C2000™ CPU Memory Built-In Self-Test

Salvatore Pezzino, Peter Ehlig

ABSTRACT

This application report describes embedded memory validation using the CPU during an active control system. The code elements involved are referred to as CPUMBIST. This document focuses on the TMS320F2837xD, TMS320F2837xS, and TMS320F2807x series of C2000™ devices, hereafter referred to as F28x7x devices.

Contents

List of Figures

List of Tables

Trademarks

C2000, SafeTI, Code Composer Studio, Piccolo, Delfino are trademarks of Texas Instruments.
All other trademarks are the property of their respective owners.

1 Introduction

The F28x7x series includes both single-core (F2837xS and F2807x) and dual-core (F2837xD) products. This application report refers to the March13n and CRC algorithms and APIs provided in the C2000 SafeTI™ Diagnostic Library. For more information regarding the software implementation of these functions, see the *Diag_Lib_TMS320F2837x_07x_Users_Guide*, which is in the /docs folder of the library release package.

1.1 Term and Definitions

Table 1. Terms and Definitions

Term	Definition
BIST	Built-In Self-Test
Bit disturbance	When a bit cell changes states without being written.
CAN	Control area network
CCS	Code Composer Studio™
Context restore	The process of restoring the CPU registers and status flags after completing a HWBIST micro-run. This is performed by software.
Context save	The process of saving the CPU registers and status flags before starting a HWBIST micro-run. This is performed by software.
CPU	Central processing unit
CRC	Cyclical redundancy check
DMA	Direct memory access
ECC	Error correcting code
EDAC	Error detection and correction
F2807x	C28x single-core Piccolo™ class device
F2837xD	C28x dual-core Delfino™ class device
F2837xS	Single-core C28x Delfino class device
Flash	Nonvolatile on-chip memory
FPU	Floating-point unit
ISR	Interrupt service routine
JTAG	Joint test action group (debugger)
MBIST	Memory Built-In Self-Test
March	A finite sequence of march elements (for example, read or write operations)
March7n	A sequence of 7 march elements. Three writes and four reads of each word in SRAM.
March13n	A sequence of 13 march elements. Five writes and eight reads of each word in SRAM.
Micro-run	Refers to the execution of a portion of a full CPUMBIST test execution. The HWBIST is designed to support executing the full coverage test in pieces to better manage interrupt latency and power. These micro-runs can be executed in smaller time-slices for more efficient task scheduling. During a micro-run, the CPU is isolated from all peripherals and memory. Also, interrupts are logged by the HWBIST controller.
NMI	Nonmaskable interrupt
Parity	A bit added to check the integrity of binary data. A parity bit is calculated and associated with a block of binary data in such a way that the number of 1s is either always even or always odd. Parity enables identification of single bit faults.
PEST	Periodic self-test
PIE	Peripheral interrupt expansion
POR	Power-on reset
POST	Power-on self-test
RAM	Random access memory
Radioactive event	An event in which a single, ionizing particle causes the state of the microelectronics (for example, SRAM) to change.
ROM	Read-only memory

Table 1. Terms and Definitions (continued)

Term	Definition
Screen	The process of exposing newly manufactured components to testing to force latent defects to manifest themselves.
Soft error	A soft error causes bit disturbances in SRAMs due to external events, such as radioactive particle penetration or voltage events in the system power plane.
TMU	Trigonometric math unit
TRM	Technical reference manual
VCU	Viterbi and complex math unit
Voltage event	When the voltage around the component goes outside the data sheet specification for the component. It could be a low-voltage event or a high-voltage event or both. In the case of SRAMs, the voltage event disturbs the value stored in one or more SRAM bit cells.

1.2 CPUMBIST Overview

CPUMBIST stands for CPU (C28x) Memory Built-In-Self-Test. The embedded C28x CPU executes a March13n test on targeted SRAM modules in the device. The C28x CPU also executes CRC testing on targeted ROM modules.

CPUMBIST is not intended to replace or supplement device-manufacture embedded memory testing. The manufacture tests are significantly more aggressive and heavily margined to ensure operation of the embedded memories across the operation range documented in the data sheet for the life of the product. The manufacturing tests are designed to screen out devices with defective or marginal memories. Stress tests are included in the manufacturing tests to accelerate latent defects so that these devices are also screened out.

Instead, CPUMBIST is available to identify embedded memory circuitry that has degraded during system use. This degradation can come from the following conditions:

- Circuit overstress due to voltage or temperature events
- Damage due to handling the components
- Latent defects that were not captured in manufacturing – These are circuit imperfections that were not screened during the manufacturing stress testing, but may fail due to the normal and expected degradation of the semiconductor material over time and use.
- Memory circuit sensitivities due to external noise – Power noise coming from off-chip

While CPUMBIST tests do not specifically target memory bit failures due to radioactive or voltage event-based bit disturbances, these failures are captured in the targeted memories. These failures are captured during the context save of the memory under test by performing memory reads and exercising the EDAC and parity logic.

The CPUMBIST can be executed as part of the start-up or POST of a system, or periodically during the active control operation (PEST). There are fewer system restrictions when executing the test as part of a start-up test. Additionally, if there are memory circuit issues, then it is better to identify them before the control loop begins executing. However, some systems do not shut down often. These systems can schedule and execute memory validation tests, such as the CPUMBIST, while the control loop is active.

2 System Challenges to Memory Validation

Validating embedded memory while the control loop is active presents numerous challenges. Many of these challenges are not an issue at start-up. This subsection does the following:

- Presents system challenges to memory validation
- Explains how the Diagnostic Library provides solutions to these challenges
- Presents system considerations when validating memory for C2000 devices

2.1 Algorithmic Coverage

Testing an SRAM memory instance can be done with a multitude of different algorithms. The device manufacturing tests execute multiple algorithms. The March13n algorithm was chosen for its historically proven coverage of bit cell and addressing faults.

It is certainly possible to execute additional algorithms or make one test target more defect mechanisms, but most, if not all, other defect mechanisms, are time-zero transistor-failure mechanisms and would have already been screened out in the manufacture tests. Some of these more aggressive algorithms suffer from significant execution time requirements (discussed in Section 2.2) as well as significant code memory requirements. Furthermore, these more aggressive algorithms do not provide significant additional coverage in the context of in-system memory failure mechanisms.

While not intrinsically part of the memory validation, the CPUMBIST test suite includes speed testing of the CPU interface to the memory instances. Because the CPUMBIST is executed at the full system clock rate, transistors and interconnects between the SRAM and CPU are also tested at the full system clock rate.

2.2 CPU Overhead of Executing Memory Validation

The March13n algorithm executes a combination of memory accesses for each address in the SRAM. Executing a full March13n algorithm across a 4096-word block of memory takes a significant number of CPU cycles:

See Equation 1, (number of words × test cycles/word) + (number of words × context save cycles/word) + (number of words × context restore cycles/word)

$$(4096 \times 6.5) + (4096 \times 3) + (4096 \times 3) = 51200 \text{ CPU cycles} \tag{1}$$

At 200 MHz, this value comes to approximately 256 µs, which is an overwhelming time-slice during an active control loop. For this reason the memories may be tested in smaller test segments, such as 16 words per execution. With this segmentation the total time involved is higher due to the overhead of branching and function calls, but the time-slice goes down to a more manageable 2.7 µs.

2.3 Interrupt Latency Tied CPUMBIST

Portions of the CPUMBIST time-slice cannot be interrupted because it is difficult to ensure the target of the memory testing is not involved in either an ISR code or the data with which the ISR works. Smaller test segment sizes allow for testing of smaller amounts of memory, but they also allow for less time in which interrupts are masked. For example, a test segment which tests 16 words masks interrupts for approximately 2.5 µs out of the approximately 2.7-µs execution time. The rest of the code segment (approximately 0.2 µs) is interruptible.

It is possible to reconfigure the code to execute with even less latency by the doing following:

- Use an even smaller segment size, for example, eight words:
  - Reduces latency by approximately 40%
  - Reducing to even fewer than eight words comes to diminishing returns.
- Use a March7n algorithm instead of a March13n algorithm:
  - Reduces latency by approximately 50%
  - Reducing below March7n reduces address fault coverage.

2.4 Saving and Restoring the Context of Tested SRAM Instances

To provide high coverage in the SRAM tests, it is necessary to write values to each tested address. Because the memory under test may also be used by the execution of the control loop, the tested memory range must be saved before the execution of the March algorithm and restored afterwards. The memory range under test cannot be read from or written to during the memory segment testing, including during the context save and restore portions. This means that interrupts must be blocked during this time.

At start-up, there is less memory content or context to protect, so running the test at start-up only reduces this overhead. Furthermore, at start-up, a March13n algorithm that does not perform a context save and restore may be appropriate. This reduces the overhead of the memory test.

2.5 CRC Checking

Some information stored in SRAM is static in nature. For example, code and data tables are often loaded at start-up and never changed during control loop execution. When this is the case, it may be less advantageous to exercise the write operations of the SRAM, and more advantageous to perform a simpler CRC test. A CRC calculation over the desired static memory contents diagnoses memory faults. This safety mechanism may be adequate and even more effective to indicate a memory fault in that range. CRC tests are included in the Diagnostic Library along with the March13n tests.

The code provided in the Diagnostic Library also includes CRC algorithms for testing static portions of the memory space. This includes ROM and flash. Additionally, there may be some static code or tables that can be tested with CRC. The trade-offs of CRC checking versus March13n testing follow:

Advantages of CRC over March13n:

- CRC takes fewer cycles per word than the March13n algorithm.
- CRC is read only, so there is no need for a context save and restore.
- CRC has negligible interrupt latency affect, because it does not disable interrupts.
- March13n memory requirements increase as the memory range under-test increases due to the context save and restore.

Advantages of March13n over CRC:

- March13n tests the write capability of the SRAM.
- March13n tests bit cell speed paths.
- March13n algorithm requires less memory overhead than a CRC algorithm without the VCU, which uses look-up tables.

2.6 Management of Memory Resources

Some of the embedded memory targeted for validation may be shared by multiple processors. Sharing of these memory resources not only suffers from the memory test violating the context of the execution of the other processor, but may also alter the execution time of the other processor. The following subsections describe this phenomenon and how it is managed. Table 2 lists the embedded memories that are accessible to the C28x cores of the F28x7x series of C2000 devices. The F2837xD superset device is used in Table 2. The memories listed in blue are covered by the CPUMBIST and those listed in red are not.

Table 2. Embedded Memory Table for F2837xD

Memory Name	Description	CPU Start Address	CPU End Address	Protection	Comments
RAM M0, RAM M1	CPU local	0x0000	0x07FF	EDAC	
PIE Vector Table	PIE RAM vector table	0x0D00	0x0EFF	Hardware redundancy	This may be tested for latent faults at start-up. This does not need to be tested due to the redundant PIE RAM table. Additionally, in-system tests become difficult due to possibly active DMA interrupts.
CLA Msg RAM	CLA message RAM	0x1480	0x157F	Parity	This may be tested, but the user must ensure that the message RAM that is actively in use by the CLA tasks is not concurrently being corrupted by the memory test.
uPP RX	UPP receive FIFO	0x6C00	0x6CFF	None	
uPP TX	UPP transmit FIFO	0x6E00	0x6EFF	None	

Table 2. Embedded Memory Table for F2837xD (continued)

Memory Name	Description	CPU Start Address	CPU End Address	Protection	Comments
RAM LS	Local shared	0x8000	0xAFFF	Parity	
RAM D0, RAM D1	CPU local	0xB000	0xBFFF	EDAC	
RAM GS	Global shared	0xC000	0x1BFFF	Parity	
IPC Msg RAM	IPC message RAM	0x3F800	0x3FFFF	Parity	This may be tested, but the user must ensure that message RAM that is actively in-use by the other CPU is not concurrently being corrupted by the memory test.
USB	USB buffers	0x41000	0x417FF	None	
CAN msg RAM	CAN message RAM	0x48800	0x4B7FF	Parity	

2.6.1 Dedicated Memories (RAM M0, M1 and RAM D0, D1)

Each C28x CPU includes memories that are dedicated to the C28x. These include the M0, M1, D0, and D1 memory instances. These memories are equipped with EDAC mechanism. Therefore, TI recommends using these memories for the system stack as well as other safety critical code or data. This may include the March13n code. In this case, to test the memory containing the March13n code, a redundant copy of the code must be placed elsewhere and used to execute the test on the memory containing the original code. Furthermore, the data activity in these dedicated memories, combined with the EDAC checking, provides high fault coverage comparable to that of a March7n algorithm.

Key takeaway:

- Dedicated memories contain EDAC.
- To test the program memory containing the CPUMBIST, a redundant copy of the program code must be stored elsewhere and executed.

2.6.2 Local Shared Memories

Each C28x CPU includes local shared (LS) memories. These memories can be shared between the C28x, FPU, VCU/CRC, TMU, and the CLA processors. However, while the C28x is executing the March13n kernel code, none of the acceleration units (FPU, VCU, and TMU) are active. Therefore, only the CLA will share the memory. As part of the initial memory system configuration performed by the C28x, the LS memories are individually configured as dedicated to the C28x, FPU, VCU, and TMU or shared with the CLA. This configuration is managed by the C28x, which can write to LSxMSEL, the C28x addressable configuration register. The user must know the state of each LS memory. If the CLA is the master of a memory block, the user also must be mindful of whether that memory block is CLA program memory or CLA data memory. Memories which are actively owned by the CLA and configured as CLA program memory cannot be tested by C28x executing the CPUMBIST. The C28x access to CLA program memory is blocked. For more information, see the respective Technical Reference Manual. For example, see the *TMS320F2837xD Dual-Core Delfino Microcontrollers Technical Reference Manual*.

Additionally, it is possible to disable C28x writes to portions of the LS memories. If CPU writes are disabled using the LSxACCPROTx registers, the CPUMBIST cannot be performed on the memories that have disabled CPU writes. The user must not set the CPUWRPROT bits for the memories they wish to test because write access is required by the CPU. Additionally, if CPU fetches are disabled using the LSxACCPROTx registers, then the CPUMBIST, or any other C28x code for that matter, cannot be executed from those memories. The user must not set the CPUWRPROT bits for memories which contain C28x program memory.

NOTE: Concurrent accesses by the C28x and CLA to the same LS block stall the CLA due to the memory arbitration. This may result in performance degradation of the CLA execution.

Key takeaway:

- Know the state of each LS memory.
- Ensure that the C28x execution of the CPUMBIST test on LS memories used by the CLA is safe.
- C28x cannot write to LS memories which are configured as CLA program memory.
- C28x cannot write to LS memories which have writes disabled by LSxACCPROTx registers.
- C28x cannot execute code from LS memories which have fetches disabled by LSxACCPROTx registers.

2.6.3 Global Shared Memories

These devices also include global shared (GS) memories. These memories can be configured between each CPU on a dual-core device (F2837xD). These memories are shared between the CPU and its associated DMA processor. Any address range of a GS memory that is shared with the DMA unit must be specially considered when memory testing. The user may wish to avoid performing a CPUMBIST memory test on memory ranges that are accessed by DMA channels. Another option is to turn off the DMA channel when testing the range of GS memory used by the DMA. However, the main justification for using DMA is to move data in a timely manner. Therefore, this option is usually less acceptable.

The dual processor version (F2837xD) allows the sharing of GS memories between the two embedded C28x cores. Only one C28x CPU can be mapped to write to a specific GS memory (as defined in the GSxMSEL register), but the other C28x CPU can read from it. Therefore, the master CPU of a particular GS memory block can perform the CPUMBIST, while the other CPU is unable.

However, if the C28x CPU that does not have write access enable is actively reading a portion of the memory, then that memory range should be blocked from being tested by the other C28x. To avoid blocking out large portions of GS memories, it is possible to flag the opposing C28x to avoid the testing of a memory range through the IPC resources. Similar controls must be in place to protect the shared IPC message RAMs.

It is possible to disable writes to a GS block even if the GS block is mapped to the CPU. This is accomplished by the use of write protect bits, CPUWRPROT, which are in GSxACCPROTx registers. The setting of these bits behaves similarly to those for the LSx memories previously discussed. Similar consideration must be made for these memories.

Key takeaway:

- Know the state of each GS memory.
- Ensure that the C28x execution of the CPUMBIST test on GS memories used by the DMA or the other CPU of dual core devices is safe.
- C28x cannot write to GS memories which have the other CPU configured as the master on dual-core devices.
- C28x cannot write to GS memories which have writes disabled by GSxACCPROTx registers.
- C28x cannot execute code from GS memories which have fetches disabled by the GSxACCPROTx registers.

2.6.4 PIE RAM Vector Table

The PIE Vector Table includes full hardware redundancy. When a vector is fetched by the PIE controller, it is fetched from two redundant tables. If there is a miss-comparison between the dual vector fetches, then the CPU generates an NMI (NMIFLG.PIEVECTERR). On the dual-core devices this NMI is generated to both C28x CPUs. Due to the seriousness of such an error, this event also generates a TRIP event to the PWM modules. This event is described in detail in the Technical Reference Manual for the device-specific documents. For example, see the *TMS320F2837xD Dual-Core Delfino Microcontrollers Technical Reference Manual*.

Key takeaway:

- The PIE RAM vector table has hardware redundancy.

2.6.5 CLA Message RAM and IPC Message RAM

The CLA message RAM is in place to pass commands and data between the C28x CPU and the CLA processor. The fundamental use of the CLA is to offload operations from the CPU to be executed in parallel. Half of this memory is for passing data from the CLA to the C28x, and the other half is for passing data from the C28x to the CLA. The C28x CPU can test the second half, but it is suggested that the CLA not read this during CPUMBIST execution. While it is possible for the CLA to read the C28x to CLA message RAM, two negative things are possible. First, the data may be corrupted by the CPUMBIST during the CLA access. This may cause the CLA to receive corrupted data. Second, the CLA performance may degrade because of the concurrent memory accesses by both processors. These aspects must be considered when determining whether or not to integrate and execute the CPUMBIST on the CLA Message RAM.

This memory includes parity protection, and upon failure it generates an NMI error to the C28x. It is possible to add C28x testing of the second half of this memory using the March13n kernel code. The two previous aspects must be considered. Managing the CPUMBIST of this memory can be achieved through the C28x and CLA communication structure that uses this message memory. Details for this are available in the Technical Reference Manual.

Additionally, there are similar IPC message RAMs between CPU1 and CPU2 on dual-core devices. This memory includes parity protection. Both CPUs can write and test only its message RAM with write access. However, the previous considerations for the CLA message RAM must be considered.

Key takeaway:

- Know the state of each message memory.
- Ensure that the C28x execution of the CPUMBIST test on message memories, which are used by the CLA or other CPU on dual-core devices, is safe.
- CLA message RAMs have 1-way write access.
- IPC message RAMs have 1-way write access.

2.6.6 Peripheral Memories

While the C28x CPU can access and test the memories embedded in the peripherals, it is not feasible to test these memories while the peripherals are running. This is due to the asynchronous nature between the CPU and the peripheral access of these memories. This is not a serious coverage hole, because most of the peripheral memory use involves protocols that check the data packets. In peripherals that keep track of data error rates, the C28x can monitor these counters to gain an indication of circuit failure inside the peripheral data path. This is a more effective monitor of the health of the peripheral than is specifically testing the SRAM buffer while the peripheral is active.

The peripheral memories include the following memories listed in Table 2.

- **uPP Receive and Transmit FIFOs:** The universal Parallel Port (uPP) is a buffered DMA engine which minimizes the overhead of interfacing between the CPUs and the ADC/DACs, as well as external peripherals such as FPGAs. The uPP manages the synchronizing between these peripherals and the CPU clock world. The uPP FIFO buffers can be tested at start-up or during system maintenance periods.
- **USB Buffers:** The universal serial bus (USB) peripheral is the master of this buffer SRAM. For the CPU to test the memory, this peripheral must be shut down. The data is highly transient in nature, meaning it does not stay in the memory for long because it is transmitted out or transferred into the system memory. Transient data is less susceptible to event-driven upsets (soft errors) because it is not in the memory for long. For more permanent failure mechanisms, the USB is a standardized communications protocol stream that includes CRC data flow checking of data packets. Receive packets include CRC values that are checked against the data. If there is a CRC error, the USB port automatically requests a retransmit of the data packet. If this fails three consecutive times, then the ERROR bit is set in the USBCSRLO register. Monitoring this and other bits in this register provides an indication that a circuit issue may exist in the USB path if there are repeated errors. Detailed descriptions are available in the USB chapter of the Technical Reference Manual. The transmitted data packets are not as well protected because the CRC is calculated after the data is loaded into the buffer memory. Memory may fail during the load of this buffer or during memory read for the CRC calculation that will corrupt the CRC value consistent with the memory corruption.

- **CAN Message RAM:** The CAN controller is a communications port similar to the USB in that the peripheral is the master of the memory, and therefore any C28x testing of this memory requires shutdown of the port. The CAN controller is also similar to the USB in that it includes CRC checking of the data passing through the buffer CAN Message buffer RAM. In addition, the CAN Message RAM includes parity protection. A system can monitor the CAN_PERR register for repeat fails at a specific address to identify likely circuit failure.

Key takeaway:

- CPUMBIST should not be executed on peripheral memories.
- Other safety mechanisms must be implemented.

2.6.7 Maintaining Confidence in CPU Code Executing CPUMBIST

In F28x7x devices, the code is executed from memory that includes either parity or EDAC protection. The Diagnostic Library provides functions for monitoring both parity and EDAC failures, including correctable and uncorrectable errors. The context save operation also detects soft errors that have occurred since the last time the targeted area was read. Because these failures can be tied to soft errors that bear no indication of circuit failure, a test failure does not necessarily indicate a damaged device. Therefore, error management must consider this. For example, when a failure is captured, the code can identify the error address. Soft errors are random and occur infrequently. Therefore, multiple errors captured at the same address are indicative of circuit marginality and the need for replacement of the component in the system. Furthermore, due to the infrequent nature of soft errors, keeping a running count on soft errors is another valid way of indicating failing circuitry, rather than naturally occurring radioactive events. Such voltage events should be detected by a system voltage monitor or other voltage related safety mechanism.

While the use of parity checking for bit failures in the memory may miss a multibit error in the memory, these are highly unlikely due to a memory circuit failure. The individual bits of a memory read are not physically close to each other, so a single defect or disturb event does not affect more than 1 bit in a word read. Therefore, a degradation or latent defect which affects multiple bits fails in a manner where multiple words experience single-bit failures, instead of a single word experiencing multiple bit failures. The possibility of a multibit failure in the same word read/write almost certainly indicates a voltage event outside the specified operating range of the device. The memory test is not targeted for identifying such voltage events.

The CPUMBIST algorithm may use four provided background or test patterns over the execution across the tested memory ranges. These background test patterns guarantee that both the 32 bits of the memory word and the parity bits protecting the word get full bit coverage. The background patterns force transitions of a EVEN/EVEN, EVEN/ODD, ODD/EVEN, ODD/ODD parity. The address parity bit transitions naturally across the even, odd addresses. TI recommends that the user choose to cycle through these background patterns when testing memory ranges.

Additionally, errors may be injected for validation of the CPUMBIST. Errors may be injected using the TEST registers in the MEM_CFG_REGS register file. Error injection techniques are discussed further in Section 4.1.2.

3 CPUMBIST Working In-System

The CPUMBIST can be executed in two different ways:

- Execute the CPUMBIST as a background task that the control loop tasks interrupt.
  - The advantage is that the memory testing is only conducted when there is idle time.
  - See Figure 1 for an example of executing the CPUMBIST as a background task.
- Execute the CPUMBIST as a time-slice in the control loop.
  - The advantage is that the time it takes to complete a cycle across all memory ranges is controlled and well known.
  - See Figure 2 for an example of executing the CPUMBIST as a time-slice in the simple example of a control loop.

This choice is left to the system integrator based on the nature of the control loop.

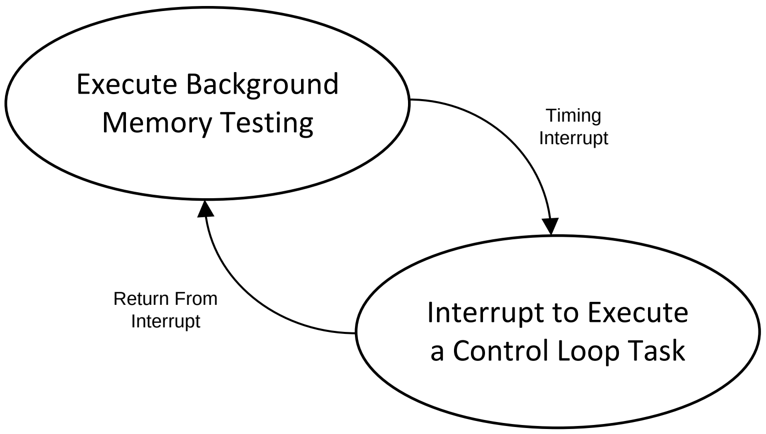
Execute Background Memory Testing
Timing Interrupt
Interrupt to Execute a Control Loop Task
Return From Interrupt

Figure 1. Execute CPUMBIST as Background Program

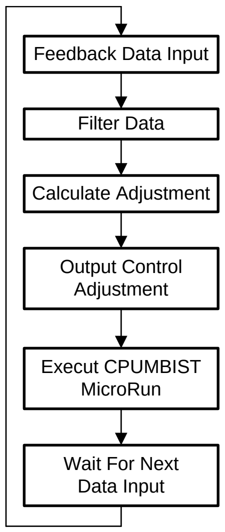

Figure 2. Execute CPUMBIST as Time-Slice

Regardless of which method of calling is used for executing the CPUMBIST time-slices, the function call and program execution of the CPUMBIST remains the same. The user may define how large the memory range under test is for each execution of the CPUMBIST. This allows the user to tune the timing of the tests in relation to the control loop.

3.1 Memory Under-Test Considerations

There may be memories which the user is unable to or does not desire to test in the system. Reasons for this may include the previous considerations in Section 2.6. However, for those memory ranges which are not specifically tested by the CPUMBIST March13n algorithm, the parity or EDAC checking still exists. Therefore, a bit failure in these memories is still captured and detected in the hardware. An uncorrectable error generates an NMI and once a threshold of correctable errors is met, a separate interrupt is generated. The user can handle these failures using the software ISR. Furthermore, memories which are difficult to test PEST may be more easily tested at POST, start-up, or during maintenance cycles of the system.

The user may also wish to account for the transient nature of some memories when considering their safeness. For example, a memory range which is not static, but dynamic or transient, may be difficult to test with the CPUMBIST. However, the transient nature of the memory may provide some level of coverage besides the parity or EDAC. Although the memory range is not guaranteed to have the coverage provided by the CPUMBIST, it does have some level of inherent coverage in the random nature of the data passing through and checked using parity, EDAC, or a CRC.

Memories which may fall into this category include:

- LS memories shared with the CLA configured as CLA data memory
- GS memories shared with the DMA
- General RAM memories which store transient data

NOTE: The March13n algorithm is not affected by physical memory boundaries.

3.2 System Example: Blocking Memories From Test

As discussed in the previous section, some memory address ranges of the available SRAM may not be tested by the CPUMBIST in-system because they are shared with other resources. A user may block these memories from being tested. The philosophy behind blocking certain memories involves prioritizing system performance over in-system memory coverage. This subsection provides an example of quantifying this decision:

Assume the following system configuration:

- CPU1 owns GS0-7
  - DMA channels in CPU1 own four address ranges inside the GS address space with 256 words each.
    - This means out of 32384 words, 31744 words (approximately 97%) are specifically targeted by the in-system test.
    - The remaining approximately 3% are still monitored by parity and is transient so this checking is often executed.
- On CPU1, the C28x owns four LS blocks and shares two blocks with the CLA.
  - If the micro-run test code is in the LS memory and takes approximately 60 words, then this means that out of 12288 words, 8132 words (approximately 66%) are specifically targeted by the in-system test.
  - The remaining 44% is still monitored by parity, but is not necessarily as transient in nature.
- On CPU1 there are 6144 words of dedicated memory.
  - If the micro-run test code is in dedicated memory and takes approximately 60 words, then this means out of 6144 words, 6084 words (approximately 99%) are specifically targeted by the in-system test.
  - The remaining < 1% are self corrected by EDAC circuits.
- In total the CPU1 has approximately 90% of these memories specifically targeted by the in-system testing.

If this is not adequate for the safety of the system, then portions of the background activity must be stopped for further coverage. This means the CLA and DMA operation must be stopped or moved to other memories periodically so that the CPU can target their memory resources. This example analysis can be expanded to include the peripheral and CLA message SRAM as well.

3.3 Micro-Run

As previously mentioned, the actual memory testing may be executed in small segments called micro-runs. These are executed across a range of memory, for example the LS or GS memories, by the C28x CPU.

The CPUMBIST micro-run function is a hand-optimized assembly function which is C-callable. The assembly language does the following:

- Context saves N-number of 32-bit words to a copy memory region.
- Disables interrupts globally (DINT)
- Executes the reads and writes (march elements) of the March13n algorithm:
  - The code does not validate the reads, but instead allows the Parity or EDAC circuitry to validate the reads.
  - The Parity and EDAC circuitry is validated using the error injection passes embedded in the test.
- Context restores the N-number of 32-bits words from the copy memory region.
- Enables interrupts globally (EINT)
- Return

NOTE: The CPUMBIST algorithm and API is described in greater detail in Section 4.

NOTE: A fault may generate an interrupt depending on the type of fault. For Parity protected memories, a single-bit fault generates an uncorrectable memory failure and an NMI. For EDAC protected memories, a multibit failure generates an uncorrectable memory failure and an NMI. For EDAC protected memories, a single-bit fault generates a correctable memory failure event and increments a counter of correctable errors. When this counter equals or exceeds a software-programmable counter, a correctable error interrupt is generated.

4 About the Code

This section describes the APIs of the CPUMBIST and their functionality. This section should be used to aid in the system integration of the CPUMBIST, CRC test, and other memory validation techniques. This section should be used along with the Diagnostic Library User’s Guide, and the Self-Test Application User’s Guide. Both of these additional documents are available in the C2000 SafeTI Diagnostic Library software release package.

4.1 CPUMBIST Code

The CPUMBIST code included in the Diagnostic Library includes four APIs:

- STL_March_testRAMCopy() – Performs March13n algorithm with context save and restore.
- STL_March_testRAM() – Performs March13n algorithm without context save and restore.
- STL_March_checkErrorStatus() – Checks the MEMCFG registers for any errors.
- STL_March_injectError() – Injects an error into the RAM.

The source code and header files for these functions are in the following:

- stl_march.c – source file
- stl_march_s.asm – source file
- stl_march.h – header file

4.1.1 March13n Functions

As previously stated, two functions perform the March13n algorithm on the RAM. The following are the API definitions and descriptions.

4.1.1.1 STL_March_testRAMCopy()

This function performs a March13n, nondestructive, memory test on the specified RAM memory range:

```
void STL_March_testRAMCopy(const STL_March_Pattern pattern,
                                  const uint32_t startAddress,
                                  const uint32_t length,
                                  const uint32_t copyAddress);
```

Input parameters:

- STL_March_Pattern pattern – The test pattern to use during the test.
- uint32_t * startAddress – The address at which to start the memory test.
- uint32_t length – The number of 32-bit words to test, minus one.
- uint32_t * copyAddress – The address at which to copy the original contents of the memory under test. The address is used to save and restore the original memory at the end of the March13n memory test.

NOTE: The length is the number of 32-bits words to test, minus one. For example, to test eight 32-bit words, the length is 7.

This function performs a March13N memory test on the RAM specified by the *startAddress* and *length* input parameters. This function performs a nondestructive memory test. This means that it begins by copying the original contents of the memory to *copyAddress*, perform the memory test, and then copies the original contents back to the memory under test. The test patterns and the March13n memory test algorithm provided, test memory for stuck-at-faults as well as boundary cases including worst-case timings tailored for the C2000 RAM bank architecture.

This test is implemented to be able to perform a memory test on any section of the RAM including the stack.

If this code is running from the RAM, be careful to not perform this memory test on itself, meaning do not perform the March13n memory test on the March13n program code in the RAM. This results in undefined behavior and likely leads to ITRAP. To test the program code for this March13n algorithm, the user can create a copy of this function in the RAM or flash, and run the memory test code from the redundant copy.

This function disables global CPU interrupts (DINT) and then reenables them after the test has completed. This function returns void. Figure 3 shows a flow chart of the STL_March_testRAMCopy() function.

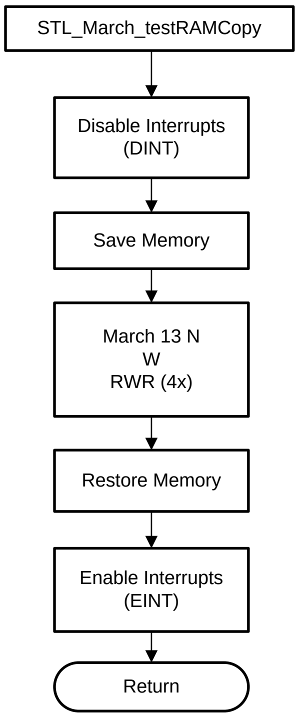

Figure 3. STL_March_testRAMCopy() Flow Chart

4.1.1.2 STL_March_testRAM()

This function performs a March13n destructive memory test on the specified RAM memory range.

```
void STL_March_testRAM(const STL_March_Pattern pattern,
                        const uint32_t startAddress,
                        const uint32_t length);
```

Input parameters:

- STL_March_Pattern pattern – The test pattern to use during the test.
- uint32_t * startAddress – The address at which to start the memory test.
- uint32_t length – The number of 32-bit words to test, minus one.

NOTE: The length is the number of 32-bit words to test, minus one. For example, to test eight 32-bit words, the length is 7.

This function performs a March13n memory test on the RAM specified by the *startAddress* and *length* input parameters. This test performs a destructive memory test, meaning the original contents are lost by this test. The test patterns and the March13n memory test algorithm provided, test memory for stuck-at-faults as well as boundary cases including worst-case timings tailored for the C2000 RAM bank architecture.

If this code is running from the RAM, be careful to not perform this memory test on itself, meaning do not perform the March13N memory test on the March13N program code in the RAM. This results in undefined behavior and likely leads to ITRAP. To test the program code for this March13n algorithm, the user can create a copy of this function in the RAM or flash, and run the memory test code from the redundant copy.

This function disables global CPU interrupts (DINT) and then reenables them after the test has completed. This function returns void.

Figure 4 shows a flow chart of the STL_March_testRAM() function.

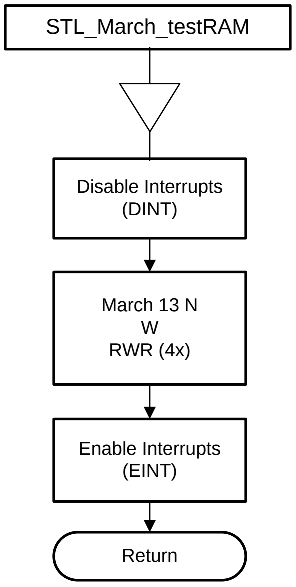

Figure 4. STL_March_testRAM()

4.1.1.3 Test Patterns

Four test patterns are provided by the Diagnostic Library in stl_march.h. These patterns are of enumerated type STL_March_Pattern. The following is a snapshot of the test patterns in stl_march.h.

```
//
//! Values that must be used to pass to determine the test pattern for
//! STL_March_testRAMCopy() and STL_March_testRAM()
//
//
typedef enum
{
    STL_MARCH_PATTERN_ONE    = 0x96966969U,    //!< Test Pattern One
    STL_MARCH_PATTERN_TWO    = 0x0000FFFEU,    //!< Test Pattern Two
    STL_MARCH_PATTERN_THREE  = 0x2AAA5555U,    //!< Test Pattern Three
    STL_MARCH_PATTERN_FOUR   = 0xCC3723CCU     //!< Test Pattern Four
} STL_March_Pattern;
```

Using all of these test patterns provides the maximal fault coverage. These four test patterns guarantee that both the 32-bits of the memory word, and the parity bits protecting the word, get full bit coverage. The test patterns force transitions of an EVEN/EVEN, EVEN/ODD, ODD/EVEN, ODD/ODD parity. Furthermore, the address parity bit transitions naturally across the even odd addresses. TI recommends that the user choose to cycle through these test patterns when testing memory ranges.

4.1.1.4 Optimization of the Test RAM Functions

The previous two March13n algorithms are highly optimized. These optimizations include both hardware and software co-design optimizations.

- **Software:** First, these functions are hand coded in assembly and optimized for register use and data accesses. As previously noted, these functions can safely test stack memory. This is due to the efficient register use. The assembly function begins by popping all input parameters off of the stack and into internal CPU registers. The function then completes without having to access the stack. This means that after popping all of the input parameters off of the stack and into internal CPU registers, the algorithm can safely test the stack. The code performs a context save of the memory contents, corrupts the stack memory under test, and then restores the stack before any subsequent stack accesses occur.
- **Hardware:** Second, these functions take into account that all RAMs on F28x7x devices are equipped with either parity or EDAC. This prevents the C28x from having to compare the data read back after writing it for each memory address. This means that the C28x can avoid many CPU cycles necessary for compare and branch instructions. The result is an efficient March13n algorithm which performs each of the march elements sequentially and allows for a single memory error or memory status check following the end of the test. Additionally, if an uncorrectable error is detected by the hardware during the test, an NMI is generated and the C28x services the interrupt immediately. Correctable errors which are less dangerous may be serviced at a later and more convenient time.

The previous optimizations result in the following:

- Efficient code execution
- Testing of SRAM
- Testing of parity and EDAC safety mechanisms
- Efficient fault handling by the system

These features, with the higher fault coverage achieved by the March13n algorithm, make the CPUMBIST on F28x7x devices an attractive safety solution for a wide variety of systems.

4.1.2 Error Injection

The Diagnostic Library provides an error injection function. This allows the user to inject errors into the memory and ensure that the software and hardware is detecting the inserted errors.

This function injects an error at a specific RAM memory address.

```
void STL_March_injectError(const STL_March_InjectErrorHandle errorHandle);
```

Input parameter:

- STL_March_InjectErrorHandle errorHandle – An inject error handle specifying where and what type of error to inject into the RAM

STL_March_InjectErrorHandle is a pointer to a STL_March_InjectErrorObj structure defined in stl_march.h. The following are their definitions:

```
//
//! \brief Defines the March memory test inject error object.
//!
typedef struct
{
    uint32_t            address;              //!<Address (32-bit aligned)
    uint32_t            ramSection;           //!<RAM section identifier
    uint32_t            xorMask;              //!<Mask to flip bits in test mode
    MemCfg_TestMode     testMode;             //!<Mode in which to inject error
} STL_March_InjectErrorObj;
//
//! Defines the RAM error logic test handle
//!
typedef STL_March_InjectErrorObj * STL_March_InjectErrorHandle;
```

This function injects an error at a specific memory address in a specific RAM block provided by the input members *address* and *ramSection*. The input member *testMode* specifies whether the error is injected in the data or ECC/Parity bits. The input *xorMask* specifies which bit or bits to flip to corrupt either the data or ECC/Parity bits.

This function returns void. Figure 5 shows a flow chart of the STL_March_injectError() function.

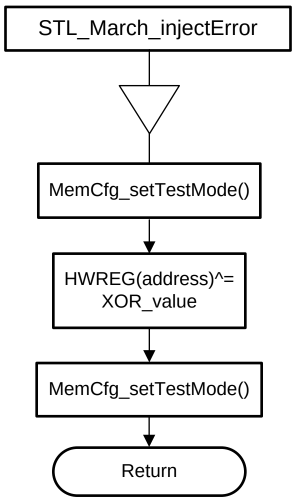

Figure 5. STL_March_injectError() Flow Chart

NOTE: Errors in the RAM are detected by the parity and EDAC hardware on memory reads. Therefore, only STL_March_testRAMCopy() can detect these injected errors. This is because it performs a memory copy during the context save. STL_March_testRAM() is unable to detect these injected errors because it begins the March13n by writing to memory, therefore overwriting the injected error before it is read by the CPU.

4.1.3 Error Handling

The Diagnostic Library provides a function for checking the memory error status registers. This function allows the check to see if any memory errors were detected by the Parity and EDAC hardware on the RAMs.

```
uint16_t STL_March_checkErrorStatus(void);
```

This function also checks if there are any correctable or uncorrectable errors indicated by the memory error registers and returns the status. Additionally, if any error is detected then the STL_UTIL_MARCH global error flag is set.

Input parameter: none

This function checks if there are any correctable or uncorrectable errors indicated by the memory error registers and returns the status. Additionally, if any error is detected then the STL_UTIL_MARCH global error flag will be set.

This function returns uint16_t. If the memory error registers indicate a correctable error and/or an uncorrectable error in RAM, then the function returns STL_MARCH_CORR_ERROR, STL_MARCH_UNC_ERROR, or STL_MARCH_BOTH_ERROR. Otherwise, the function returns STL_MARCH_PASS.

Figure 6 shows a flow chart of the STL_March_checkErrorStatus() function.

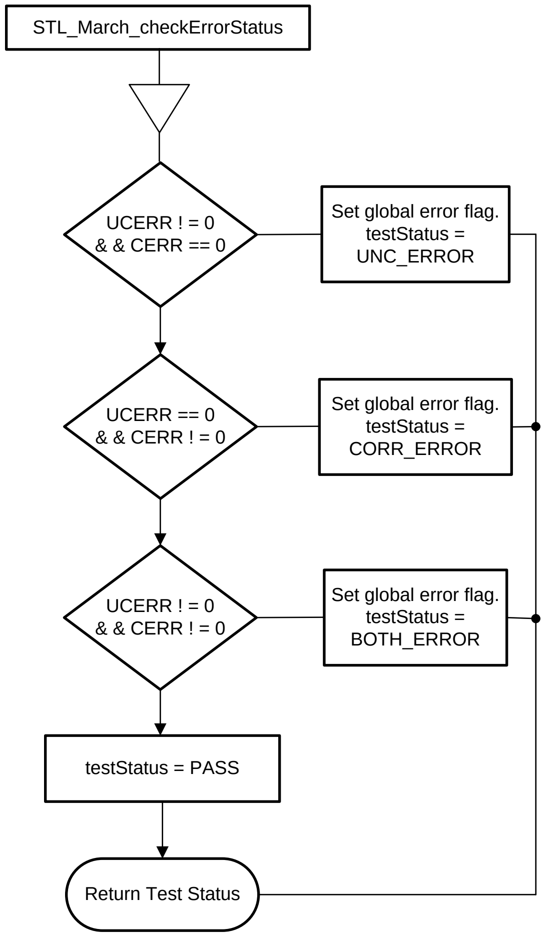
STL_March_checkErrorStatus
UCERR ! = 0 & & CERR == 0
Set global error flag. testStatus = UNC_ERROR
UCERR == 0 & & CERR ! = 0
Set global error flag. testStatus = CORR_ERROR
UCERR ! = 0 & & CERR ! = 0
Set global error flag. testStatus = BOTH_ERROR
testStatus = PASS
Return Test Status

Figure 6. STL_March_checkErrorStatus() Flow Chart

As previously mentioned, the CPUMBIST makes use of the Parity and EDAC circuitry to identify failing bits in the SRAM testing. This means that uncorrectable errors generate an NMI, and when the number of correctable errors meets or exceeds a software programmable threshold, a correctable error interrupt is generated. The following are some software examples to aid the customer in handling these errors in their system.

4.1.3.1 Enable and Register NMI ISR and Correctable Error ISR

When developing and testing code with the debugger connected, the C28x CPU may not have executed through Boot ROM which enables the NMI. Therefore, the user may need to enable the NMI in their system during development. Additionally, the user needs to register their own NMI handler or ISR. The following is an example of code to do this.

```
//
// Enable NMI if loading from debugger and absent from GEL file.
//
EALLOW;
HWREGH(NMI_BASE + NMI_O_CFG) |= NMI_CFG_NMIE;
EDIS;
//
// Register NMI ISR.
//
Interrupt_register(INT_NMI, STL_March_errorNMIISR);
```

Additionally, users may wish to enable and register a correctable error interrupt. The following is an example of code to do this.

```
//
// Enable a Correctable RAM Error ISR.
//
Interrupt_register(INT_RAM_CORR_ERR, STL_March_corrErrorISR);

//
// Enable a Correctable RAM Error ISR in the PIE.
//
Interrupt_enable(INT_RAM_CORR_ERR);
//
// Enable a Correctable RAM Error ISR.
//
MemCfg_enableCorrErrorInterrupt(MEMCFG_CERR_CPUREAD);
```

4.1.3.2 Uncorrectable Error Handler

When an uncorrectable error is encountered, a NMI is generated and the C28x begins to execute the NMI ISR. The following is an example NMI handler code. The user may wish to log the useful information from the uncorrectable error and then safely shut down the system.

```
//*****************************************************************************
//
// STL_March_errorNMIISR(void)
//
//*****************************************************************************
__interrupt void STL_March_errorNMIISR(void)
{
    //
    // Check for Uncorrectable RAM Error.
    //
    if((SysCtl_getNMIFlagStatus() & NMI_FLG_RAMUNCERR) == NMI_FLG_RAMUNCERR)
    {

        //
        // Clear the Uncorrectable RAM Error NMI flag and NMIINT flag if it is
        // the only flag left.
        //
        SysCtl_clearNMIStatus(NMI_FLG_RAMUNCERR);

        //
        // Check what cause the Uncorrectable RAM Error: C28x, CLA, DMA.
        //
        uint32_t ucerrType = HWREG(MEMORYERROR_BASE + MEMCFG_O_UCERRFLG);

        //
        // Get the address of the Uncorrectable RAM Error.
        //
        uint32_t cpuUCAddress = HWREG(MEMORYERROR_BASE + MEMCFG_O_UCCPUREADDR);
        uint32_t dmaUCAddress = HWREG(MEMORYERROR_BASE + MEMCFG_O_UCDMAREADDR);
        uint32_t claUCAddress = HWREG(MEMORYERROR_BASE + MEMCFG_O_UCCLA1READDR);

        //
        //** ToDo:
        //** Add code to manage the Uncorrectable Error fault and
        //** gracefully shut down system.
        //
    }
}
```

4.1.3.3 Correctable Error Handler

When the number of correctable errors has met or exceeded a software programmable threshold, a correctable error interrupt is generated and the C28x begins to execute the ISR. The following is an example ISR handler code. The user may wish to log the useful information from the uncorrectable error and monitor the location of the failure.

```
//*****************************************************************************
//
// STL_March_corrErrorISR(void)
//
//*****************************************************************************
__interrupt void STL_March_corrErrorISR(void)
{
    //
    // Check what cause the Correctable RAM Error: C28x, CLA, DMA.
    //
    uint32_t cerrType = HWREG(MEMORYERROR_BASE + MEMCFG_O_CERRFLG);

    //
    // Get the address of the Correctable RAM Error.
    //
    uint32_t cpuCerrAddress = HWREG(MEMORYERROR_BASE + MEMCFG_O_CCPUREADDR);

    //
    // Get the Correctable RAM Error count.
    //
    uint32_t cerrCount = MemCfg_getCorrErrorCount();

    //
    //** ToDo:
    //** Add code to manage the Correctable Error fault and
    //** monitor the location of the fault.
    //

    //
    // Clear the Correctable RAM Error status.
    //
    MemCfg_clearCorrErrorInterruptStatus(MEMCFG_CERR_CPUREAD);

    //
    // Acknowlege the PIE Group.
    //
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP12);
}
```

Dedicated RAM on F28x7x devices is equipped with EDAC. Therefore, single-bit errors are corrected by the hardware. In a system, the expected number of single-bit error corrections is sensitive to multiple factors including altitude, working environment, and so on. Therefore, the system developer and integrator must take these factors into consideration. Except under extreme conditions, like outer space or in the presence of radioactive emissions from other materials/equipment local to the system, the number of correctable errors should be quite low over the life of the product. It is not reasonable to consider a single correction as a system failure. Furthermore, if the CPUMBIST is executed over the memory containing the single-bit error, then the error is corrected by the memory copy.

4.1.4 Performance

Table 3 lists the performance of the CPUMBIST on the F28x7x series of devices with Parity and EDAC on the RAMs while executing from RAM. The recorded time, in microseconds, corresponds to either F2837xD or F2837xS devices running at 200 MHz.

Table 3. CPUMBIST Performance Chart

Length	Number of Words	STL_March_testRAMCopy()		STL_March_testRAM()	
		Cycles	Time (µs)	Cycles	Time (µs)
3	8	299	1.50	247	1.24
7	16	529	2.65	427	2.14
15	32	983	4.92	787	3.94
31	64	1897	9.49	1507	7.54
2047	4096	51272	256.4	26691	133.46

NOTE: Users must keep in mind that performance may degrade when executing the CPUMBIST from flash or wait-stated memories.

4.2 CRC Code

Data integrity is an important component of safe systems, whether it is for communications or code or data. CRC calculation and check is a useful safety mechanism for memory validation. A CRC check may be more suitable for memory which is static. For example, static code or static data may not require the memory validation offered from a CPUMBIST, but a more simple CRC calculation may suffice.

The Diagnostic Library provides CRC functions to be used for data integrity checks. On F2837xD and F2837xS devices, the CRC software takes advantage of the VCU and accelerated CRC instructions. On F2807x devices, the CRC software uses a look-up table to speed up the calculation.

4.2.1 STL_CRC_checkCRC()

The Diagnostic Library provides a useful API for executing a CRC calculation and check against a golden CRC value. This function calculates a 32-bit CRC value for a specific memory range and compares it with the golden CRC value. The polynomial used is 0x04c11db7.

```
uint16_t STL_CRC_checkCRC(const uint32_t startAddress,
const uint32_t endAddress,
const uint32_t goldenCRC)
```

Input parameters:

- uint32_t startAddress – The start address of the CRC calculation.
- uint32_t endAddress – The end address of the CRC calculation, inclusive.
- uint32_t goldenCRC – The golden CRC value to compare the calculated value against.

This function performs a test of the memory range by calculating the 32-bit CRC value for the input memory range and comparing it against the golden CRC value.

NOTE: This function can be used to validate many memory types, including flash and boot ROM.

This function, STL_CRC_PASS, if the calculated CRC matches the golden CRC value. Otherwise, the function returns STL_CRC_FAIL and sets the STL_UTIL_CRC global error flag. Figure 7 shows a flow chart for the STL_CRC_checkCRC() function.

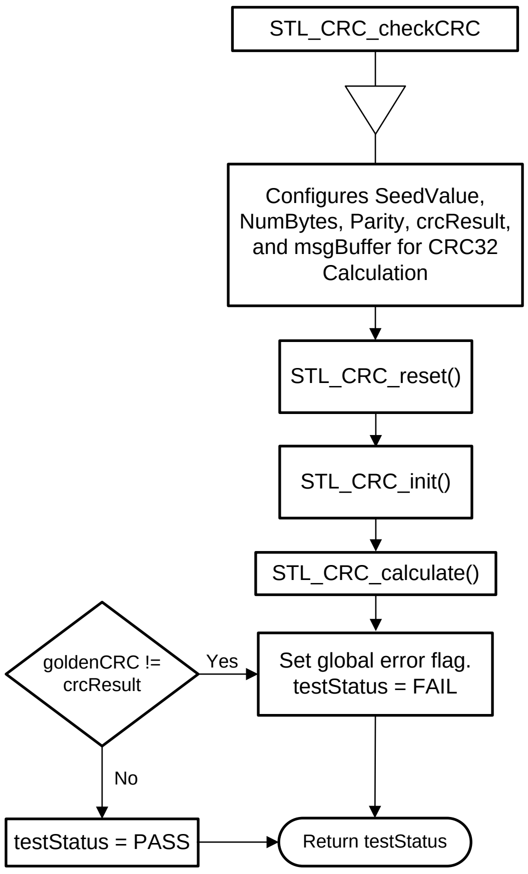
STL_CRC_checkCRC
Configures SeedValue, NumBytes, Parity, crcResult, and msgBuffer for CRC32 Calculation
STL_CRC_reset()
STL_CRC_init()
STL_CRC_calculate()
goldenCRC != crcResult
Yes
Set global error flag. testStatus = FAIL
No
testStatus = PASS
Return testStatus

Figure 7. STL_CRC_checkCRC() Flow Chart

5 References

- Texas Instruments, *Quality and Reliability: Soft Error Rate FAQs*
- Texas Instruments,*TMS320F2837xD Dual-Core Delfino Microcontrollers Technical Reference Manual*
- Texas Instruments,*TMS320F2837xD Dual-Core Delfino™ Microcontrollers Data Manual*
- Texas Instruments,*TMS320F2837xD Dual-Core Delfino™ MCUs Silicon Revisions C, B, A, 0 Silicon Errata*
- Texas Instruments,*TMS320F2837xS Delfino Microcontrollers Technical Reference Manual*
- Texas Instruments,*TMS320F2837xS Delfino™ Microcontrollers Data Manual*
- Texas Instruments,*TMS320F2837xS Delfino™ MCUs Silicon Revisions C, B Silicon Errata*
- Texas Instruments,*TMS320F2807x Piccolo Microcontrollers Technical Reference Manual*
- Texas Instruments,*TMS320F2807x Piccolo™ Microcontrollers Data Manual*
- Texas Instruments,*TMS320F2807x Piccolo™ MCUs Silicon Revisions C, B Silicon Errata*
- Texas Instruments,*C2000™ SafeTI™ Diagnostic Software Library for F2837xD, F2837xS, and F2807x Devices*