Application Note

C2000™ Hardware Built-In Self-Test

Salvatore Pezzino, Peter Ehlig and Whitney Dewey

ABSTRACT

This application note discusses the Hardware Built-In Self-Test (HWBIST) feature in C2000™ real-time controllers. The HWBIST provides a method of reaching a high level of diagnostic coverage on the C28x CPU, which is often needed to satisfy safety standards.

Table of Contents

List of Figures

List of Tables

Trademarks

C2000™ and Code Composer Studio™ are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

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

Table 1-1. 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
C28x	The name of the core 32-bit CPU in TMS320C28x devices
CS	Code Composer Studio™
CLA	Control Law Accelerator
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
DC	Diagnostic coverage
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 multi-core devices
SDL	Software Diagnostic Library
STL	Self-Test Library
TMU	Trigonometric math unit

Table 1-1. Terms and Abbreviations (continued)

Abbreviation	Term
TRM	Technical reference manual
VCU	Viterbi and complex math unit

On C2000 devices that have HWBIST, 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 as described in the device-specific safety manual.

For the availability of HWBIST, see the device-specific data sheet. You can also check the documents below for HWBIST and other functional safety features and collateral:

- Automotive Functional Safety for C2000™ Real-Time Microcontrollers
- Industrial Functional Safety for C2000™ Real-Time Microcontrollers

While this document focuses on HWBIST, for functional safety compliant C2000 devices without HWBIST (like those in the F28004x device family), a C28x Self-Test Library (C28x STL) is provided to test the integrity of the CPU logic using a software-based method. Similarly, HWBIST does not cover the Control Law Accelerator (CLA), so for devices with CLA, a separate CLA Self-Test Library (CLA STL) is available. These libraries are provided upon request—reach out to your local TI Sales Person to request access.

1.1 HWBIST Overview

Figure 1-1 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.

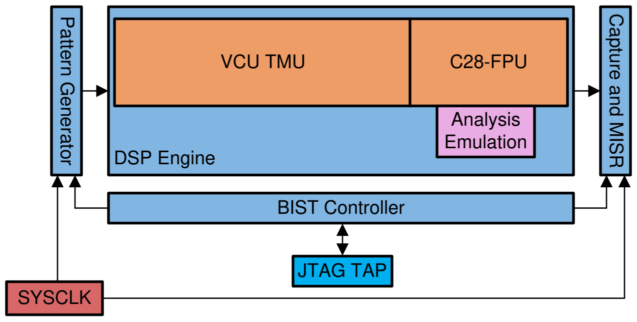

Figure 1-1. HWBIST Block Diagram

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 1-2 shows a block diagram, which includes the additional circuitry to support this option.

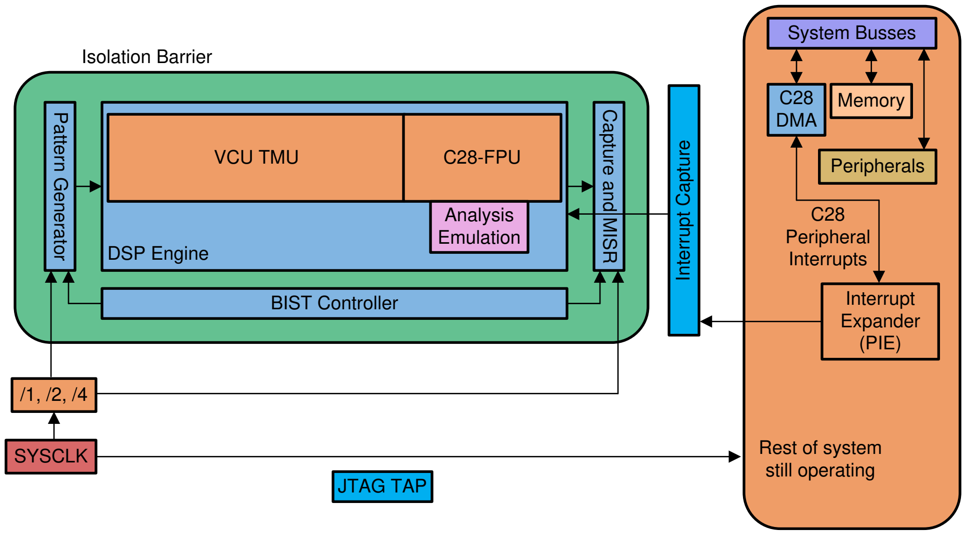

Figure 1-2. HWBIST In-System Block Diagram

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 1-2, 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, mapping 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 where the HWBIST status register can be read

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.

All of the operations listed in the previous outline are executed in the C2000 Software Diagnostic Library (SDL), which is a component of C2000Ware. The SDL provides implementations of functional safety software mechanisms to help system developers meet their safety goals. The details of how the application code can call the provided HWBIST driver are documented in the User's Guide that is included in the SDL package. The SDL User's Guide is the best source for device-specific details about HWBIST, such as supported diagnostic coverage and execution time. Look for this document in `<C2000Ware install directory>/libraries/diagnostic/<device>/docs`.

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 1-3 shows the flow of this action in the state diagram.

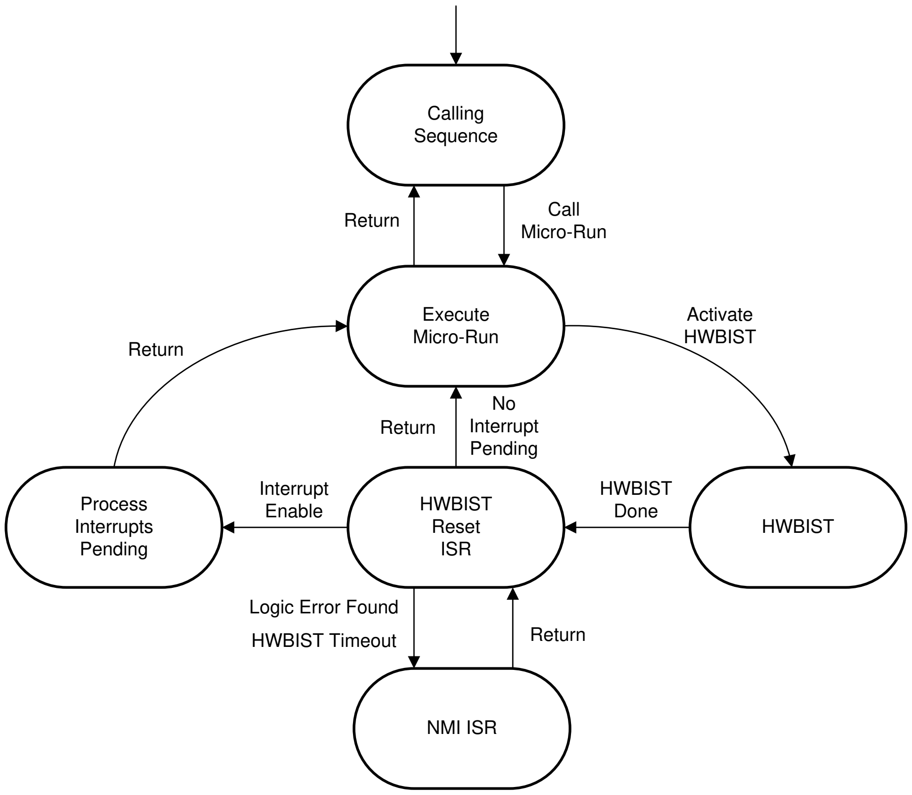

Figure 1-3. HWBIST State Diagram

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 are shown below:

- 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 is most likely 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 measurable 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 are 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 C2000 Software Diagnostic Library. This section only provides a summary of how HWBIST is executed, but additional details are provided in the SDL User's Guide found in C2000Ware. This section also details the considerations for running on dual-core devices and tips for debugging.

Note

The software for the configuration and execution of the HWBIST, released in the C2000 Software Diagnostic Library, generally should not be modified by the user. While minor adjustments for NMI handler or error flag management may be made, changing the HWBIST initialization and overall execution flow may result in a lower diagnostic coverage level than documented.

2.1 Fundamental HWBIST Operation

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 Software Diagnostic Library functions that can be called by the user's application. The function definitions are provided in the header file, stl_hwbist.h. For more details on these function descriptions, see the stl_hwbist.h or the device-specific SDL User's Guide, which is in the /docs folder of the library release package.

There are up to 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);
```

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. If on a multi-core device, this function expects the HWBIST semaphore to be claimed already by the CPU trying to execute a run on the HWBIST. This function initializes the HWBIST registers as follows:

- Number of cycles per micro-run for the first <= 95% coverage and number of cycles per micro-run for the incremental coverage to get to 99% (if supported on that device):
  - Minimizes the time-slice of the micro-run
  - Minimizes the context latency
  - Minimizes the power consumption during the micro-run
- HWBIST clock configuration
- Return address is 0x0000, which is the beginning of the RAMM0 block of memory
- Level of coverage as specified by the input parameter *coverage*

2.1.2 Executing HWBIST

The HWBIST executes a number of micro-run operations until the full coverage is met. The Software Diagnostic Library provides two options for completing HWBIST. The first option is to initialize the HWBIST controller once per full HWBIST run, using STL_HWBIST_init(), and then execute STL_HWBIST_runMicro() periodically until the HWBIST completes. This option allows for smaller time-slicing of the HWBIST. The second option is to call STL_HWBIST_runFull(), 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 run.

2.1.2.1 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 one-time initialization has been performed, you must call the following function:

```
STL_HWBIST_runMicro();
```

This function performs an HWBIST micro-run of the CPU under test and returns the status of the micro-run. This function is designed to be used as a periodic self-test (PEST).

Figure 2-1 shows a flow chart detailing the design of the STL_HWBIST_runMicro() function. This information is also available in the Diagnostic Library User's Guide.

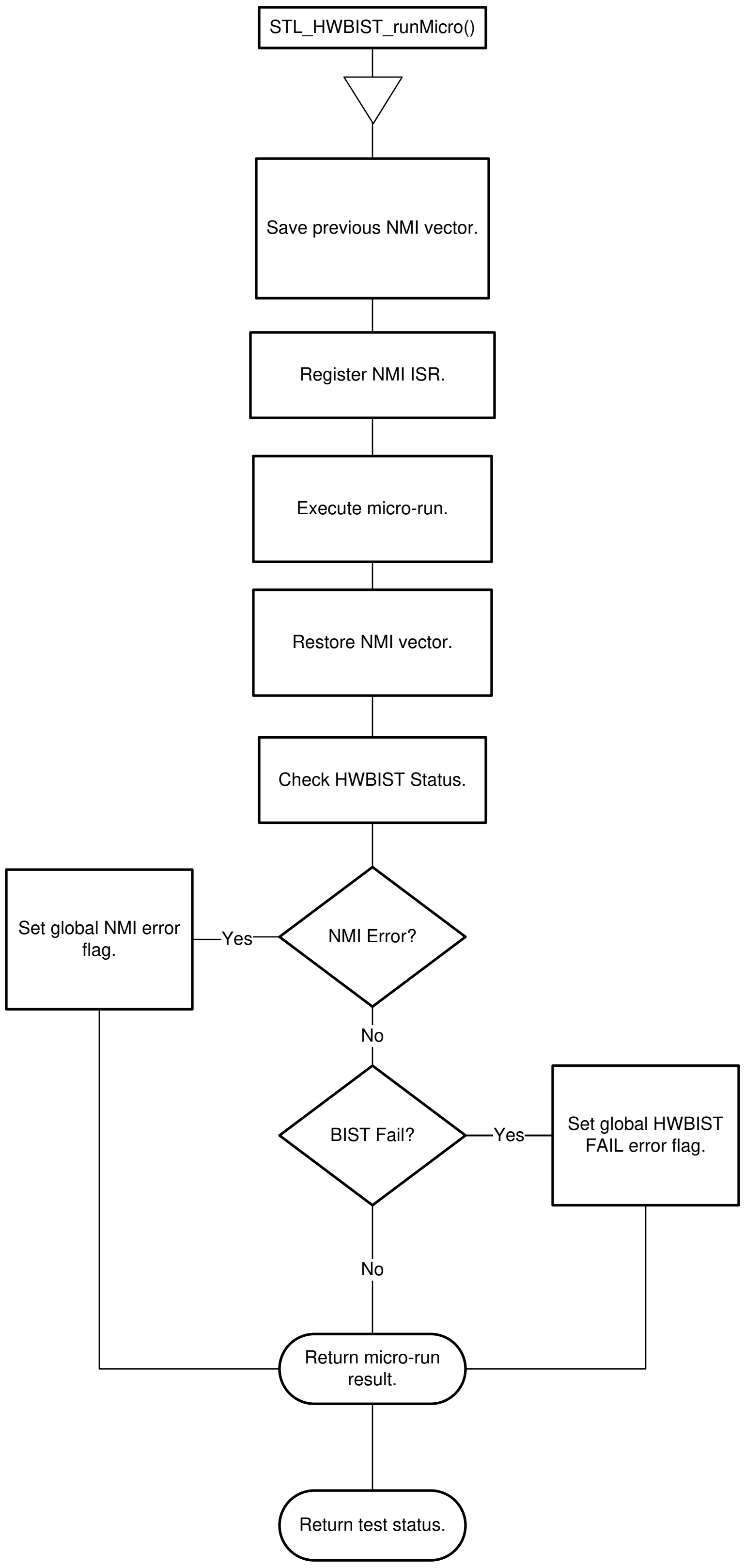

Figure 2-1. STL_HWBIST_runMicro() Flow Chart

To perform a full HWBIST for a coverage level less than or equal to 95% using the STL_HWBIST_runMicro() function, the following sequence of functions must be executed:

1. If on a multi-core device, claim the HWBIST semaphore.

   ```
   STL_HWBIST_claimSemaphore();
   ```

2. Initialize the HWBIST for 95% coverage or less depending on the options available on your device.

   ```
   STL_HWBIST_init(STL_HWBIST_95_LOS);
   ```

3. Execute a HWBIST micro-run.

   ```
   STL_HWBIST_runMicro();
   ```

4. Repeat Step 3 until complete or an error is observed. Execute STL_HWBIST_runMicro() 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. Keep track of the number of times that STL_HWBIST_runMicro() is called. If after the expected number of micro-runs for your coverage level the return value still does not indicate that HWBIST is complete, consider it a time out error. The number of micro-runs is defined in the code as *STL_HWBIST_MICRO_LIMIT_<coverage>* in stl_hwbist.h. For example, on the F28002x STL_HWBIST_MICRO_LIMIT_90 is set to 750, meaning to reach 90% diagnostic coverage the application needs to call STL_HWBIST_runMicro() 750 times.
5. If on a multi-core device, release the HWBIST semaphore.

   ```
   STL_HWBIST_releaseSemaphore();
   ```

To perform a full HWBIST with 99% coverage (only supported on F2837x, F2807x, and F2838x devices) using the STL_HWBIST_runMicro() function, you typically follow the steps above to reach 95%, reinitialize HWBIST for 99% coverage, and then repeatedly call STL_HWBIST_runMicro() until it completes or an error is found. These steps vary slightly from device to device. For more details, see the device-specific SDL User's Guide.

Figure 2-2 shows a flow chart of the setup and execution of a single time-sliced micro-run.

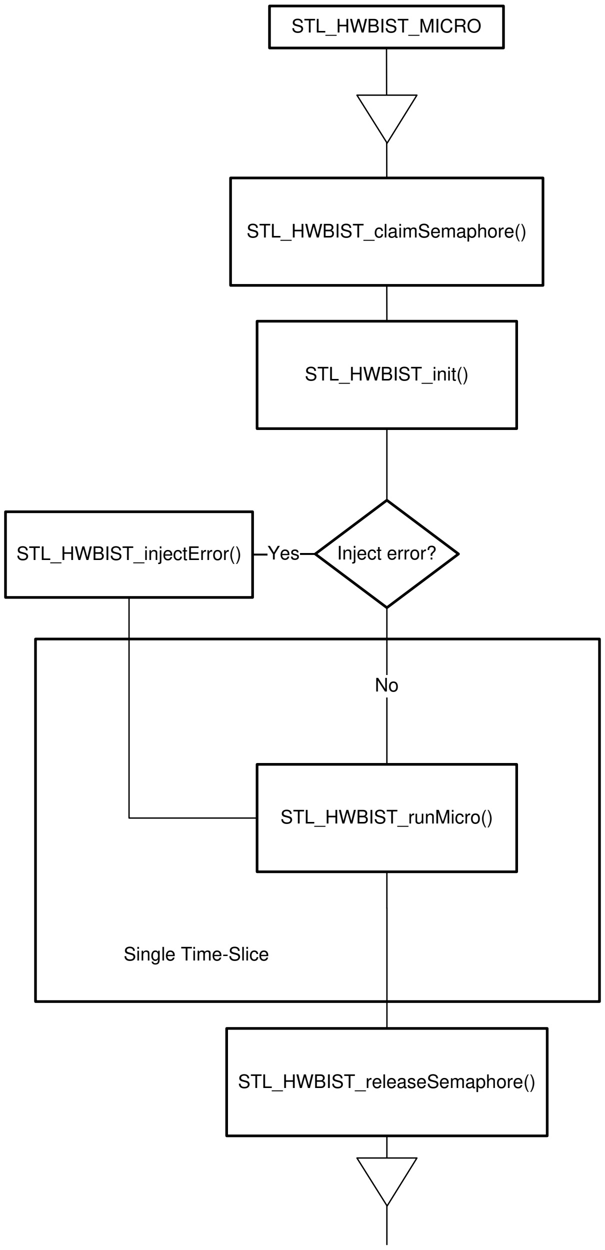

Figure 2-2. Flow Chart of Time-Sliced Micro-Run Execution

2.1.2.2 Executing HWBIST Full-Run

To execute a full run of the HWBIST of the CPU under test, call the following function:

```
STL_HWBIST_runFull();
```

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. If on a multi-core device, this function expects the CPU trying to run a full HWBIST to claim the HWBIST semaphore before calling it. This function initializes the HWBIST engine and then injects the *errorType*. It also registers the STL_HWBIST_errorNMIISR() function as the NMI handler. The function then performs a full HWBIST run, achieving the maximum coverage supported by the device. 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 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*.

For more information about these types of errors, see the SDL User's Guide.

Figure 2-3 shows a flow chart detailing the design of the STL_HWBIST_runFull() function on a device that supports 99% coverage.

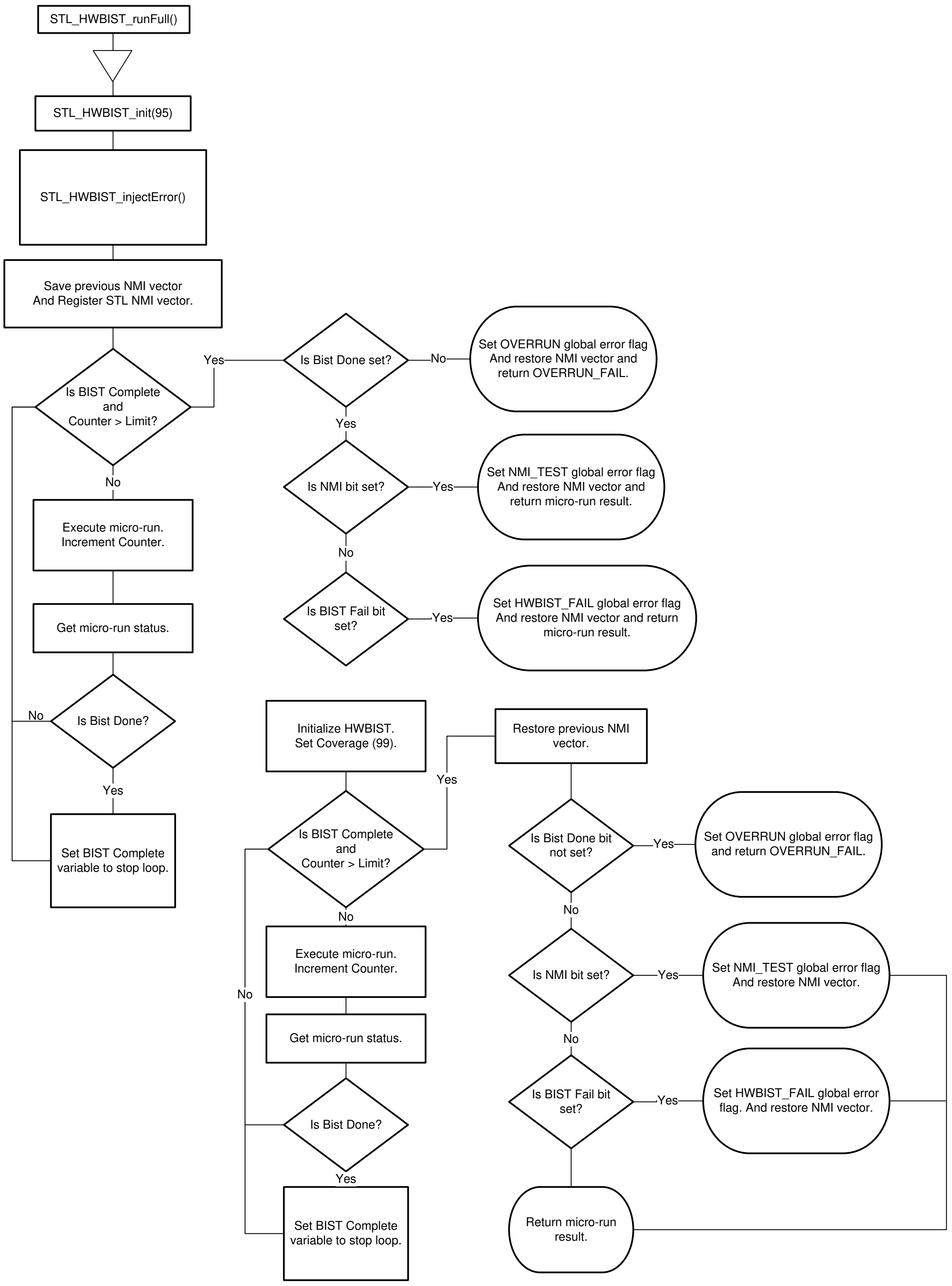
STL_HWBIST_runFull()
STL_HWBIST_init(95)
STL_HWBIST_injectError()
Save previous NMI vector And Register STL NMI vector.
Is BIST Complete and Counter > Limit?
Yes
No
Is Bist Done set?
No
Set OVERRUN global error flag And restore NMI vector and return OVERRUN_FAIL.
Yes
Is NMI bit set?
Yes
Set NMI_TEST global error flag And restore NMI vector and return micro-run result.
No
Is BIST Fail bit set?
Yes
Set HWBIST_FAIL global error flag And restore NMI vector and return micro-run result.
Execute micro-run. Increment Counter.
Get micro-run status.
Is Bist Done?
No
Yes
Set BIST Complete variable to stop loop.
Initialize HWBIST. Set Coverage (99).
Is BIST Complete and Counter > Limit?
Yes
No
Restore previous NMI vector.
Is Bist Done bit not set?
Yes
Set OVERRUN global error flag and return OVERRUN_FAIL.
No
Is NMI bit set?
Yes
Set NMI_TEST global error flag And restore NMI vector.
No
Is BIST Fail bit set?
Yes
Set HWBIST_FAIL global error flag. And restore NMI vector.
Return micro-run result.
Execute micro-run. Increment Counter.
Get micro-run status.
Is Bist Done?
No
Yes
Set BIST Complete variable to stop loop.

Figure 2-3. 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. If on a multi-core device, claim the HWBIST semaphore.

   ```
   STL_HWBIST_claimSemaphore();
   ```

2. Execute a HWBIST full run.

   ```
   STL_HWBIST_runFull();
   ```

3. If on a multi-core device, release the HWBIST semaphore.

   ```
   STL_HWBIST_releaseSemaphore();
   ```

This sequence executes a full HWBIST in a single time-slice. Figure 2-4 shows a full HWBIST run in a single time-slice.

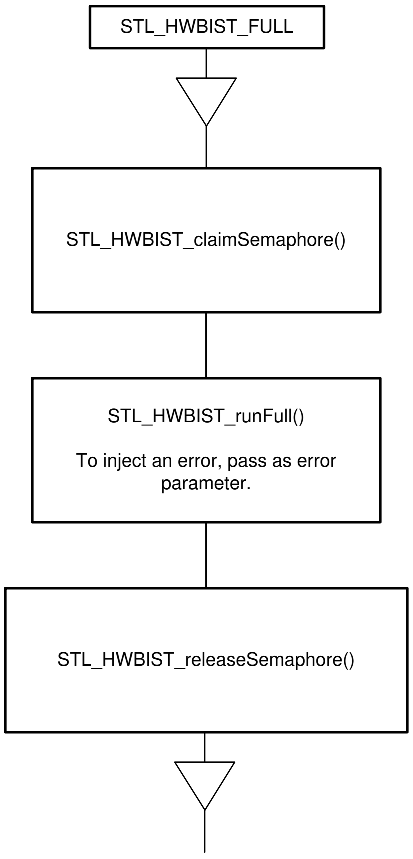
STL_HWBIST_FULL
STL_HWBIST_claimSemaphore()
STL_HWBIST_runFull()
To inject an error, pass as error parameter.
STL_HWBIST_releaseSemaphore()

Figure 2-4. Full HWBIST in Single Time-Slice

2.1.3 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:

   ```
   Interrupt_enablePIE();
   ```

   The ENPIE bit of the PIECTRL register is set on the CPU being tested by the HWBIST software, but it must also be set on the other CPU (in a dual-CPU device) for it to respond to the failure.

For more details about the NMIFLG register and the HWBIST flags it contains, see the device-specific Technical Reference Manual.

2.2 Managing HWBIST on Dual-Core Device

F2837xD and F2838xD 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 is handled using the following function calls:

```
bool STL_HWBIST_claimSemaphore(const STL_HWBIST_Core core);

void 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

The length of time it takes to run a minimum-sized micro-run varies between device families, so you should refer to the profiling data for the STL_HWBIST_runMicro() function provided in the SDL User's Guide. In general, it takes approximately 2.5 µs on 200-MHz SYSCLK devices and approximately 4 µs on 100-MHz SYSCLK devices. These values were measured with the HWBIST functions 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 this delay?*

  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 another 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 in need of an update from the feedback within this delay period?*

  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 this time slice 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 more power than code running on the CPU. You 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 allows 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 requires a source code change and additional testing of the HWBIST functions in the Software 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 in stl_hwbist.c.

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 entry point that the boot ROM jumps to after a HWBIST reset and is used by the *hwbist* memory section. It cannot be moved to another address.
- *hwbiststack* section used for a full context save and restore. It must reside within a SP addressable region. The size of the save/restore is different across device families. At most, like on F2838x, which needs to accommodate the FPU64 registers, 82 words are required.

- Many of the HWBIST functions are placed in *.TI.ramfunc* for best performance. Size may vary slightly depending on device family and compiler version and options. You can remove the CODE_SECTION #pragmas from the HWBIST library files and rebuild the library if you prefer to have them execute from Flash.

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 2-1 lists the injected error types, values, and expected behaviors.

Table 2-1. 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.
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 CPUs 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, you 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 Software Diagnostic Library.

If the code is lost during debugging error management, the most likely cause is that the NMI execution is not appropriately initialized. For more information, see Section 2.1.3.

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 breakpoints, watch points, single stepping, or even running 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 you 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. When running HWBIST on a CPU, it is recommended to remove any association with cross-triggers (if any are set) for that particular CPU.

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.
- You can place loops in the code to "halt" it in place of breakpoints and track its progress.
- 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.
  - If on a dual-core device, it is 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. On a dual-core device, it is possible that the other CPU is still 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, one possible 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 Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F2837xD Dual-Core Microcontrollers Data Sheet*
- Texas Instruments: *TMS320F2837xD Dual-Core Microcontrollers Silicon Errata*
- Texas Instruments: *TMS320F2837xS Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F2837xS Microcontrollers Data Sheet*
- Texas Instruments: *TMS320F2837xS Microcontrollers Silicon Errata*
- Texas Instruments: *TMS320F2807x Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F2807x Microcontrollers Data Sheet*
- Texas Instruments: *TMS320F2807x Microcontrollers Silicon Errata*
- Texas Instruments: *Functional Safety Manual for TMS320F2837xD, TMS320F2837xS and TMS320F2807x*
- Texas Instruments: *TMS320F2838x Real-Time Microcontrollers With Connectivity Manager Technical Reference Manual*
- Texas Instruments: *TMS320F2838x Real-Time Microcontrollers With Connectivity Manager Data Sheet*
- Texas Instruments: *TMS320F2838x Real-Time Microcontrollers Silicon Errata*
- Texas Instruments: *TMS320F28003x Real-Time Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F28003x Real-Time Microcontrollers Data Sheet*
- Texas Instruments: *TMS320F28003x Real-Time Microcontrollers Silicon Errata*
- Texas Instruments: *TMS320F28002x Real-Time Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F28002x Real-Time Microcontrollers Data Sheet*
- Texas Instruments: *TMS320F28002x Real-Time Microcontrollers Silicon Errata*
- Texas Instruments: *Safety Manual for TMS320F28002x*
- Texas Instruments: *Automotive Functional Safety for C2000™ Real-Time Microcontrollers*
- Texas Instruments: *Industrial Functional Safety for C2000™ Real-Time Microcontrollers*
- C2000™ SafeTI™ Diagnostic Software Library for F2837xD, F2837xS, and F2807x Devices
- C2000Ware, contains Software Diagnostic Library for F2838x, F28003x, and F28002x

4 Revision History

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

Changes from Revision * (October 2017) to Revision A (September 2022) **Page**

- Updated the numbering format for tables, figures and cross-references throughout the document....................2
- Updated Section 2.4.. 18
- Updated Section 3.. 19