Application Report
SPRACA7–October 2017

C2000™ Hardware Built-In Self-Test

Salvatore Pezzino, Peter Ehlig

ABSTRACT

This applications note discusses the Hardware Built-In Self-Test (BIST) feature in the F28X7x family of C2000™ devices. This family includes both single-core (F2837xS and F2807x) and dual-core (F2837xD) products.

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

HWBIST refers to circuitry and scan patterns generated by an ATPG tool used to screen out logic failures within the targeted circuitry. This methodology is used extensively in semiconductor device testing. All C2000 devices make use of some level of hardware-assisted test during device manufacturing. Some of the newer C2000 devices support customer use of this test technology as part of their system test, to test the integrity of the CPUs (US 8,799,713 B2). This document describes how and why to make use of HWBIST at a system level.

Table 1 lists the terms and abbreviations used in this application report.

Table 1. Terms and Abbreviations

Abbreviation	Term
ATPG	Automatic test pattern generation
BIST	Built-In Self-Test
Capture	The embedded circuitry capturing the results of the changing logic as the seeds are clocked through the logic under test
CCS	Code Composer Studio™
Context restore	The process of restoring the central processing unit (CPU) registers and status flags after completing a hardware BIST micro-run. This is performed by the software.
Context save	The process of saving the CPU registers and status flags before starting a hardware BIST micro-run. This is performed by the software.
Core bounding	The CPU core is disconnected from peripherals and interrupt signals during a micro-run test. After the test, the core is reconnected to these signals.
Coverage	The percentage of the CPU logic that is covered by the hardware BIST.
CPU	Central processing unit
CRC	Cyclical redundancy check
F2807x	Single C28x core Piccolo™ class device
F2837xD	Dual C28x core Delfino™ class device
F2837xS	Single C28x core Delfino class device
Flash	Nonvolatile on-chip memory
FPU	Floating point unit
HWBIST	Hardware Built-In Self-Test
ISR	Interrupt service routine
JTAG	Joint test action group. JTAG is a scan-based communications protocol (like I2C) which allows for scanning to either test circuitry or emulation circuitry.
Micro-run	Execution of a portion of a full HWBIST 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 must be executed in smaller time-slices for more efficient task scheduling. During a micro-run, the CPU is isolated from all peripherals and memory. In addition, interrupts are logged by the HWBIST controller.
MISR	Multiple-input signature-register
NMI	Nonmaskable interrupt
PEST	Periodic self-test
PLL	Phase-locked loop
POR	Power-on reset
POST	Power-on self-test
RAM	Random access memory
ROM	Read-only memory
Seed	Initial states which are loaded into the circuitry using scan paths, so that the circuitry starts out in a known state before testing begins
Semaphore	A mechanism to acquire write access to certain self-test registers by CPU1 or CPU2 used on F2837xD devices
TMU	Trigonometric math unit
TRM	Technical reference manual
VCU	Viterbi and complex math unit

On the F28X7x devices, the HWBIST targets the C28x CPU and the FPU, VCU, CRC, and TMU accelerators. Also included is the emulation analysis circuitry, which manages communications between these processing elements and the emulator, as well as manages features like breakpoints, watch points, and single stepping.

The HWBIST does not target the rest of the logic on the device. The other logic on the device may be tested with other self-test or diagnostic mechanisms.

1.1 HWBIST Overview

Figure 1 shows a block diagram of HWBIST, as used in the C2000 device. The orange and pink portions show the logic targeted for testing. In system use, this logic is the processing engine for the system code. Data flows through the latches as the executing system code instructs.

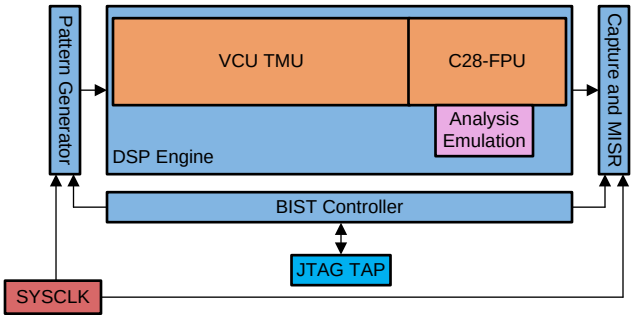

Figure 1. HWBIST Block Diagram

However, these same latches include scan access so that during tests of this logic, a high-speed test flow can validate the operation of the gates in the circuitry. In the case discussed here there are many parallel scan paths through the logic so that significant portions of the logic can be tested in parallel. While in this test mode, the logic does not operates like the processor would when running code.

The Pattern Generator provides seeds to these parallel scan paths to provide activity necessary to logically validate the operation of the targeted gates. These seeds are computer generated and the coverage is validated with standard ATPG tools. The seeds are optimized to meet a particular fault grade target in a minimum number of cycles. The vendors of these optimizers take great pride in this optimization.

NOTE: This optimization means the switching rates of the transistors is significantly higher than those occurring when this logic is executing system code. Additionally, this provides very high fault coverage.

The capture and MISR portion picks up the results of the scanning operation across all the parallel chains. The interaction of the stepping of the scan patterns through the paths interacts with other logic gates in the circuitry tied to the latches. The optimization software injects faults into the gates, and if the MISR does not recognize a failure, then additional seeds are necessary to validate the faulted gate. The optimizer is given a coverage target and will continue to generate seeds until this metric is met. Reaching coverage of 60% is relatively simple; reaching 95% takes significantly more seeds, and reaching 99% requires significantly more seeds than 95% does.

NOTE: As the bits are driven through the parallel scan paths, the contexts of all the targeted latches are changed multiple times. Stated differently, any context in these latches before testing is completely lost during the test. The context is restored through a combination of hardware logic and software.

The clocking of the scan operations is driven by SYSCLK. The BIST controller manages how the data is shifted and clocked during the scan flow. The BIST controller also manages the loading of seeds and comparison values for the MISR. In device manufacturing test flow, the BIST controller and clock source are established using a device test port like JTAG.

This is an oversimplified description of this testing methodology. A number of detailed and scholarly articles on scan-based testing are available on the web.

1.1.1 HWBIST Working In-System

As stated earlier, some C2000 devices support the use of the HWBIST to screen the CPU for logic failures in the system rather than just during device manufacture testing. Figure 2 shows a block diagram, which includes the additional circuitry to support this option.

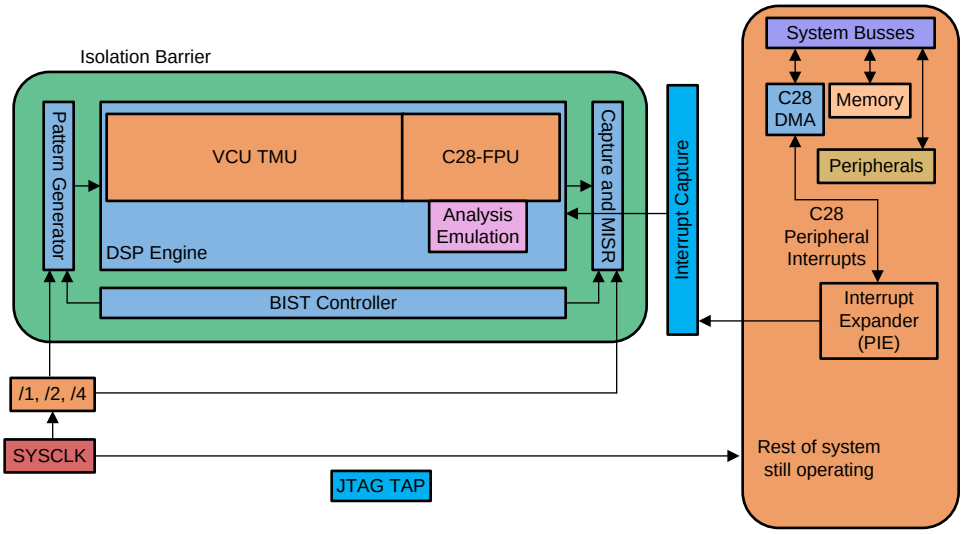

Figure 2. HWBIST In-System Block Diagram

When using HWBIST in a device test environment, testers manage each targeted logic portion to provide the overall test of the device. However, when the device is in a system, aspects of the system may be adversely affected by the activity of the under test targeted logic. These aspects are both on the device and outside the device. For this reason, the HWBIST includes a barrier around the targeted logic so that the activity of the HWBIST testing is isolated from the rest of the system. The CPU is disconnected from peripherals and interrupt signals during a micro-run test. After the test, the core is reconnected to these signals. This is known as *core bounding*. In Figure 2, the isolation barrier is shown in green. It is also true that the logic under test must be isolated from activity elsewhere in the system. This barrier provides this as well.

However, if the system must get the attention of the CPUs under test, then it can provide interrupts. These interrupts are captured in a buffer and provided to the CPU logic under test when the BIST controller releases the targeted logic. The complete coverage testing of the targeted logic takes a while. The higher the coverage goal, the longer it takes. The BIST controller in the C2000 devices executes and validates the total coverage seeds in small portions to minimize the latency to these captured interrupts. This also addresses some of the power concerns of the higher transistor switching rates generated through the parallel scan paths.

In extreme situations, the system resources can generate an NMI that halts the HWBIST operation and brings back the CPU under test using a HWBIST Reset. As soon as the context restore is complete, the NMI vector is taken and the NMI service routine can decode the NMI flag register to determine the source of the interruption. The NMI will trap before the HWBIST software returns to the calling sequence. The user application must manage the NMI responses accordingly.

A significant difference between device-manufacturing HWBIST and in-system HWBIST, is that the device tester communicates with the BIST controller over a test port, while the in-system HWBIST uses the CPU to communicate with the BIST controller. The CPU that the HWBIST is testing, is the CPU which manages the HWBIST controller. More specifically, the C28x CPU under test, where all the latches are changed multiple times, controls the BIST operation.

Here is how this process works, code running on the CPU does the following:

1. Initializes the mode of operation in the HWBIST – Maps the CPU reset to respond to the HWBIST return to service routine specially coded for return from HWBIST.
2. Turns on the Interrupt Capture Buffer.
3. Saves the context of the CPU and the associated code-based accelerators.
4. Starts up the small time-slice of the HWBIST execution – At this point the CPU stops being a CPU and starts being logic under test.
5. Upon completion of the small time-slice of the HWBIST execution the HWBIST controller:
   - Captures the results in a status register – If a logic failure is detected, the BIST controller generates a NMI to the CPU.
   - Generates CPU reset to the CPU logic:
     - This reset puts the CPU into a known and controlled state.
     - Upon release of this reset, the logic under test becomes a CPU again.
6. The CPU executes the HWBIST reset service routine:
   - Restores saved context
   - Shuts down residuals of the HWBIST controller operation
   - Releases the interrupts stored in the Interrupt Capture Buffer
   - Returns to the calling sequence with the resulting status provided by the return statement

All of the operations listed in the previous outline are executed in the C2000 SafeTI™ Diagnostic Library. The details of how the system code calls this driver are provided in Section 2 of this document.

While the HWBIST is executing, other aspects of the system can operate as well. As previously mentioned, interrupts coming from off-chip or on-chip sources are saved in the Interrupt Capture Buffer. However, triggers mapped to DMA channels are processed while the HWBIST is actively testing the CPU. For example, system-related commands from a SCI or I2C port can be collected and moved from the port to system memory to be processed as soon as Step 6 is completed. This is because all the device buses are isolated using the HWBIST Barrier.

1.2 HWBIST Failure Response

As mentioned earlier, when the C28x CPU starts the HWBIST controller, the CPU shuts down so that the logic inside can be tested by the HWBIST engine. Figure 3 shows the flow of this action in the state diagram.

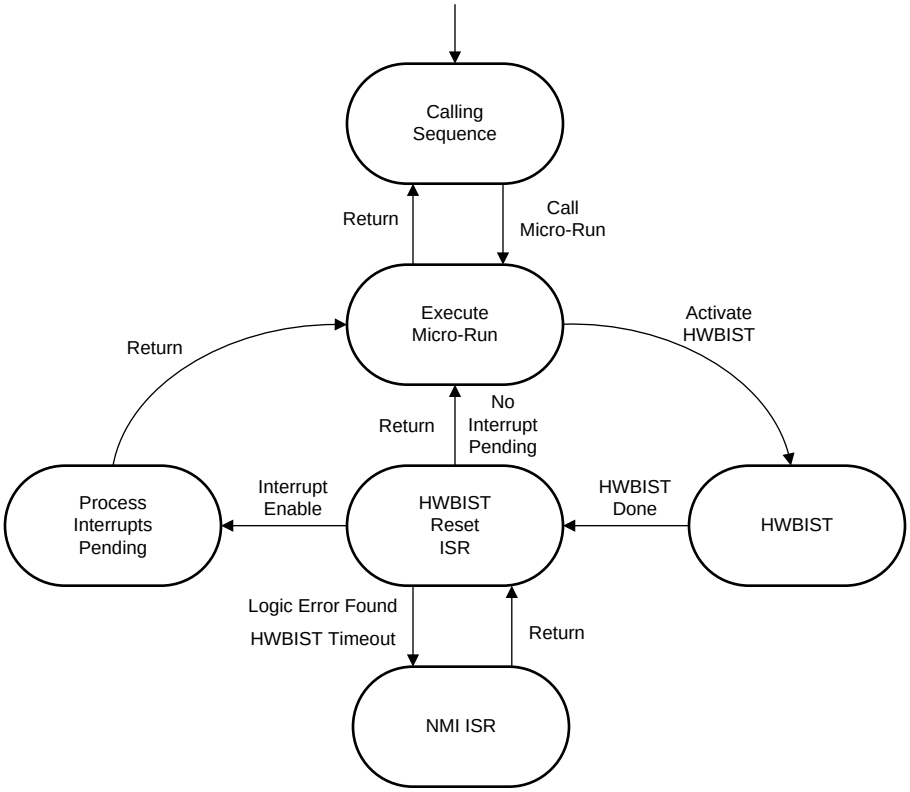

Figure 3. HWBIST State Diagram

When the HWBIST detects a failure, it sets the appropriate bit in the HWBIST Status register and exits the HWBIST operation. This error can come in the form of the following:

- Logic failure was detected
- HWBIST controller timed out without the micro-run completing

In either case, the HWBIST controller saves the failure information into the HWBIST status register, generates a NMI to the processor, and sets the appropriate bits in the NMI flag register. In a dual processor device, the HWBIST controller generates NMIs to each processor.

1.3 Advantages of Using HWBIST In-System

There are a number of reasons to use HWBIST in a system. Five legitimate examples follow:

- Validate that the C2000 device is correctly connected in the system during the initial design validation and debug of the system.

  HWBIST may be too rigorous for this aspect of prototype debug. The emulator provides simpler methods for this effort.

- Validate that the C2000 device is still functional after being attached to the board.

  As part of the system manufacture, it is useful to know that the part has not been damaged during board manufacturing. A board manufacturing event is most likely to catastrophically damage the device, in which case the HWBIST cannot be run in-system. Additionally, the damage will most likely be done to the pin driver/buffers, peripheral circuitry, or embedded memories, which are not tested by the HWBIST. It is highly unlikely that board or system manufacturing events would damage only the circuitry targeted by the HWBIST. It is uncommon for damage to occur to the device during board manufacture. However, if the device is damaged, it is good to know early so that adjustments can be made on the board manufacturing line.

- Check whether the device has been damaged after working properly in the system – Damage to the device is most likely to occur due to one of the following causes:
  - Overstress during power up
  - Overstress during power down
  - Voltage overstress due to power supply event
  - Temperature overstress

    Running the HWBIST at system start-up addresses the first two causes. System temperature and voltage monitors address the remaining two causes

- Monitor the device for manufacture test escapes.

  This is not an effective use for the HWBIST in the system, because the HWBIST has already been run in the device tester environment where it can be executed with significantly higher margin, both voltage and temperature. However, if the HWBIST does capture a failure, this is a cause for concern that something in the system is operating well outside the operating range defined in the data sheet. This may not be measureable at the pins of the device, because it may be a momentary event.

- Monitor the device for degrading mechanisms.

  Some level of transistor degradation is normal and expected with use of the circuitry. This is minor and the design and device testing includes a margin to compensate for this drift.

  Additionally, there will be some latent defects that are not screenable with normal device testing methods. These defect mechanisms require some level of stressing to accelerate failures. Stress testing is used in the device manufacturing test to accelerate the majority of these degrading defect mechanisms.

  Lastly, the HWBIST helps identify these degrading mechanisms that escape the aggressive device manufacture testing.

2 Using HWBIST In-System

This section describes how to use the HWBIST in a system and is tightly coupled with the software releases in the C2000 SafeTI Diagnostic Library for F28X7x devices. The description starts with a simple explanation of running the HWBIST on a single core. The section then details the additional code necessary to run on a dual-core device. This section finishes with a discussion of debugging suggestions.

NOTE: The software for the configuration and execution of the HWBIST, released in the C2000 SafeTI Diagnostic Library, must not be modified by the user. If the initialization and execution of the HWBIST is modified, the documented coverage is not guaranteed.

2.1 Fundamental HWBIST on Single-Core Device

Executing the HWBIST involves the following four code segments:

- Initialize the HWBIST controller
- Execute the HWBIST
- Recover from the HWBIST
- Manage results

Most of this is accomplished by Diagnostic Library functions which can be called. The function definitions are provided in the header file, stl_hwbist.h. More details on these function descriptions is in stl_hwbist.h or Diag_Lib_TMS320F2837x_07x_Users_Guide, which is in the /docs folder of the library release package.

There are eight functions included, as follows:

```
__interrupt void STL_HWBIST_errorNMIISR(void);
uint16_t STL_HWBIST_runFull(const STL_HWBIST_Error errorType);
uint16_t STL_HWBIST_runMicro(void);
void STL_HWBIST_restoreContext(void);
void STL_HWBIST_init(const STL_HWBIST_Coverage coverage);
void STL_HWBIST_injectError(const STL_HWBIST_Error errorType);
bool STL_HWBIST_claimSemaphore(const STL_HWBIST_Core core);
void STL_HWBIST_releaseSemaphore(void);
```

Use of these routines is described in the following subsections.

2.1.1 Initializing the HWBIST Controller

Initializing the HWBIST controller is accomplished by calling this library function:

```
void STL_HWBIST_init(const STL_HWBIST_Coverage coverage);
```

This function initializes the HWBIST controller for operation. The *coverage* parameter is an enumerated type STL_HWBIST_Coverage and specifies the coverage to achieve. This function expects the HWBIST semaphore to be claimed by the CPU trying to execute a run on the HWBIST. This function initializes the HWBIST registers for the following configuration:

- 46-cycle micro-runs for the first 95% coverage and 61-cycle micro-runs for the incremental coverage to get to 99%:
  - Minimizes the time-slice of the micro-run
  - Minimizes the context latency
  - Minimizes the power consumption during the micro-run
- HWBIST clock equal to system clock
- Return address is 0x0000, which is the beginning of the RAMM0 block of memory.
- 95% or 99% coverage – This is specified by the input parameter *coverage*.

2.1.2 Code Composer Studio™ Project Configuration

The HWBIST executes across both the VCU and FPU, and the Diagnostic Library HWBIST code executes a full context save and restore for these components. Therefore, for the project to correctly compile and link, it is necessary to specify both fpu32 and VCU2 support in the Processor Options window of the Project Properties. To get to this, type ALT-Enter or right-click on the project in the Project Explorer and clock Properties to open the Project Properties window, then click on Processor Options to bring up the Specify options.

In addition, the following line must be added at the beginning of the linker command file:

```
-u _STL_HWBIST_restoreContext
```

This line allows the HWBIST recovering code to map the return vector (0x0000).

Also, a HWBIST linker section must be defined, as follows, in the linker command file:

```
Linker Command File MEMORY:
HWBIST           : origin = 0x000000, length = 0x000020
```

The hwbist memory section must be mapped to the HWBIST linker section. See the following for an example where hwbist is initially loaded to flash. Linker command symbols are created to facilitate a memory copy to the HWBIST linker section at 0x0000.

```
Linker Command File SECTIONS:
   /* Must be placed at 0x0000 */
   hwbist            : LOAD = FLASHJ,
                       RUN = HWBIST,
                       LOAD_START(_HwbistLoadStart),
                       LOAD_SIZE(_HwbistLoadSize),
                       LOAD_END(_HwbistLoadEnd),
                       RUN_START(_HwbistRunStart),
                       RUN_SIZE(_HwbistRunSize),
                       RUN_END(_HwbistRunEnd),
                       PAGE = 0, ALIGN(4)
```

Furthermore, the hwbiststack memory section must be allocated to RAM and contained within a 16-bit memory address. This hwbiststack section is used for a context save and restore for each micro-run.

Lastly, source code must be added to the main application, to perform the necessary memory copy of hwbist to the return vector 0x0000, defined by the HWBIST linker section.

Source Code File Externs:

```
extern uint16_t HwbistLoadStart;
extern uint16_t HwbistLoadSize;
extern uint16_t HwbistRunStart;
```

Source Code File Code:

```
memcpy(&HwbistRunStart, &HwbistLoadStart, (size_t)&HwbistLoadSize);
```

For a more complete example, see the self-test application (STA) in the Diagnostic Library software release package.

2.1.3 Executing HWBIST

The HWBIST executes a number of micro-run operations until the full coverage is met. The Diagnostic library provides two options for completing HWBIST. The first option is to initialize the HWBIST controller once per full HWBIST, using STL_HWBIST_init(), and then execute STL_HWBIST_runMicro() periodically until the HWBIST completes. This options allows for smaller time-slicing of the HWBIST. The second option is to call STL_HWBIST_runMicro(), which completes a full HWBIST run, and then returns to the user's code. This option takes a longer amount of time and is more useful for a power-on self-test or whenever more time may be allocated to perform a full HWBIST.

2.1.3.1 Executing HWBIST Micro-Run

To execute one micro-run of the HWBIST after the semaphore has been claimed by the CPU core under test and a 1-time initialization has been performed, the user must call the following function:

```
STL_HWBIST_runMicro();
```

This function performs a HWBIST micro-run of the CPU under test.

This function expects the HWBIST semaphore to be claimed by the CPU trying to run a full HWBIST, by calling this function multiple times in-system. This function also expects the HWBIST engine to be initialized with STL_HWBIST_init(). This function performs a HWBIST micro-run and returns the status of the micro-run. Before returning, the function restores the previous NMI vector.

This function performs a HWBIST micro-run and is designed to be used as a periodic self-test (PEST).

If the HWBIST is being used on a dual-core device, then for the HWBIST to run and test the CPU in use, the HWBIST semaphore must be claimed by that CPU trying to use the HWBIST engine. It uses STL_HWBIST_claimSemaphore() until it returns true, and then releases the semaphore using STL_HWBIST_releaseSemaphore() so the other CPU can claim the HWBIST semaphore.

If the HWBIST micro-run test passes with no errors, then this function returns the status, the value is either STL_HWBIST_MACRO_DONE or a bitwise OR of the values STL_HWBIST_BIST_DONE, and STL_HWBIST_MACRO_DONE. If the test fails, then the status of the HWBIST and the return value of the function is a bitwise OR of some combination of the following values: STL_HWBIST_NMI, STL_HWBIST_BIST_FAIL, STL_HWBIST_INT_COMP_FAIL, and STL_HWBIST_TO_FAIL. These macros are defined in stl_hwbist.h.

Table 2 lists the meaning of each macro or bit of the return value.

Table 2. STL_HWBIST_runMicro() Return Values

STL_HWBIST Macro	Meaning
STL_HWBIST_DONE	The full HWBIT operation is complete. This could mean the HWBIST has completed the necessary micro-runs to meet the coverage metric, or that the HWBIST has detected an error.
STL_HWBIST_MACRO_DONE	The micro-run has completed.
STL_HWBIST_NMI	An NMI was generated by the HWBIST controller. This could be due to the following: • External NMI • Time-out failure in the controller • Logic error was detected by the HWBIST operation.
STL_HWBIST_BIST_FAIL	The HWBIST detected an error. This could be due to the following: • Time-out failure in the controller • Logic error was detected by the HWBIST operation.
STL_HWBIST_INT_COMP_FAIL	The HWBIST detected a logic failure.
STL_HWBIST_TO_FAIL	The HWBIST controller detected a time-out failure.

Figure 4 shows is a flow chart detailing the design of the STL_HWBIST_runMicro() function. This information is also available in the *Diagnostic Library User's Guide*.

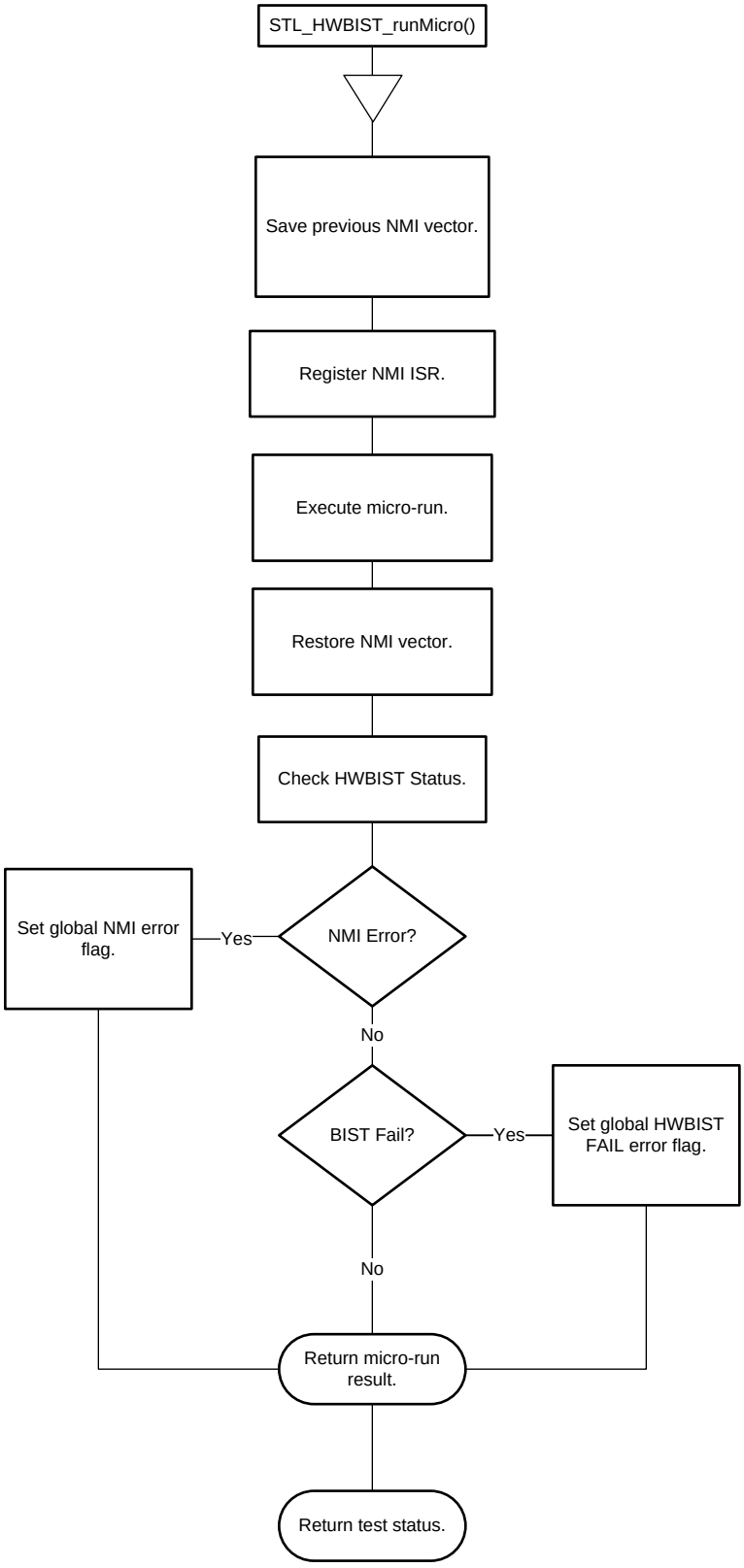

Figure 4. STL_HWBIST_runMicro() Flow Chart

To perform a full HWBIST with 95% coverage using the STL_HWBIST_runMicro() function, the following sequence of functions must be executed:

1. Claim the HWBIST semaphore.
   ```
   STL_HWBIST_claimSemaphore();
   ```
2. Initialize the HWBIST for 95% coverage.
   ```
   STL_HWBIST_init(STL_HWBIST_95_LOS);
   ```
3. Execute a HWBIST micro-run.
   ```
   STL_HWBIST_runMicro();
   ```
4. Repeat Step 3 until complete or no error is observed – Execute STL_HWBIT_runMicro() 1700 times until it is complete with no error, or until an error is observed through the return value, a global error flag is set, or an NMI is triggered.
5. Release the HWBIST semaphore.
   ```
   STL_HWBIST_releaseSemaphore();
   ```

To perform a full HWBIST with 99% coverage using the STL_HWBIST_runMicro() function, the following sequence of functions must be executed:

1. Claim the HWBIST semaphore.
   ```
   STL_HWBIST_claimSemaphore();
   ```
2. Initialize the HWBIST for 95% coverage.
   ```
   STL_HWBIST_init(STL_HWBIST_95_LOS);
   ```
3. Execute a HWBIST micro-run.
   ```
   STL_HWBIST_runMicro();
   ```
4. Repeat Step 3 until complete or no error is observed – Execute STL_HWBIT_runMicro() 1700 times until it is complete with no error, or until an error is observed through the return value, a global error flag is set, or an NMI is triggered.
5. Initialize the HWBIST for 99% coverage.
   ```
   STL_HWBIST_init(STL_HWBIST_99_LOS);
   ```
6. Execute a HWBIST micro-run.
   ```
   STL_HWBIST_runMicro();
   ```
7. Repeat Step 6 until complete or no error is observed – Execute STL_HWBIT_runMicro() 300 times until it is complete with no error, or until an error is observed through the return value, a global error flag is set, or an NMI is triggered.
8. Release the HWBIST semaphore.
   ```
   STL_HWBIST_releaseSemaphore();
   ```

As previously detailed, to perform a HWBIST with 99% coverage, the HWBIST controller must first complete a HWBIST full run for 95% coverage, and then be reinitialized for 99% and executed again until it completes. It takes 1700 micro-runs to achieve 95% coverage and an additional 300 micro-runs to achieve the additional 4% for a total of 99% coverage.

Figure 5 shows a flow chart of the setup and execution of a single time-sliced micro-run.

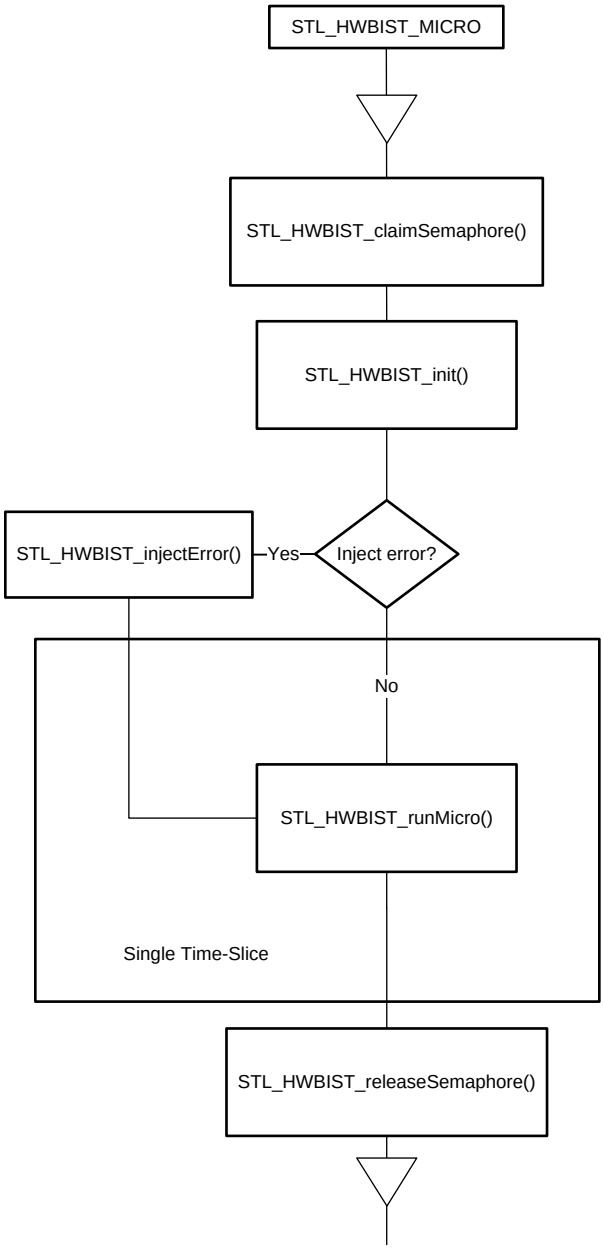

Figure 5. Flow Chart of Time-Sliced Micro-Run Execution

2.1.3.2 Executing HWBIST Full-Run

To execute a full run of the HWBIST after the semaphore has been claimed by the CPU core under test, call the following function:

```
STL_HWBIST_runFull();
```

This function performs a full HWBIST of the CPU under test.

The errorType parameter is an enumerated type STL_HWBIST_Error, which specifies the type of error to inject before executing a full run of HWBIST test. This function expects the CPU trying to run a full HWBIST to claim the HWBIST semaphore by calling it. This function initializes the HWBIST engine and then injects the errorType. It also registers the STL_HWBIST_NMIISR as the NMI vector. The function then performs a full HWBIST, achieving first the 95% launch-on-shift coverage and then achieving 99% coverage by testing for stuck-at-faults. If there is a failure in the HWBIST, then a global error flag is set and the return value specifies a failure. Additionally, if the coverage is not achieved in the expected micro-runs then the test fails due to an overrun. Before returning, the function restores the previous NMI vector.

If the HWBIST is being used on a dual-core device, then for the HWBIST to run and test the CPU in use, the HWBIST semaphore must be claimed by that CPU trying to use the HWBIST engine, using STL_HWBIST_claimSemaphore(), until it returns true and then release the semaphore using STL_HWBIST_releaseSemaphore(), for the other CPU to be able to claim the HWBIST semaphore.

If the HWBIST full run test passes with no errors within the expected number of micro-runs, then this function returns the status of the HWBIST and the value will be a bitwise OR of the values STL_HWBIST_BIST_DONE, and STL_HWBIST_MACRO_DONE. If the test fails, then the status of the HWBIST and the return value of the function is a bitwise OR of some combination of the following values: STL_HWBIST_NMI, STL_HWBIST_BIST_FAIL, STL_HWBIST_INT_COMP_FAIL, STL_HWBIST_TO_FAIL, and STL_HWBIST_OVERRUN_FAIL.

Table 3 lists the meaning of each macro or bit of the return value.

Table 3. STL_HWBIST_runFull() Return Values

STL_HWBIST Macro	Meaning
STL_HWBIST_DONE	The full HWBIT operation is complete. This could mean the HWBIST has completed the necessary micro-runs to meet the coverage metric, or it could mean that HWBIST has detected an error.
STL_HWBIST_MACRO_DONE	The micro-run is complete.
STL_HWBIST_NMI	An NMI was generated by the HWBIST controller. This could be due to either: • External NMI • Time-out failure in the controller • Logic error detected by the HWBIST operation
STL_HWBIST_BIST_FAIL	The HWBIST detected an error. This could be due to either: • Time-out failure in the controller • Logic error was detected by the HWBIST operation.
STL_HWBIST_INT_COMP_FAIL	The HWBIST detected a logic failure.
STL_HWBIST_TO_FAIL	The HWBIST controller detected a time-out failure.
STL_HWBIST_OVERRUN_FAIL	The HWBIST controller did not complete a full HWBIST in the expected number of micro-runs.

Figure 6 shows a flow chart detailing the design of the STL_HWBIST_runFull() function. This information is also available in the *Diagnostic Library User's Guide*.

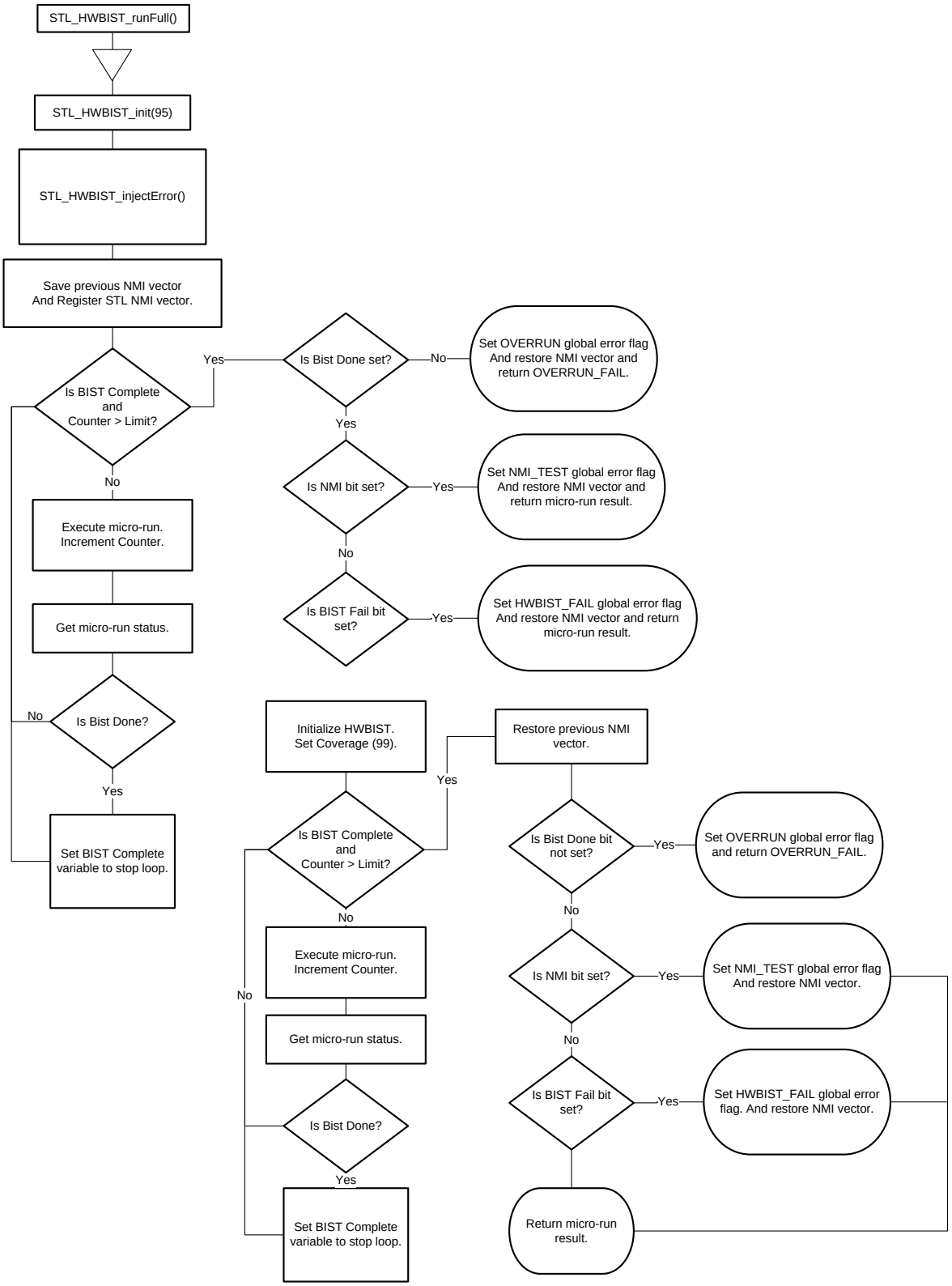

Figure 6. STL_HWBIST_runFull() Flow Chart

To perform a full HWBIST with 99% coverage using the STL_HWBIST_runFull() function, execute the following sequence of functions:

1. Claim the HWBIST semaphore.
   ```
   STL_HWBIST_claimSemaphore();
   ```
2. Execute a HWBIST full run.
   ```
   STL_HWBIST_runFull();
   ```
3. Release the HWBIST semaphore.
   ```
   STL_HWBIST_releaseSemaphore();
   ```

This sequence executes a full HWBIST in a single time-slice. Figure 7 shows a full HWBIST run in a single time-slice.

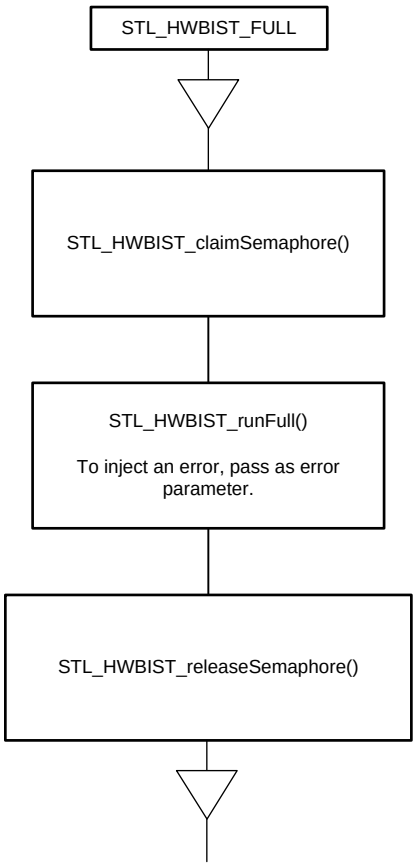

Figure 7. Full HWBIST in Single Time-Slice

2.1.4 Error Management

A failing condition from a HWBIST execution is a serious situation. If this occurs, then the behavior of the CPU that failed cannot be ensured. Code that takes appropriate action to gracefully shut down the system must be included. This can be done by decoding the return value of STL_HWBIST_runFull() or STL_HWBIST_runMicro().

The code can be managed through a trap to the NMI. This is the quicker method for managing a HWBIST failure, especially in the case of a dual CPU device, because the NMI is sent to both CPUs. To take advantage of the NMI traps, the system code must do the following:

1. Clear any NMI trap residual from the NMI flag register:
   ```
   SysCtl_clearNMIStatus(STL_HWBIST_NMI_CPU1_HWBISTERR);
   SysCtl_clearNMIStatus(STL_HWBIST_NMI_CPU2_HWBISTERR);
   ```

2. Map the NMI vector to the Interrupt Service Routine that handles HWBIST:

```
Interrupt_register(INT_NMI, STL_HWBIST_errorNMIISR);
```

3. Enable the PIE controller:

```
HWREGH(PIECTRL_BASE + PIE_O_CTRL) |= PIE_CTRL_ENPIE;
```

   This bit is set on the CPU being tested by the HWBIST as part of the HWBIST execution while loop, but it must also be set on the other CPU (in a dual-CPU device) for it to respond to the failure.

The following code shows how the NMI results may be decoded. Code must be added to manage the graceful shut-down of the system. This code may be different depending on whether it is run on CPU1 or CPU2 and which CPU fails.

NMI Interrupt Service Routine to Handle HWBIST Errors:

```
//*****************************************************************************
//
// STL_HWBIST_NMIISR(void)
//
//*****************************************************************************
__interrupt void STL_HWBIST_errorNMIISR(void)
{
    //
    // Check for HWBIST error.
    //
    if((SysCtl_getNMIFlagStatus() & STL_HWBIST_NMI_CPU1_HWBISTERR) ==
                                            STL_HWBIST_NMI_CPU1_HWBISTERR)
    {
        //
        // Report global error.
        //
        STL_Util_setErrorFlag(STL_UTIL_HWBIST_NMI_INT);

        //
        // Clear the NMI CPU1 HWBIST Error flag and NMIINT flag if it is
        // the only flag left.
        //
        SysCtl_clearNMIStatus(STL_HWBIST_NMI_CPU1_HWBISTERR);

        //
        //** To Do:
        //** Add code to manage HWBIST fault and gracefully shut down system.
        //
    }
    if((SysCtl_getNMIFlagStatus() & STL_HWBIST_NMI_CPU2_HWBISTERR) ==
                                            STL_HWBIST_NMI_CPU2_HWBISTERR)
    {
        //
        // Report global error.
        //
        STL_Util_setErrorFlag(STL_UTIL_HWBIST_NMI_INT);

        //
        // Clear the NMI CPU2 HWBIST Error flag and NMIINT flag if it is
        // the only flag left.
        //
        SysCtl_clearNMIStatus(STL_HWBIST_NMI_CPU2_HWBISTERR);

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

Figure 8 shows the NMIFLG register with the CPU1HWBISTERR and CPU2HWBISTERR bits from the TMS320F2837xD Dual-Core Delfino Technical Reference Manual for F2837xD devices. Corresponding bits exist in the TRMs for F2837xS and F2807x devices.

Figure 8. NMIFLG Register

15	14	13	12	11	10	9	8
RESERVED				RESERVED	CPU2NMIWDRSn	CPU2WDRSn	RESERVED
R-0h				R-0h	R-0h	R-0h	R-0h
7	6	5	4	3	2	1	0
RESERVED	PIEVECTERR	CPU2HWBISTERR	CPU1HWBISTERR	FLUNCERR	RAMUNCERR	CLOCKFAIL	NMIINT
R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h	R-0h

Table 4. NMIFLG Register Field Descriptions

Bit	Field	Type	Reset	Description
15-11	RESERVED	R	0h	Reserved
10	CPU2NMIWDRSn	R	0h	CPU2 NMIWDRSn Reset Indication Flag(1). This bits indicates if NMIWDRSn of CPU2 was fired or not: • 0 = CPU2.NMIWDRsn was not fired. • 1 = CPU2.NMIWDRSn was fired to CPU2. Reset type: XRSn
9	CPU2WDRSn	R	0h	CPU2 WDRSn Reset Indication Flag(1). This bits indicates if WDRSn of CPU2 was fired or not: • 0 = CPU2.WDRsn was not fired. • 1 = CPU2.WDRSn was fired to CPU2. Reset type: XRSn
8-7	RESERVED	R	0h	Reserved
6	PIEVECTERR	R	0h	PIE Vector Fetch Error Flag. This bit indicates if an error occurred on an Vector Fect by the other CPU in the device. For example, CPU1.NMIWD gets an NMI on an Vector fetch Error on CPU2. This bit can only be cleared by the user writing to the corresponding clear bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No Vector Fetch Error condition (on the other CPU) pending • 1 = Vector Fetch error condition (on the other CPU) generated Reset type: XRSn
5	CPU2HWBISTERR	R	0h	HWBIST Error NMI Flag. This bit indicates if the time-out error or a signature mismatch error condition during hardware BIST of C28 CPU2 occurred. This bit can only be cleared by the user writing to the corresponding clear bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No C28 HWBIST error condition pending • 1 = C28 BIST error condition generated Reset type: XRSn
4	CPU1HWBISTERR	R	0h	HWBIST Error NMI Flag: This bit indicates if the time-out error or a signature mismatch error condition during hardware BIST of C28 CPU1 occurred. This bit can only be cleared by the user writing to the corresponding clear bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No C28 HWBIST error condition pending • 1 = C28 BIST error condition generated Reset type: XRSn

(1) This bit is reserved for the CPU2.NMIFLG register.

Table 4. NMIFLG Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
3	FLUNCERR	R	0h	Flash Uncorrectable Error NMI Flag: This bit indicates if an uncorrectable error occurred on a C28 flash access and that condition is latched. This bit can only be cleared by the user writing to the corresponding clear bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No C28 flash uncorrectable error condition pending • 1 = C28 flash uncorrectable error condition generated Reset type: XRSn
2	RAMUNCERR	R	0h	RAM Uncorrectable Error NMI Flag: This bit indicates if an uncorrectable error occurred on a RAM access (by any master) and that condition is latched. This bit can only be cleared by the user writing to the corresponding clear bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No RAM uncorrectable error condition pending • 1 = RAM uncorrectable error condition generated Reset type: XRSn
1	CLOCKFAIL	R	0h	Clock Fail Interrupt Flag: These bits indicates if the CLOCKFAIL condition is latched. These bits can only be cleared by the user writing to the respective bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No CLOCKFAIL condition pending • 1 = CLOCKFAIL condition generated Reset type: XRSn
0	NMIINT	R	0h	NMI Interrupt Flag: This bit indicates if an NMI interrupt was generated. This bit can only be cleared by the user writing to the respective bit in the NMIFLGCLR register or by an XRSn reset: • 0 = No NMI Interrupt generated • 1 = NMI Interrupt generated No further NMI interrupts pulses are generated until this flag is cleared by the user. Reset type: XRSn

2.2 Managing HWBIST on Dual-Core Device

F2837xD dual-core devices support HWBIST testing on each core. Only one core at a time can run HWBIST. To manage this HWBIST controller, certain semaphore registers allow one processor to own the HWBIST controller until it is complete. Upon completion of a full HWBIST, the tested processor should release the semaphore control to the other processor so that it can run the HWBIST.

2.2.1 Semaphore Management

Semaphore management on F2837xD devices is handled using the following function calls, which access the CSTCSEM register:

```
STL_HWBIST_claimSemaphore(const STL_HWBIST_Core core);
STL_HWBIST_releaseSemaphore(void);
```

- After system reset, the HWBIST semaphore = 0.
  - CPU1 can access the HWBIST resources and change the semaphore.
  - CPU2 can change the semaphore.
- When CPU1 decides to execute HWBIST, it sets the semaphore = 2.
  - Grants CPU1 access of the HWBIST resources
  - Blocks CPU2 access of the HWBIST resources and blocks a change of the semaphore
- When CPU1 completes the HWBIST test, it sets the semaphore = 3, which grants access to the semaphore by either CPU.
- When CPU2 decides to execute HWBIST, it sets the semaphore = 1.
  - Grants CPU2 access of the HWBIST resources.
  - Blocks CPU1 access to the HWBIST resources and blocks a change of the semaphore.
- When CPU2 completes the HWBIST test, it sets the semaphore = 3, which grants access to the semaphore by either CPU.

2.2.2 Interprocessor Communications

The Interprocessor Communications (IPC) peripheral can easily manage the level of communications needed to keep each processor (CPU1 and CPU2) informed if the other intends to run the HWBIST. For example, if a critical system interrupt must be monitored and mapped to CPU1, it may be advantageous to map this interrupt to CPU2 while CPU1 executes HWBIST operations. IPC messages and interrupts can be employed to achieve this interprocessor communication between CPU1 and CPU2. IPC can also be used to implement some handshaking between the two processors when handling the semaphore management.

2.3 System Considerations When Using HWBIST

In summary, while the HWBIST micro-run executes, the targeted CPU is, for all practical purposes, gone from the system.

2.3.1 Interrupt Latency

With a 200-MHz system clock, a minimum-sized micro-run takes approximately 2.5 µs. The 2.5 µs value implies running from 0 wait state SRAM and takes longer with the wait-states of the flash memory. This brings up the following points to consider:

- *Are there any system-critical interrupts that cannot wait out the 2.5 µs?* Think in terms of interrupts that want to shut down the control operation due to an identified system fault. If there are, then these interrupts must be rerouted to the other processor, or a DMA channel for emergency processing while the HWBIST micro-run owns the CPU circuitry. Additionally, the system critical interrupt or task may be mapped to the NMI which would stop the HWBIST micro-run execution.
- *Is the control loop within 2.5 µs of needing an update from the feedback?* If so, do not start the HWBIST until after this update is completed and you have adequate time before the next update, or manage the update with a DMA channel.

- *Is there a time slot in the control loop where 2.5 µs is available for executing the HWBIST micro-run?* If yes, then this may be a suitable time slot to perform a HWBIST micro-run.

2.3.2 Power Considerations

A minimum-sized micro-run takes an additional 40 to 70 mA more power than a code running on the CPU. Users should consider the following:

- Immediately after a Power-On Reset (POR) and the completion of the boot ROM start-up code, most of the peripherals are not yet running. Therefore, running HWBIST before the system begins executing control loops will allow for a more-than-adequate power margin to handle the extra current.
- If the HWBIST is executed while a significant amount of the device circuitry is active and at a high temperature, then either:
  - Include some extra power margining in the system design
  - Execute micro-runs with /2 or /4 clocking

This second option increases the execution and interrupt latency time to 2x or 4x, respectively. Additionally, this option would require a source code change and additional testing of the HWBIST functions in the Diagnostic Library. To divide the clock by 2, a value of 1 must be written to bits 18-19 of CSTCGCR7. To divide the clock by 4, a value of 2 must be written to bits 18-19 of CSTCGCR7. This source code modification should be made in the appropriate location of the STL_HWBIST_init() function.

2.3.3 HWBIST Memory Requirements

Three ranges of memory are reserved for HWBIST operation, as follows:

- 32 words starting at CPU address 0x0000 – This is the only specific memory address requirement and is used by the *hwbist* memory section.
- 80 words of the stack range for a full context save:
  - Must be contained within a 16-bit memory address
  - Used to place the *hwbiststack* memory section
- Approximately 150 words for the HWBIST utility code

2.3.4 Injecting Errors

The HWBIST includes some error injection features to help validate the system error handling code. These errors can be invoked by running the following function:

```
void STL_HWBIST_injectError(const STL_HWBIST_Error errorType);
```

Table 5 lists the injected error types, values, and expected behaviors.

Table 5. Injecting Errors, Values, and Behaviors

STL_HWBIST_ERROR Type	Value	Description and Behavior
STL_HWBIST_NO_ERROR	0x00000000	Clears the error injection feature for future operation The HWBIST passes under normal operation, if no faults present.
STL_HWBIST_TIMEOUT	0x0000000A	Invokes a time-out error This is related to the timer in the HWBIST controller. If the HWBIST controller times out during a micro-run, then the micro-run has lost control. This generates a time-out failure flag and an NMI to the CPU or to both CPUs in a dual-core device.
STL_HWBIST_FINAL_COMPARE	0x000000A0	Corrupts the MISR compare The HWBIST executes as normal, but compares to a corrupted MISR upon completion. This does not cause a failing condition, but it does allow the CPU to check the MISR for circuit issues. No NMI is generated, and no fail status is generated.

Table 5. Injecting Errors, Values, and Behaviors (continued)

STL_HWBIST_ERROR Type	Value	Description and Behavior
STL_HWBIST_NMI_TRAP	0x00000A00	Forces an NMI to the HWBIST controller to invoke a shut down and returns control to the CPU The micro-run is stopped before beginning the HWBIST micro-run execution and an NMI is generated to the CPU under test. **NOTE:** Although an NMI is triggered, no NMI flags are set.
STL_HWBIST_LOGIC_FAULT	0x00002000	Injects a logic error into the circuitry under test to see if it is caught by the HWBIST This results in the appropriate HWBIST fail status bits being set, and generates an NMI to the CPU or to both processors if a dual CPU device. **NOTE:** Valid logic error injection values are from 0x00001000 to 0xFFFFF000. The Diagnostic Library only supplies one value (0x00002000). However, the user may wish to modify the source code to allow for writing other or multiple logic error injection values to the CSTCTEST register. See the source code in stl_hwbist.h of the Diagnostic Library.

If the code is lost during debugging error management, the most likely cause is that the NMI execution is not appropriately initialized. Additionally, see Section 2.1.4.

Table 6 lists the expected results when injecting errors into a HWBIST full run, for example, STL_HWBIST_runFull(const STL_HWBIST_Error errorType).

Table 6. Injected Errors Expected Results For HWBIST Full Run

CPU1					CPU2				
ErrorType	**Error Value**	**Return Status**	**NMI Trap**	**NMIFLG**	**ErrorType**	**Error Value**	**Return Status**	**NMI Trap**	**NMIFLG**
NO_ERROR	0x0000	0x0003	No	0x0000	NO_ERROR	0x0000	0x0003	No	0x0000
TIMEOUT	0x000A	0x0029	Yes	0x0011	NO_ERROR	0x0000	0x0003	Yes	0x0011
FINAL_COMPARE	0x00A0	0x0003	No	0x0000	NO_ERROR	0x0000	0x0003	No	0x0000
NMI_TRAP	0x0A00	0x0005	Yes	0x0000*	NO_ERROR	0x0000	0x0003	No	0x0000*
LOGIC_FAULT	0x2000	0x001B	Yes	0x0011	NO_ERROR	0x0000	0x0003	Yes	0x0011
NO_ERROR	0x0000	0x0003	No	0x0000	NO_ERROR	0x0000	0x0003	No	0x0000
NO_ERROR	0x0000	0x0003	Yes	0x0021	TIMEOUT	0x000A	0x0029	Yes	0x0021
NO_ERROR	0x0000	0x0003	No	0x0000	FINAL_COMPARE	0x00A0	0x0003	No	0x0000
NO_ERROR	0x0000	0x0003	Yes	0x0000*	NMI_TRAP	0x0A00	0x0005	Yes	0x0000*
NO_ERROR	0x0000	0x0003	Yes	0x0021	LOGIC_FAULT	0x2000	0x001B	Yes	0x0021

2.4 Debugging HWBIST In-System

While the HWBIST micro-run is executing, the emulation connection to the targeted CPU is, for all practical purposes, gone from the system. This means features like breakpoint, Watch-point, Single Step, or even run are not available for debugging the system code. This comes from the following two aspects of the HWBIST operation:

- Like the CPU, the emulation analysis circuitry is being scanned so the breakpoint (and other emulation analysis features) is managed by latches that are actively being corrupted by the HWBIST controller.
- The context of the analysis circuitry cannot be saved and restored.

TI recommends that users do not leave software breakpoints enabled in the code while executing the HWBIST. For this reason, it is necessary to disable the HWBIST operations while debugging the system code. While validating or debugging the HWBIST operations, the CPU code execution must be started using the Free Run operation. If running a dual-core device, then both CPUs must be started using the Free Run operation.

Some helping hints for debugging the HWBIST code follow:

- Use observable points on the board to monitor progress during execution – For example, use one or more GPIO pins tied to a scope or tied to LEDs.
- Store debug and progression updates or statuses in the SRAM:
  - Ideally, this is in a range of memory that is not initialized by either the BootROM or the Emulator GEL script.
  - Good to store these updates in a memory that both CPUs can access:
    - For example, shared memory, IPC registers, or message RAM.
    - The CPU that is not running the HWBIST may be able to halt cleanly and display the debug and progression information.
- Sometimes after running the HWBIST the emulator Halt operation invokes the following pop-up message:

  Trouble Halting Target CPU: (Error -1156 @ 0x0). Device may be operating in low-power mode. Do you want to bring it out of this mode? (Emulation package 5.1.636.0).

  This message is normal and caused by the emulator losing sync with the processor. The emulator always loses sync, but sometimes regains sync without the aid of this operation. This has nothing to do with low-power mode.
  - If this happens, click the Yes button.
  - If this does not work, then the CPU can be disconnected and reconnected to regain control – It is possible that the other CPU is accessible, therefore if debug and progression values have been saved, then the other CPU can provide access to them.
- If the CPU comes back, but it vectors into the BootROM or flash, the most likely reason is that the PIE is not enabled – HWBIST executes a CPU reset upon completion, but if PIE is not enabled, then the CPU vectors to the BootROM instead of the Diagnostic Library STL_HWBIST_restoreContext() code.

3 References

- Texas Instruments, TMS320F2837xD Dual-Core Delfino™ Microcontrollers, technical reference manual
- Texas Instruments, TMS320F2837xD Dual-Core Delfino™ Microcontrollers, data sheet
- Texas Instruments, TMS320F2837xD Dual-Core Delfino™ Microcontrollers, silicon errata
- Texas Instruments, TMS320F2837xS Delfino™ Microcontrollers, technical reference manual
- Texas Instruments, TMS320F2837xS Delfino™ Microcontrollers, data sheet
- Texas Instruments, TMS320F2837xS Delfino™ Microcontrollers, silicon errata
- Texas Instruments, TMS320F2807x Piccolo™ Microcontrollers, technical reference manual
- Texas Instruments, TMS320F2807x Piccolo™ Microcontrollers, data sheet
- Texas Instruments, TMS320F2807x Piccolo™ Microcontrollers, silicon errata
- Texas Instruments, C2000™ SafeTI™ Diagnostic Software Library for F2837xD, F2837xS, and F2807x Devices, C2000 diagnostic software package

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