas Instruments Incorporated.  All rights reserved.
; TI Information - Selective Disclosure
;
;*******************************************************************************

;*******************************************************************************
;
; void EPIWorkaroundWordWrite(uint32_t *pui32Addr, uint32_t ui32Value)
;
;*******************************************************************************
    .sect ".text:EPIWorkaroundWordWrite"
    .global EPIWorkaroundWordWrite
EPIWorkaroundWordWrite:
    ;
    ; Include a no-op to ensure that we don't have a flash data access
    ; immediately before the EPI access.
    ;
    nop

    ;
    ; Store the word in EPI memory.
    ;
    str r1, [r0]

    ;
```

```
    ; Make a dummy read from the stack to ensure that we don't have a flash
    ; data access immediately after the EPI access.
    ;
    ldr r1, [sp]

    ;
    ; Return to the caller.
    ;
    bx lr

    .align 4

;*******************************************************************************
;
; uint32_t EPIWorkaroundWordRead(uint32_t *pui32Addr)
;
;*******************************************************************************
    .sect ".text:EPIWorkaroundWordRead"
    .global EPIWorkaroundWordRead
EPIWorkaroundWordRead:
    ;
    ; Include a no-op to ensure that we don't have a flash data access
    ; immediately before the EPI access.
    ;
    nop

    ;
    ; Read the word from EPI memory.
    ;
    ldr r0, [r0]

    ;
    ; Make a dummy read from the stack to ensure that we don't have a flash
    ; data access immediately after the EPI access.
    ;
    ldr r1, [r13]

    ;
    ; Return to the caller.
    ;
    bx lr

    .align 4


;*******************************************************************************
;
; void EPIWorkaroundHWordWrite(uint16_t *pui16Addr, uint16_t ui16Value)
;
;*******************************************************************************
    .sect ".text:EPIWorkaroundHWordWrite"
    .global EPIWorkaroundHWordWrite
EPIWorkaroundHWordWrite:
    ;
    ; Include a no-op to ensure that we don't have a flash data access
    ; immediately before the EPI access.
    ;
    nop

    ;
    ; Store the word in EPI memory.
    ;
    strh r1, [r0]

    ;
    ; Make a dummy read from the stack to ensure that we don't have a flash
```

```
    ; data access immediately after the EPI access.
    ;
    ldr r1, [sp]

    ;
    ; Return to the caller.
    ;
    bx lr

    .align 4

;*******************************************************************************
;
; uint16_t EPIWorkaroundHWordRead(uint16_t *pui16Addr)
;
;*******************************************************************************
    .sect ".text:EPIWorkaroundHWordRead"
    .global EPIWorkaroundHWordRead
EPIWorkaroundHWordRead:
    ;
    ; Include a no-op to ensure that we don't have a flash data access
    ; immediately before the EPI access.
    ;
    nop

    ;
    ; Read the half word from EPI memory.
    ;
    ldrh r0, [r0]

    ;
    ; Make a dummy read from the stack to ensure that we don't have a flash
    ; data access immediately after the EPI access.
    ;
    ldr r1, [r13]

    ;
    ; Return to the caller.
    ;
    bx lr

    .align 4

;*******************************************************************************
;
; void EPIWorkaroundByteWrite(uint8_t *pui8Addr, uint8_t ui8Value)
;
;*******************************************************************************
    .sect ".text:EPIWorkaroundByteWrite"
    .global EPIWorkaroundByteWrite
EPIWorkaroundByteWrite:
    ;
    ; Include a no-op to ensure that we don't have a flash data access
    ; immediately before the EPI access.
    ;
    nop

    ;
    ; Store the byte in EPI memory.
    ;
    strb r1, [r0]

    ;
    ; Make a dummy read from the stack to ensure that we don't have a flash
    ; data access immediately after the EPI access.
    ;
```

```
    ldr r1, [sp]

    ;
    ; Return to the caller.
    ;
    bx lr

    .align 4

;*******************************************************************************
;
; uint8_t EPIWorkaroundByteRead(uint8_t *pui8Addr)
;
;*******************************************************************************
    .sect ".text:EPIWorkaroundByteRead"
    .global EPIWorkaroundByteRead
EPIWorkaroundByteRead:
    ;
    ; Include a no-op to ensure that we don't have a flash data access
    ; immediately before the EPI access.
    ;
    nop

    ;
    ; Read the byte from EPI memory.
    ;
    ldrb r0, [r0]

    ;
    ; Make a dummy read from the stack to ensure that we don't have a flash
    ; data access immediately after the EPI access.
    ;
    ldr r1, [r13]

    ;
    ; Return to the caller.
    ;
    bx lr

    .align 4

.end
```

8 Appendix 3

To address the erratum SYSCTL#22, "*Change to the PLL Clock Divider may Cause System Clock to be out of Specification*" and SYSCTL#23, "*MOSC as the source to OSCCLK may Cause a bus Fault on Reset*", the SysCtlClockFreqSet API must be updated as follows.

8.1 Change 1

The first change to be made is to add the define PLL_Q_TO_REG(q) in the driverlib file sysctl.c.

```
//*****************************************************************************
//
// These macros are used in the g_pui32XTALtoVCO table to make it more
// readable.
//
//*****************************************************************************
#define PLL_M_TO_REG(mi, mf)                                                  \
    ((uint32_t)mi | (uint32_t)(mf << SYSCTL_PLLFREQ0_MFRAC_S))
#define PLL_N_TO_REG(n)                                                       \
        ((uint32_t)(n - 1) << SYSCTL_PLLFREQ1_N_S)
```

NOTE: Add the following code here:

```
#define PLL_Q_TO_REG(q)
        ((uint32_t)(q - 1) << SYSCTL_PLLFREQ1_Q_S)
```

8.2 Change 2

The second change is to replace the crystal to VCO lookup table as given below in the driverlib file sysctl.c.

```
//*****************************************************************************
//
// Look up of the values that go into the PLLFREQ0 and PLLFREQ1 registers.
//
//*****************************************************************************
static const uint32_t g_pppui32XTALtoVCO[MAX_VCO_ENTRIES][MAX_XTAL_ENTRIES][2] =
{
    {
        //
        // VCO 320 MHz
        //
        { PLL_M_TO_REG(64, 0),   PLL_N_TO_REG(1) },     // 5 MHz
        { PLL_M_TO_REG(62, 512), PLL_N_TO_REG(1) },     // 5.12 MHz
        { PLL_M_TO_REG(160, 0),  PLL_N_TO_REG(3) },     // 6 MHz
        { PLL_M_TO_REG(52, 85),  PLL_N_TO_REG(1) },     // 6.144 MHz
        { PLL_M_TO_REG(43, 412), PLL_N_TO_REG(1) },     // 7.3728 MHz
        { PLL_M_TO_REG(40, 0),   PLL_N_TO_REG(1) },     // 8 MHz
        { PLL_M_TO_REG(39, 64),  PLL_N_TO_REG(1) },     // 8.192 MHz
        { PLL_M_TO_REG(32, 0),   PLL_N_TO_REG(1) },     // 10 MHz
        { PLL_M_TO_REG(80, 0),   PLL_N_TO_REG(3) },     // 12 MHz
        { PLL_M_TO_REG(26, 43),  PLL_N_TO_REG(1) },     // 12.288 MHz
        { PLL_M_TO_REG(23, 613), PLL_N_TO_REG(1) },     // 13.56 MHz
        { PLL_M_TO_REG(22, 358), PLL_N_TO_REG(1) },     // 14.318180 MHz
        { PLL_M_TO_REG(20, 0),   PLL_N_TO_REG(1) },     // 16 MHz
        { PLL_M_TO_REG(19, 544), PLL_N_TO_REG(1) },     // 16.384 MHz
        { PLL_M_TO_REG(160, 0),  PLL_N_TO_REG(9) },     // 18 MHz
        { PLL_M_TO_REG(16, 0),   PLL_N_TO_REG(1) },     // 20 MHz
        { PLL_M_TO_REG(40, 0),   PLL_N_TO_REG(3) },     // 24 MHz
        { PLL_M_TO_REG(64, 0),   PLL_N_TO_REG(5) },     // 25 MHz
    },
    {
        //
        // VCO 480 MHz
        //
        { PLL_M_TO_REG(96, 0),   PLL_N_TO_REG(1) },     // 5 MHz
        { PLL_M_TO_REG(93, 768), PLL_N_TO_REG(1) },     // 5.12 MHz
        { PLL_M_TO_REG(80, 0),   PLL_N_TO_REG(1) },     // 6 MHz
        { PLL_M_TO_REG(78, 128), PLL_N_TO_REG(1) },     // 6.144 MHz
        { PLL_M_TO_REG(65, 107), PLL_N_TO_REG(1) },     // 7.3728 MHz
        { PLL_M_TO_REG(60, 0),   PLL_N_TO_REG(1) },     // 8 MHz
        { PLL_M_TO_REG(58, 608), PLL_N_TO_REG(1) },     // 8.192 MHz
        { PLL_M_TO_REG(48, 0),   PLL_N_TO_REG(1) },     // 10 MHz
        { PLL_M_TO_REG(40, 0),   PLL_N_TO_REG(1) },     // 12 MHz
        { PLL_M_TO_REG(39, 64),  PLL_N_TO_REG(1) },     // 12.288 MHz
        { PLL_M_TO_REG(35, 408), PLL_N_TO_REG(1) },     // 13.56 MHz
        { PLL_M_TO_REG(33, 536), PLL_N_TO_REG(1) },     // 14.318180 MHz
        { PLL_M_TO_REG(30, 0),   PLL_N_TO_REG(1) },     // 16 MHz
        { PLL_M_TO_REG(29, 304), PLL_N_TO_REG(1) },     // 16.384 MHz
        { PLL_M_TO_REG(80, 0),   PLL_N_TO_REG(3) },     // 18 MHz
        { PLL_M_TO_REG(24, 0),   PLL_N_TO_REG(1) },     // 20 MHz
        { PLL_M_TO_REG(20, 0),   PLL_N_TO_REG(1) },     // 24 MHz
        { PLL_M_TO_REG(96, 0),   PLL_N_TO_REG(5) },     // 25 MHz
    },
};
```

NOTE: Replace the code block above with the code block below.

```
//*****************************************************************************
//
// Look up of the values that go into the PLLFREQ0 and PLLFREQ1 registers.
//
//*****************************************************************************
static const uint32_t g_pppui32XTALtoVCO[MAX_VCO_ENTRIES][MAX_XTAL_ENTRIES][3] =
{
    {
        //
        // VCO 320 MHz
        //
        { PLL_M_TO_REG(64, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 5 MHz
        { PLL_M_TO_REG(62, 512), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 5.12 MHz
        { PLL_M_TO_REG(160, 0), PLL_N_TO_REG(3), PLL_Q_TO_REG(2) }, // 6 MHz
        { PLL_M_TO_REG(52, 85), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 6.144 MHz
        { PLL_M_TO_REG(43, 412), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 7.3728 MHz
        { PLL_M_TO_REG(40, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 8 MHz
        { PLL_M_TO_REG(39, 64), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 8.192 MHz
        { PLL_M_TO_REG(32, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 10 MHz
        { PLL_M_TO_REG(80, 0), PLL_N_TO_REG(3), PLL_Q_TO_REG(2) }, // 12 MHz
        { PLL_M_TO_REG(26, 43), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 12.288 MHz
        { PLL_M_TO_REG(23, 613), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 13.56 MHz
        { PLL_M_TO_REG(22, 358), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 14.318180 MHz
        { PLL_M_TO_REG(20, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 16 MHz
        { PLL_M_TO_REG(19, 544), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 16.384 MHz
        { PLL_M_TO_REG(160, 0), PLL_N_TO_REG(9), PLL_Q_TO_REG(2) }, // 18 MHz
        { PLL_M_TO_REG(16, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 20 MHz
        { PLL_M_TO_REG(40, 0), PLL_N_TO_REG(3), PLL_Q_TO_REG(2) }, // 24 MHz
        { PLL_M_TO_REG(64, 0), PLL_N_TO_REG(5), PLL_Q_TO_REG(2) }, // 25 MHz
    },
    {
        //
        // VCO 480 MHz
        //
        { PLL_M_TO_REG(96, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 5 MHz
        { PLL_M_TO_REG(93, 768), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 5.12 MHz
        { PLL_M_TO_REG(80, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 6 MHz
        { PLL_M_TO_REG(78, 128), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 6.144 MHz
        { PLL_M_TO_REG(65, 107), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 7.3728 MHz
        { PLL_M_TO_REG(60, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 8 MHz
        { PLL_M_TO_REG(58, 608), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 8.192 MHz
        { PLL_M_TO_REG(48, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 10 MHz
        { PLL_M_TO_REG(40, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 12 MHz
        { PLL_M_TO_REG(39, 64), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 12.288 MHz
        { PLL_M_TO_REG(35, 408), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 13.56 MHz
        { PLL_M_TO_REG(33, 536), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 14.318180 MHz
        { PLL_M_TO_REG(30, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 16 MHz
        { PLL_M_TO_REG(29, 304), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 16.384 MHz
        { PLL_M_TO_REG(80, 0), PLL_N_TO_REG(3), PLL_Q_TO_REG(2) }, // 18 MHz
        { PLL_M_TO_REG(24, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 20 MHz
        { PLL_M_TO_REG(20, 0), PLL_N_TO_REG(1), PLL_Q_TO_REG(2) }, // 24 MHz
        { PLL_M_TO_REG(96, 0), PLL_N_TO_REG(5), PLL_Q_TO_REG(2) }, // 25 MHz
    },
};
```

8.3 Change 3

The third change is to replace the VCO frequencies in the driverlib file sysctl.c.

```
//*****************************************************************************
//
// Look up of the possible VCO frequencies.
//
//*****************************************************************************
static const uint32_t g_pui32VCOFrequencies[MAX_VCO_ENTRIES] =
{
    320000000,                // VCO 320
    480000000,                // VCO 480
};
```

NOTE: Replace the code block above with the code block below.

```
//*****************************************************************************
//
// Look up of the possible VCO frequencies.
//
//*****************************************************************************
static const uint32_t g_pui32VCOFrequencies[MAX_VCO_ENTRIES] =
{
    160000000,                // VCO 320
    240000000,                // VCO 480
};
```

8.4 Change 4

Final change is to replace the existing SysCtlClockFreqSet API with the code given below.

```
uint32_t
SysCtlClockFreqSet(uint32_t ui32Config, uint32_t ui32SysClock)
{
    int32_t i32Timeout, i32VCOIdx, i32XtalIdx;
    uint32_t ui32MOSCCTL;
    uint32_t ui32SysDiv, ui32Osc, ui32OscSelect, ui32RSClkConfig;
    bool bNewPLL;

    //
    // TM4C123 devices should not use this function.
    //
    if(CLASS_IS_TM4C123)
    {
        return(0);
    }

    //
    // Get the index of the crystal from the ui32Config parameter.
    //
    i32XtalIdx = SysCtlXtalCfgToIndex(ui32Config);

    //
    // Determine which non-PLL source was selected.
    //
    if((ui32Config & 0x38) == SYSCTL_OSC_INT)
    {
        //
        // Use the nominal frequency for the PIOSC oscillator and set the
        // crystal select.
        //
        ui32Osc = 16000000;
        ui32OscSelect = SYSCTL_RSCLKCFG_OSCSRC_PIOSC;
        ui32OscSelect |= SYSCTL_RSCLKCFG_PLLSRC_PIOSC;
```

```
        //
        // Force the crystal index to the value for 16-MHz.
        //
        i32XtalIdx = SysCtlXtalCfgToIndex(SYSCTL_XTAL_16MHZ);
    }
    else if((ui32Config & 0x38) == SYSCTL_OSC_INT30)
    {
        //
        // Use the nominal frequency for the low frequency oscillator.
        //
        ui32Osc = 30000;
        ui32OscSelect = SYSCTL_RSCLKCFG_OSCSRC_LFIOSC;
    }
    else if((ui32Config & 0x38) == (SYSCTL_OSC_EXT32 & 0x38))
    {
        //
        // Use the RTC frequency.
        //
        ui32Osc = 32768;
        ui32OscSelect = SYSCTL_RSCLKCFG_OSCSRC_RTC;
    }
    else if((ui32Config & 0x38) == SYSCTL_OSC_MAIN)
    {
        //
        // Bounds check the source frequency for the main oscillator.  The is
        // because the PLL tables in the g_pppui32XTALtoVCO structure range
        // from 5MHz to 25MHz.
        //
        if((i32XtalIdx > (SysCtlXtalCfgToIndex(SYSCTL_XTAL_25MHZ))) ||
           (i32XtalIdx < (SysCtlXtalCfgToIndex(SYSCTL_XTAL_5MHZ))))
        {
            return(0);
        }

        ui32Osc = g_pui32Xtals[i32XtalIdx];

        //
        // Set the PLL source select to MOSC.
        //
        ui32OscSelect = SYSCTL_RSCLKCFG_OSCSRC_MOSC;
        ui32OscSelect |= SYSCTL_RSCLKCFG_PLLSRC_MOSC;

        //
        // Clear MOSC power down, high oscillator range setting, and no crystal
        // present setting.
        //
        ui32MOSCCTL = HWREG(SYSCTL_MOSCCTL) &
                      ~(SYSCTL_MOSCCTL_OSCRNG | SYSCTL_MOSCCTL_PWRDN |
                        SYSCTL_MOSCCTL_NOXTAL);

        //
        // Increase the drive strength for MOSC of 10 MHz and above.
        //
        if(i32XtalIdx >= (SysCtlXtalCfgToIndex(SYSCTL_XTAL_10MHZ) -
                          (SysCtlXtalCfgToIndex(SYSCTL_XTAL_5MHZ))))
        {
            ui32MOSCCTL |= SYSCTL_MOSCCTL_OSCRNG;
        }

        HWREG(SYSCTL_MOSCCTL) = ui32MOSCCTL;
    }
    else
    {
        //
        // This was an invalid request because no oscillator source was
        // indicated.
```

```
        //
        ui32Osc = 0;
        ui32OscSelect = SYSCTL_RSCLKCFG_OSCSRC_PIOSC;
    }

    //
    // Check if the running with the PLL enabled was requested.
    //
    if((ui32Config & SYSCTL_USE_OSC) == SYSCTL_USE_PLL)
    {
        //
        // ui32Config must be SYSCTL_OSC_MAIN or SYSCTL_OSC_INT.
        //
        if(((ui32Config & 0x38) != SYSCTL_OSC_MAIN) &&
           ((ui32Config & 0x38) != SYSCTL_OSC_INT))
        {
            return(0);
        }

        //
        // Get the VCO index out of the ui32Config parameter.
        //
        i32VCOIdx = (ui32Config >> 24) & 7;

        //
        // Check that the VCO index is not out of bounds.
        //
        ASSERT(i32VCOIdx < MAX_VCO_ENTRIES);

        //
        // Set the memory timings for the maximum external frequency since
        // this could be a switch to PIOSC or possibly to MOSC which can be
        // up to 25MHz.
        //
        HWREG(SYSCTL_MEMTIM0) = _SysCtlMemTimingGet(25000000);

        //
        // Clear the old PLL divider and source in case it was set.
        //
        ui32RSClkConfig = HWREG(SYSCTL_RSCLKCFG) &
                          ~(SYSCTL_RSCLKCFG_PSYSDIV_M |
                            SYSCTL_RSCLKCFG_OSCSRC_M |
                            SYSCTL_RSCLKCFG_PLLSRC_M | SYSCTL_RSCLKCFG_USEPLL);

        //
        // Update the memory timings to match running from PIOSC.
        //
        ui32RSClkConfig |= SYSCTL_RSCLKCFG_MEMTIMU;

        //
        // Update clock configuration to switch back to PIOSC.
        //
        HWREG(SYSCTL_RSCLKCFG) = ui32RSClkConfig;

        //
        // The table starts at 5 MHz so modify the index to match this.
        //
        i32XtalIdx -= SysCtlXtalCfgToIndex(SYSCTL_XTAL_5MHZ);

        //
        // If there were no changes to the PLL do not force the PLL to lock by
        // writing the PLL settings.
        //

        if((HWREG(SYSCTL_PLLFREQ1) !=
            (g_pppui32XTALtoVCO[i32VCOIdx][i32XtalIdx][1] |
```

```
     g_pppui32XTALtoVCO[i32VCOIdx][i32XtalIdx][2])) ||
   (HWREG(SYSCTL_PLLFREQ0) !=
    (g_pppui32XTALtoVCO[i32VCOIdx][i32XtalIdx][0] |
     SYSCTL_PLLFREQ0_PLLPWR)))

{
    bNewPLL = true;
}
else
{
    bNewPLL = false;
}

//
// If there are new PLL settings write them.
//
if(bNewPLL)
{
    //
    // Set the oscillator source.
    //
    HWREG(SYSCTL_RSCLKCFG) |= ui32OscSelect;

    //
    // Set the M, N and Q values provided from the table and preserve
    // the power state of the main PLL.
    //
    HWREG(SYSCTL_PLLFREQ1) =
        g_pppui32XTALtoVCO[i32VCOIdx][i32XtalIdx][1];
    HWREG(SYSCTL_PLLFREQ1) |=
        g_pppui32XTALtoVCO[i32VCOIdx][i32XtalIdx][2];
    HWREG(SYSCTL_PLLFREQ0) =
        (g_pppui32XTALtoVCO[i32VCOIdx][i32XtalIdx][0] |
         (HWREG(SYSCTL_PLLFREQ0) & SYSCTL_PLLFREQ0_PLLPWR));
}

//
// Calculate the System divider such that we get a frequency that is
// the closest to the requested frequency without going over.
//
ui32SysDiv = (g_pui32VCOFrequencies[i32VCOIdx] + ui32SysClock - 1) /
             ui32SysClock;

//
// Calculate the actual system clock.
//
ui32SysClock = _SysCtlFrequencyGet(ui32Osc) / ui32SysDiv;

//
// Set the Flash and EEPROM timing values.
//
HWREG(SYSCTL_MEMTIM0) = _SysCtlMemTimingGet(ui32SysClock);

//
// Check if the PLL is already powered up.
//
if(HWREG(SYSCTL_PLLFREQ0) & SYSCTL_PLLFREQ0_PLLPWR)
{
    if(bNewPLL == true)
    {
        //
        // Trigger the PLL to lock to the new frequency.
        //
        HWREG(SYSCTL_RSCLKCFG) |= SYSCTL_RSCLKCFG_NEWFREQ;
    }
}
```

```
        else
        {
            //
            // Power up the PLL.
            //
            HWREG(SYSCTL_PLLFREQ0) |= SYSCTL_PLLFREQ0_PLLPWR;
        }

        //
        // Wait until the PLL has locked.
        //

        for(i32Timeout = 32768; i32Timeout > 0; i32Timeout--)
        {
            if((HWREG(SYSCTL_PLLSTAT) & SYSCTL_PLLSTAT_LOCK))
            {
                break;
            }
        }

        //
        // If the loop above did not timeout then switch over to the PLL
        //
        if(i32Timeout)
        {
            ui32RSClkConfig = HWREG(SYSCTL_RSCLKCFG);
            ui32RSClkConfig |= ((ui32SysDiv - 1) <<
                                SYSCTL_RSCLKCFG_PSYSDIV_S) | ui32OscSelect |
                               SYSCTL_RSCLKCFG_USEPLL;
            ui32RSClkConfig |= SYSCTL_RSCLKCFG_MEMTIMU;

            //
            // Set the new clock configuration.
            //
            HWREG(SYSCTL_RSCLKCFG) = ui32RSClkConfig;
        }
        else
        {
            ui32SysClock = 0;
        }
    }
    else
    {

        //
        // Set the Flash and EEPROM timing values for PIOSC.
        //
        HWREG(SYSCTL_MEMTIM0) = _SysCtlMemTimingGet(16000000);

        //
        // Make sure that the PLL is powered down since it is not being used.
        //
        HWREG(SYSCTL_PLLFREQ0) &= ~SYSCTL_PLLFREQ0_PLLPWR;

        //
        // Clear the old PLL divider and source in case it was set.
        //
        ui32RSClkConfig = HWREG(SYSCTL_RSCLKCFG);
        ui32RSClkConfig &= ~(SYSCTL_RSCLKCFG_OSYSDIV_M |
                             SYSCTL_RSCLKCFG_OSCSRC_M |
                             SYSCTL_RSCLKCFG_USEPLL);

        //
        // Update the memory timings.
        //
        ui32RSClkConfig |= SYSCTL_RSCLKCFG_MEMTIMU;
```

```
    //
    // Set the new clock configuration.
    //
    HWREG(SYSCTL_RSCLKCFG) = ui32RSClkConfig;

    //
    // If zero given as the system clock then default to divide by 1.
    //
    if(ui32SysClock == 0)
    {
        ui32SysDiv = 0;
    }
    else
    {
        //
        // Calculate the System divider based on the requested
        // frequency.
        //
        ui32SysDiv = ui32Osc / ui32SysClock;

        //
        // If the system divisor is not already zero, subtract one to
        // set the value in the register which requires the value to
        // be n-1.
        //
        if(ui32SysDiv != 0)
        {
            ui32SysDiv -= 1;
        }

        //
        // Calculate the system clock.
        //
        ui32SysClock = ui32Osc / (ui32SysDiv + 1);

    }

    //
    // Set the memory timing values for the new system clock.
    //
    HWREG(SYSCTL_MEMTIM0) = _SysCtlMemTimingGet(ui32SysClock);

    //
    // Set the new system clock values.
    //
    ui32RSClkConfig = HWREG(SYSCTL_RSCLKCFG);
    ui32RSClkConfig |= (ui32SysDiv << SYSCTL_RSCLKCFG_OSYSDIV_S) |
                       ui32OscSelect;

    //
    // Update the memory timings.
    //
    ui32RSClkConfig |= SYSCTL_RSCLKCFG_MEMTIMU;

    //
    // Set the new clock configuration.
    //
    HWREG(SYSCTL_RSCLKCFG) = ui32RSClkConfig;
}

//
// Finally change the OSCSRC back to PIOSC
//
HWREG(SYSCTL_RSCLKCFG) &= ~(SYSCTL_RSCLKCFG_OSCSRC_M);
```

```
        return(ui32SysClock);
    }
```

Trademarks

Tiva is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from H Revision (May 2016) to G Revision	Page
• Added new advisory: PWM#05, *Generator Load with Global Sync May Lead to Erroneous Pulse Width*	65
• Added new advisory: PWM#06, *PWM Output May Generate a Continuous High Instead of a Low or a Continuous Low Instead of High*	66

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources 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.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

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