




C2000™ Software Diagnostic Library for TMS320F28002x

API USER'S GUIDE

Copyright

Copyright © 2021 Texas Instruments Incorporated. All rights reserved. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
<http://www.ti.com/c2000>



Revision Information

This is version 2.01.00 of this document, last updated on Fri Feb 12 19:23:45 IST 2021.

Table of Contents

Copyright	1
Revision Information	1
1 Introduction	3
1.1 Terms and Abbreviations	4
1.2 References	4
2 Diagnostic Self-Test Library	5
3 Using the Diagnostic STL	6
3.1 Overview of the Library	6
3.2 Using the Library	8
4 Library Modules	9
4.1 CAN Message RAM Test API Functions	9
4.2 CPU Register API Functions	13
4.3 CRC API Functions	16
4.4 HWBIST API Functions	20
4.5 March13N Test API Functions	26
4.6 Oscillator CPU Timer API Functions	32
4.7 Oscillator HRPWM API Functions	35
4.8 PIE RAM API Functions	37
4.9 Utilities API Functions	41
5 Self-Test Application	44
5.1 Getting Started	44
5.2 Example Behavior	45
5.3 STA_Tests API Functions	47
5.4 STA_Timer API Functions	49
5.5 STA_Comm API Functions	51
5.6 STA_Util API Functions	52
5.7 STA_User API Functions	54
6 Integration Notes	56
6.1 Cycle Time	56
6.2 Memory Usage	57
7 Safety Feature and Diagnostic Examples	58
7.1 Using DCSM for Freedom From Interference (FFI)	58
7.2 Test of ECC logic in Flash	59
7.3 Test of Flash Prefetch, Data Cache and Wait-States	59
7.4 Missing clock detection	60
7.5 RAM access protection violation detection	60
7.6 Test of parity/ECC logic in SRAM	61
7.7 Software test of watchdog operation	62
IMPORTANT NOTICE	63

1 Introduction

Terms and Abbreviations	4
References	4

This is a release of the C2000™ Software Diagnostic Library for F28002x devices. The Diagnostic Library is a collection of self-test libraries (STL) and examples. This document will provide an overview of the Software Diagnostic Library (SDL) and its functions.

Manufacturers of end equipments must take steps to ensure safe and reliable operation of their products in order to meet specific industry safety standards. Specific safety measures and diagnostic tests are recommended or required in order to meet each particular safety standard. The SDL is designed to enable customers to more efficiently and effectively work to meet such safety standards by their systems.

Note that the SDL has not itself undergone any third-party assessment or certification to a particular standard. It is intended to provide reference implementations of diagnostics and usage examples of hardware diagnostic features. The system integrator should consider which safety mechanisms are required by the standard they are targeting and whether the SDL will aid in achieving those safety mechanisms.

Considerations for integrating the SDL should include but are not limited to:

- the safety mechanism as such
- the targeted faults of the diagnostic test
- the API definitions (parameters, functionality, return values)
- the global error flags which could be set on a detected fault
- the error injection techniques available
- the memory requirements of the software components
- the time or cycle requirements for the diagnostic test
- the effect of the diagnostic test on the state of the peripheral and/or system
- the interrupt service routines used in the SDL

The source code of the SDL is provided to enable the system integrator to modify and test the software components as desired. For instance, a subset of SDL components utilize their own interrupt service routines. It may be desirable to integrate the functionality of the SDL ISRs into the system ISRs. In this case, the same ISR could be used to serve the purpose of the system as well as the diagnostic test.

For more information about diagnostic libraries for C2000 microcontrollers, visit the [C2000 Safety Software Libraries page on ti.com](http://ti.com).

1.1 Terms and Abbreviations

API	Application Programming Interface
C2000Ware	A cohesive set of development software and documentation designed to minimize software development time. From device-specific drivers and libraries to device peripheral examples, C2000Ware provides a solid foundation to begin development and evaluation of your product.
C28x Driverlib	A set of header files describing the hardware and a set of drivers that use those header files to access the peripherals. The Driverlib and its examples are intended to be used with Code Composer Studio and TI's C2000 C Compiler. They will be delivered within C2000Ware.
CCS	Code Composer Studio
PEST	Periodic Self-Test
POST	Power-On Self-Test
SFO	Scale Factor Optimization
SM	Safety Mechanisms
STA	Self-Test Application
STL	Self-Test Library
SW	Software

1.2 References

SPRSP45	TMS320F28002x Piccolo™ Microcontrollers datasheet
SPRZ466	TMS320F28002x MCUs silicon errata
SPRUIN7	TMS320F28002x Microcontrollers Technical Reference Manual
SAFETI_CQKIT	SafeTI™ Compiler Qualification Kit

2 Diagnostic Self-Test Library

Current and future C2000™ microcontrollers are designed with integrated features that can be used in safety-critical systems. Manufacturers of end equipment must take steps to ensure safe and reliable operation of their systems in order to meet the relevant standards.

Hardware features such as write-protected registers and supervisory circuits have all been integrated in C2000 MCUs. These features, leveraged to the diagnostic test and safety mechanism requirements of the targeted safety standard, make compliance in the electronic segment of the tests more easily achievable.

The C2000™ Software Diagnostic Library is a collection of optimized independent test functions for C2000 MCUs. The library includes C-callable optimized test functions that return the status of the test performed. In case of a failure returned signifying a detected fault, it is up to the user application to determine the appropriate action to take.

The SDL consists of individual functions that are called and managed by the user's application software. Each function is designed with a specific task to verify functionality of a device component using a specific safety mechanism consistent with safety standards with minimal impact on the real-time control performance of the target MCU. STL functions are provided to support POST, PEST, or both.

In addition to the library modules, the SDL also contains examples to demonstrate the use of the library and the configuration of hardware diagnostic features.

3 Using the Diagnostic STL

Overview of the Library	6
Using the Library	8

3.1 Overview of the Library

The following subsections briefly list and describe the error reporting mechanisms, contents, and coverage of the diagnostic library. The library is delivered with several example CCS projects. The `test_application` demonstrates the use of all the library modules. See the Self-Test Application section of this document for more details on the example.

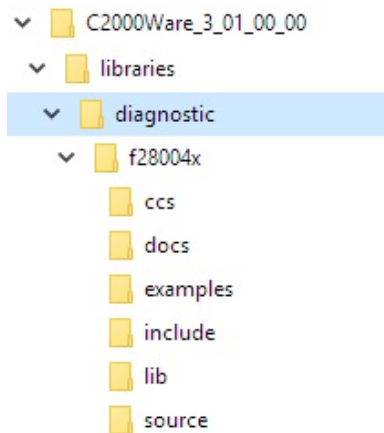


Figure 3.1: Directory Structure

Error Reporting

All the API functions in the diagnostic library return the status of the test. If the test fails, the functions set a unique error code to a 32-bit global variable, **STL_Util_globalErrorFlags**. All codes are defined in `stl_util.h` and start with **STL_UTIL_** followed by a representation of the test. For example, **STL_UTIL_OSC_TIMER2** represents the error code the oscillator frequency test.

Library Files

The diagnostic library consists of the following files. All source files with a `.c` and `.asm` extension have an associated header file that contains function prototypes and data structures. For example for `stl_osc_ct.c` there is an associated `stl_osc_ct.h` file that declares the function prototypes, data types, and `#defines`.

File Name	Source Description
<code>stl_can_ram.c</code>	CAN Message RAM parity test

stl_cpu_reg.asm	C-callable assembly test of CPU, FPU, VCRC registers
stl_crc.c	CRC code for E2E communications and memory tests
stl_crc_s.asm	C-callable assembly CRC code
stl_march.c	March test
stl_march_s.asm	C-callable assembly for March test
stl_osc_ct.c	Internal OSC Frequency test
stl_osc_hr.c	High Resolution OSC (HRPWM) Frequency test
stl_pie_ram.c	PIE RAM Redundancy test
stl_util.c	Utility functions, primarily for managing error flags

3.2 Using the Library

This section describes all the necessary steps that are required to rebuild and use the library in an application.

Library Build Options

The current version of the library was built with CCSv10 using C28x Codegen Tools 20.2.1.LTS with the following options:

- -v28
- -ml
- -mt
- -vcu_support=vcrc
- -float_support=fpu32
- -tmu_support=tmu0

Rebuilding the Library

The diagnostic safety library ships with the original CCS projectspec file that was used to build the library. This project can be used as a template to modify and rebuild the library.

1. Create a new workspace.
2. Import the projectspec file found in `..\ccs\`
3. Make the required changes.
4. Right click project and select Build Configurations -> Clean All and when that completes Build Configurations -> Build All.
5. The new library will be placed in the `..\lib\` folder.

4 Library Modules

CAN Message RAM API Functions	9
CPU Registers Test API Functions	13
CRC API Functions	16
HWBIST API Functions	20
March13N Test API Functions	26
Oscillator CPU Timer API Functions	32
Oscillator HRPWM API Functions	35
PIE RAM API Functions	37
Utilities API Functions	41

4.1 CAN Message RAM Test API Functions

Macros

- #define **STL_CAN_RAM_PASS**
- #define **STL_CAN_RAM_ERROR**
- #define **STL_CAN_RAM_ADDR_OFFSET**
- #define **STL_CAN_RAM_MAX_ADDR_OFFSET**
- #define **STL_CAN_RAM_PARITY_OFF**
- #define **STL_CAN_RAM_NO_COPY**

Functions

- void **STL_CAN_RAM_testRAM** (const uint32_t canBase, const **STL_March_Pattern** pattern, const uint32_t startAddress, const uint32_t endAddress, const uint32_t copyAddress)
- void **STL_CAN_RAM_injectError** (const uint32_t canBase, const uint32_t address, const uint32_t xorMask)
- uint16_t **STL_CAN_RAM_checkErrorStatus** (const uint32_t canBase)

4.1.1 Detailed Description

The code for this module is contained in `source/stl_can_ram.c` and `source/stl_can_ram_s.asm`, with `include/stl_can_ram.h` containing the API declarations for use by applications.

Error Injection

Here are techniques to inject error:

STL_CAN_RAM_testRAM(), **STL_CAN_RAM_checkErrorStatus()**

- Call **STL_CAN_RAM_injectError()** with an *xorMask* value that will cause errors.

4.1.2 Macro Definition Documentation

4.1.2.1 #define STL_CAN_RAM_NO_COPY

Value to indicate to [STL_CAN_RAM_testRAM\(\)](#) through the **copyAddress** parameter that the test should run a destructive test on the region of CAN message RAM without saving and restoring its contents.

Referenced by [STA_Tests_testDevice\(\)](#).

4.1.3 Function Documentation

4.1.3.1 void STL_CAN_RAM_testRAM (const uint32_t *canBase*, const **STL_March_Pattern** *pattern*, const uint32_t *startAddress*, const uint32_t *endAddress*, const uint32_t *copyAddress*)

Performs a March13N memory test on the specified range of CAN message RAM objects.

Parameters

<i>canBase</i>	is the base address of the CAN controller.
<i>pattern</i>	is the test pattern to use.
<i>startAddress</i>	is the starting address (inclusive) of the CAN message RAM range to test.
<i>endAddress</i>	is the end address (inclusive) of the CAN message RAM range to test.
<i>copyAddress</i>	is the address to copy the original contents of the memory under test. It will be used to restore the original memory at the end of the March13N memory test. If no save and restore is required, use a value of STL_CAN_RAM_NO_COPY .

This function performs a March13N memory test on the range of CAN message RAM objects specified by **canBase**, **startAddress** and **endAddress**. The test can save and restore the original contents of the message RAM by passing an address to a back up buffer through the **copyAddress** parameter. The test will copy the original contents of the memory to **copyAddress**, perform the memory test, and then copy the original contents back to the memory under test. To skip the save and restore, use a value of **STL_CAN_RAM_NO_COPY** for the **copyAddress** and a destructive test will be performed instead.

The test patterns and the March13N memory test algorithm provided test the memory for stuck-at-faults. The parity bits will show if any were detected. Use [STL_CAN_RAM_checkErrorStatus\(\)](#) to read the parity status.

Note

Note to take care calculating the size of memory needed for **copyAddress**. The March13N function is not specific to the CAN message RAM and does not take the byte addressability of the memory into account. Use a buffer twice the size of the actual message RAM region you are testing.

The STL_March functions called by this test disable global CPU interrupts (DINT) and then re-enable them after the test has completed.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.1.3.2 void STL_CAN_RAM_injectError (const uint32_t *canBase*, const uint32_t *address*, const uint32_t *xorMask*)

Injects an error into a CAN message RAM address.

Parameters

<i>canBase</i>	is the base address of the CAN controller.
<i>address</i>	is the address of the word in the message RAM where the error will be injected.
<i>xorMask</i>	mask of the bit to flip in address .

This function injects an error at a specific memory **address**. **xorMask** specifies which bit to flip in order to corrupt the data or parity bits.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.1.3.3 uint16_t STL_CAN_RAM_checkErrorStatus (const uint32_t *canBase*)

Returns the status of the CAN message RAM parity error detected bit.

Parameters

<i>canBase</i>	is the base address of the CAN controller.
----------------	--

This function checks if any parity errors have been detected by the CAN message RAM parity logic.

Returns

If the CAN error status register indicates a parity error the function returns **STL_CAN_RAM_ERROR**. Otherwise, the function returns **STL_CAN_RAM_PASS**.

Note

Several bits in the CAN error status register (CAN_ES) are cleared on a CPU read of the register which this function performs. If the potential clearing of these bits (see device Technical Reference Manual for details) is undesired behavior for your application, use different means of reading the status.

Referenced by [STA_Tests_testDevice\(\)](#), and [STA_User_canParityErrorISR\(\)](#).

4.2 CPU Register API Functions

Macros

- `#define STL_CPU_REG_PASS`
- `#define STL_CPU_REG_FAIL`

Functions

- `uint16_t STL_CPU_REG_testCPURegisters (bool injectError)`
- `uint16_t STL_CPU_REG_testFPURegisters (bool injectError)`
- `uint16_t STL_CPU_REG_testVCRCRegisters (uint32_t *scratchRAM, bool injectError)`
- `static uint16_t STL_CPU_REG_checkCPURegisters (bool injectError)`
- `static uint16_t STL_CPU_REG_checkFPURegisters (bool injectError)`
- `static uint16_t STL_CPU_REG_checkVCRCRegisters (bool injectError)`

4.2.1 Detailed Description

The code for this module is contained in `source/stl_cpu_reg.c` and `source/stl_cpu_reg.s.asm`, with `include/stl_cpu_reg.h` containing the API declarations for use by applications.

Error Injection

The functions in this module all have an *injectError* parameter that when set to true, will overwrite an incorrect test pattern to a register instead of the expected pattern in order to generate a failure.

4.2.2 Function Documentation

4.2.2.1 `static uint16_t STL_CPU_REG_checkCPURegisters (bool injectError)`
`[inline], [static]`

Tests CPU registers.

Parameters

<i>injectError</i>	when <i>false</i> the test writes the test pattern to the registers as expected. When <i>true</i> , the test will write an unexpected error pattern to ACC to simulate a stuck bit, causing the test to fail.
--------------------	---

This function tests CPU core registers for stuck bits. The following registers are tested:

- ACC
- P
- XAR0 to XAR7
- XT

- SP
- IFR, IER and DBGIER
- ST0
- ST1 (excluding IDLESTAT and LOOP bits)
- DP

The values of ST0, ST1, DP, IER, IFR, and DBGIER and the save-on-entry XAR registers, as defined by the compiler calling convention, are saved and restored in this test.

Note that the IDLESTAT and LOOP bits of ST1 are not tested by this function as they are read-only. IDLESTAT is set when the IDLE instruction is used to enter a low-power mode. Performing a functional test of the low-power mode will indirectly serve as a test of IDLESTAT.

The LOOP bit is set by the LOOPZ and LOOPNZ instructions. The C2000 compiler will not generate code that uses these instructions. If an application contains hand-coded assembly that uses them, the LOOP bit may also be tested by performing a functional test of the applicable LOOP instruction.

Note

You should disable interrupts before calling this test.

Returns

If the test passes, the routine returns **STL_CPU_REG_PASS**. Otherwise, it returns **STL_CPU_REG_FAIL**.

References [STL_UTIL_CPU_REG](#), and [STL_Util_setErrorFlag\(\)](#).

Referenced by [STA_Tests_testDevice\(\)](#).

4.2.2.2 `static uint16_t STL_CPU_REG_checkFPURegisters (bool injectError)`
`[inline], [static]`

Tests FPU registers.

Parameters

<i>injectError</i>	when <i>false</i> the test writes the test pattern to the registers as expected. When <i>true</i> , the test will write an unexpected error pattern to R5H to simulate a stuck bit, causing the test to fail.
--------------------	---

This function tests FPU registers for stuck bits. The following registers are tested:

- R0H to R7H
- RND32, TF, ZI, NI, ZF, NF bits of STF register.
- Shadow registers for R0H to R7H and STF

The values of STF and the save-on-entry RnH registers, as defined by the compiler calling convention, are saved and restored in this test.

Note

You should disable interrupts before calling this test.

Returns

If the test passes, the routine returns **STL_CPU_REG_PASS**. Otherwise, it returns **STL_CPU_REG_FAIL**.

References [STL_UTIL_FPU_REG](#), and [STL_Util_setErrorFlag\(\)](#).

Referenced by [STA_Tests_testDevice\(\)](#).

4.2.2.3 `static uint16_t STL_CPU_REG_checkVCRCRegisters (bool injectError)`
`[inline], [static]`

Tests VCRC registers.

Parameters

<i>injectError</i>	when <i>false</i> the test writes the test pattern to the registers as expected. When <i>true</i> , the test will write an unexpected error pattern to VCRCPOLY to simulate a stuck bit, causing the test to fail.
--------------------	--

This function tests VCRC (Cyclic Redundancy Check Unit) registers for stuck bits. The following registers are tested:

- VCRCPOLY
- VCRCSIZE
- VCRC
- VSTATUS

The values of VCRCPOLY, VCRCSIZE, and VSTATUS are saved and restored in this test.

Note

You should disable interrupts before calling this test.

Returns

If the test passes, the routine returns **STL_CPU_REG_PASS**. Otherwise, it returns **STL_CPU_REG_FAIL**.

References [STL_Util_setErrorFlag\(\)](#), and [STL_UTIL_VCRC_REG](#).

Referenced by [STA_Tests_testDevice\(\)](#).

4.3 CRC API Functions

Data Structures

- struct [STL_CRC_Obj](#)

Macros

- #define [STL_CRC_INIT_CRC](#)
- #define [STL_CRC_PASS](#)
- #define [STL_CRC_FAIL](#)

Typedefs

- typedef [STL_CRC_Obj](#) * [STL_CRC_Handle](#)

Enumerations

- enum [STL_CRC_Parity](#) { [STL_CRC_PARITY_EVEN](#), [STL_CRC_PARITY_ODD](#) }

Functions

- void [STL_CRC_reset](#) (void)
- void [STL_CRC_calculate](#) (const [STL_CRC_Handle](#) crcHandle)
- void [STL_CRC_calculateLowBytes](#) (const [STL_CRC_Handle](#) crcHandle)
- uint16_t [STL_CRC_checkCRC](#) (const uint32_t startAddress, const uint32_t endAddress, const uint32_t goldenCRC)

4.3.1 Detailed Description

The code for this module is contained in `source/stl_crc.c` and `source/stl_crc_s.asm` (F2837x only), with `include/stl_crc.h` containing the API declarations for use by applications.

4.3.2 Data Structure Documentation

4.3.2.1 struct STL_CRC_Obj

CRC structure.

Data Fields

uint32_t	seedValue	Initial value of the CRC calculation.
uint16_t	numBytes	Number of bytes in the message buffer.
STL_CRC_Parity	parity	Start the CRC from the low byte or high byte
uint32_t	crcResult	The calculated CRC.
void *	msgBuffer	Pointer to the message buffer.

4.3.3 Enumeration Type Documentation

4.3.3.1 enum **STL_CRC_Parity**

Parity enumeration.

The parity is used by the CRC algorithm to determine whether to begin calculations from the low byte (EVEN) or from the high byte (ODD) of the first word (16-bit) in the message.

For example, if your message had 10 bytes and started at the address 0x8000 but the first byte was at the high byte position of the first 16-bit word, the user would call the CRC function with odd parity i.e. `STL_CRC_PARITY_ODD`

Address: HI LO

0x8000 : B0 XX

0x8001 : B2 B1

0x8002 : B4 B3

0x8003 : B6 B5

0x8004 : B8 B7

0x8005 : XX B9

However, if the first byte was at the low byte position of the first 16-bit word, the user would call the CRC function with even parity i.e. `STL_CRC_PARITY_EVEN`

Address: HI LO

0x8000 : B1 B0

0x8001 : B3 B2

0x8002 : B5 B4

0x8003 : B7 B6

0x8004 : B9 B8

Enumerator

`STL_CRC_PARITY_EVEN` Even parity, CRC starts at the low byte of the first word (16-bit)

`STL_CRC_PARITY_ODD` Odd parity, CRC starts at the high byte of the first word (16-bit)

4.3.4 Function Documentation

4.3.4.1 void `STL_CRC_reset` (void)

Workaround to the silicon issue of first VCU calculation on power up being erroneous.

Due to the internal power-up state of the VCU module, it is possible that the first CRC result will be incorrect. This condition applies to the first result from each of the eight CRC instructions. This rare condition can only occur after a power-on reset, but will not necessarily occur on every power on. A warm reset will not cause this condition to reappear.

Workaround(s): The application can reset the internal VCU CRC logic by performing a CRC calculation of a single byte in the initialization routine. This routine only needs to perform one CRC calculation and can use any of the CRC instructions.

Returns

None.

4.3.4.2 void STL_CRC_calculate (const **STL_CRC_Handle** *crcHandle*)

Runs the 32-bit CRC routine using polynomial 0x04c11db7.

Parameters

<i>crcHandle</i>	handle to the CRC object
------------------	--------------------------

Calculates the 32-bit CRC using polynomial 0x04c11db7 on the VCU or VCRC. Depending on the parity chosen the CRC begins at either the low byte (STL_CRC_PARITY_EVEN) or the high byte (STL_CRC_PARITY_ODD) of the first word (16-bit).

Note

The size of the message (bytes) is limited to 65535 bytes.

Returns

None.

4.3.4.3 void STL_CRC_calculateLowBytes (const **STL_CRC_Handle** *crcHandle*)

Runs the 32-bit CRC routine using polynomial 0x04c11db7.

Parameters

<i>crcHandle</i>	handle to the CRC object
------------------	--------------------------

Calculates the 32-bit CRC using polynomial 0x04c11db7 on the VCU or VCRC. This algorithm performs a CRC32 only on the low bytes (LSB) of each 16-bit word. The input **parity** has no effect. This function works on unpacked data.

Note

The size of the message (bytes) is limited to 65535 bytes.

Returns

None.

4.3.4.4 uint16_t STL_CRC_checkCRC (const uint32_t *startAddress*, const uint32_t *endAddress*, const uint32_t *goldenCRC*)

Calculates a CRC-32 value for specific memory range and compares it with the goldenCRC value.

Parameters

<i>startAddress</i>	- start address of CRC calculation.
<i>endAddress</i>	- end address of CRC calculation, inclusive.
<i>goldenCRC</i>	- golden CRC value.

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

Note

This function could be used with many memory types including Flash and Boot ROM.

Returns

If the calculated CRC matches the golden CRC, then the function returns **STL_CRC_PASS**.
Otherwise, it returns **STL_CRC_FAIL**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.4 HWBIST API Functions

Macros

```

■ #define STL_HWBIST_BIST_DONE
■ #define STL_HWBIST_MACRO_DONE
■ #define STL_HWBIST_NMI
■ #define STL_HWBIST_BIST_FAIL
■ #define STL_HWBIST_INT_COMP_FAIL
■ #define STL_HWBIST_TO_FAIL
■ #define STL_HWBIST_OVERRUN_FAIL
■ #define HWBIST_O_CSTCGCR0
■ #define HWBIST_O_CSTCGCR1
■ #define HWBIST_O_CSTCGCR2
■ #define HWBIST_O_CSTCGCR3
■ #define HWBIST_O_CSTCGCR4
■ #define HWBIST_O_CSTCGCR5
■ #define HWBIST_O_CSTCGCR6
■ #define HWBIST_O_CSTCGCR7
■ #define HWBIST_O_CSTCGCR8
■ #define HWBIST_O_CSTCPCNT
■ #define HWBIST_O_CSTCCONFIG
■ #define HWBIST_O_CSTCSADDR
■ #define HWBIST_O_CSTCTEST
■ #define HWBIST_O_CSTCRET
■ #define HWBIST_O_CSTCCRD
■ #define HWBIST_O_CSTGSTAT
■ #define HWBIST_O_CSTCCPCR
■ #define HWBIST_O_CSTCCADDR
■ #define HWBIST_O_CSTCSEM
■ #define HWBIST_CSTCSEM_SEMAPHORE
■ #define PIEVECTTABLE_O_NMI
■ #define STL_HWBIST_REF_STACK
■ #define STL_HWBIST_MOV_SP_STACK
■ #define STL_HWBIST_C28OBJ
■ #define STL_HWBIST_C28ADDR
■ #define STL_HWBIST_C28MAP
■ #define STL_HWBIST_CLRC_PAGE0
■ #define STL_HWBIST_MO VW_DP_0
■ #define STL_HWBIST_CLRC_OVM
■ #define STL_HWBIST_SPM_0
■ #define STL_HWBIST_REF_HANDLE_RESET_FXN
■ #define STL_HWBIST_LCR_HANDLE_RESET_FXN

```

Enumerations

```

■ enum STL_HWBIST_Error {
    STL_HWBIST_NO_ERROR, STL_HWBIST_TIMEOUT, STL_HWBIST_FINAL_COMPARE,
    STL_HWBIST_NMI_TRAP,
    STL_HWBIST_LOGIC_FAULT }
■ enum STL_HWBIST_Coverage { STL_HWBIST_90_LOS }

```

Functions

- `__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)`
- `static void STL_HWBIST_injectError (const STL_HWBIST_Error errorType)`

4.4.1 Detailed Description

The code for this module is contained in `source/stl_hwbist.c` and `source/stl_hwbist_s.asm`, with `include/stl_hwbist.h` containing the API declarations for use by applications.

Error Injection

Here are techniques to inject error:

`STL_HWBIST_runFull()`

- Call `STL_HWBIST_injectError()` with a parameter of `STL_HWBIST_Error` type.

4.4.2 Enumeration Type Documentation

4.4.2.1 enum **STL_HWBIST_Error**

Values that must be used as parameter to `STL_HWBIST_injectError()` and `STL_HWBIST_runFull()` in order to specify the type of error to inject before executing HWBIST.

Enumerator

- `STL_HWBIST_NO_ERROR`** No error.
- `STL_HWBIST_TIMEOUT`** Time-out error.
- `STL_HWBIST_FINAL_COMPARE`** Final MISR compare error.
- `STL_HWBIST_NMI_TRAP`** NMI trap error.
- `STL_HWBIST_LOGIC_FAULT`** Logic error.

4.4.2.2 enum **STL_HWBIST_Coverage**

Values that must be used as a parameter to `STL_HWBIST_init()` in order to initialize the HWBIST engine before a micro-run for a specific target coverage. Note that not all devices support multiple coverage levels.

Enumerator

- `STL_HWBIST_90_LOS`** 90% launch-on-shift

4.4.3 Function Documentation

4.4.3.1 `uint16_t STL_HWBIST_runFull (const STL_HWBIST_Error errorType)`

Performs a hardware built-in self-test of the CPU under test.

Parameters

<i>errorType</i>	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 initializes the HWBIST engine and then injects the **errorType**. It also registers the STL_HWBIST_NMIISR as the NMI vector. It then performs a full hardware built-in self-test achieving the 90% launch-on-shift coverage after 750 micro-runs. If there is a failure in the HWBIST, then a global error flag will be set and the return value will specify a failure. Additionally, if the coverage is not achieved in the expected number of micro-runs, then the test will fail due to an overrun. Before returning, the function will restore the previous NMI vector.

Note

There is a corner case where a spurious CPU Timer 1 or CPU Timer 2 interrupt may be triggered when HWBIST completes. This test contains a workaround which clears the TIE bits of Timer 1 and 2 before starting HWBIST interrupt logging and restores them after HWBIST runs.

Returns

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 **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 will have contain 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**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.4.3.2 uint16_t STL_HWBIST_runMicro (void)

Performs a micro-run of the hardware built-in self-test of the CPU under test.

This function expects the HWBIST engine to already be initialized with [STL_HWBIST_init\(\)](#) before it is called. This function performs a HWBIST micro-run and returns its status. Before returning, the function will also restore the previous NMI vector.

In order to achieve 90% coverage, the user needs to initialize the HWBIST controller for 90% launch-on-shift coverage and then execute 750 micro-runs. The HWBIST will complete in 750 micro-runs if there are no detected faults.

Note

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

There is a corner case where a spurious CPU Timer 1 or CPU Timer 2 interrupt may be triggered when HWBIST completes. This test contains a workaround which clears the TIE bits of Timer 1 and 2 before starting HWBIST interrupt logging and restores them after HWBIST runs.

Returns

If the HWBIST micro-run test passes with no errors, then this function returns the status of the HWBIST and the value will be either **STL_HWBIST_MACRO_DONE** or a bitwise OR of **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 will have contain 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**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.4.3.3 void STL_HWBIST_restoreContext (void)

Begins the context restore after a CPU reset after a HWBIST micro-run.

This function should not be called by the user, but must be placed at the beginning of RAMM0, memory address 0x0000. After a HWBIST micro-run completes, the CPU will reset and begin executing instructions from 0x0000. The user is responsible for placing this function at memory address 0x0000. This can be done using the linker command file and program sections.

Returns

None.

4.4.3.4 void STL_HWBIST_init (const **STL_HWBIST_Coverage** coverage)

Initializes the HWBIST engine for operation.

Parameters

<i>coverage</i>	is an enumerated type STL_HWBIST_Coverage which specifies the coverage to achieve.
-----------------	---

This function initializes the HWBIST engine for the specified level of coverage. This function is intended to be used with [STL_HWBIST_runMicro\(\)](#). This function should be called once to initialize the HWBIST and not called again until the HWBIST is done which is indicated by a return value of **STL_HWBIST_BIST_DONE** as this function resets the HWBIST engine.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.4.3.5 static void STL_HWBIST_injectError (const **STL_HWBIST_Error** errorType) [inline], [static]

Injects an error into the HWBIST engine for operation.

Parameters

<i>errorType</i>	is an enumerated type STL_HWBIST_Error which specifies the error to inject.
------------------	--

This function injects an error into the HWBIST using the HWBIST registers.

Note

This function should be called after [STL_HWBIST_init\(\)](#) because [STL_HWBIST_init\(\)](#) resets the HWBIST engine and initializes the test.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.5 March13N Test API Functions

Data Structures

- struct [STL_March_InjectErrorObj](#)

Macros

- #define [STL_MARCH_PASS](#)
- #define [STL_MARCH_CORR_ERROR](#)
- #define [STL_MARCH_UNC_ERROR](#)
- #define [STL_MARCH_BOTH_ERROR](#)

Typedefs

- typedef [STL_March_InjectErrorObj](#) * [STL_March_InjectErrorHandle](#)

Enumerations

- enum [STL_March_Pattern](#) { [STL_MARCH_PATTERN_ONE](#), [STL_MARCH_PATTERN_TWO](#), [STL_MARCH_PATTERN_THREE](#), [STL_MARCH_PATTERN_FOUR](#) }

Functions

- void [STL_March_testRAMCopy](#) (const [STL_March_Pattern](#) pattern, const uint32_t startAddress, const uint32_t length, const uint32_t copyAddress)
- void [STL_March_testRAM](#) (const [STL_March_Pattern](#) pattern, const uint32_t startAddress, const uint32_t length)
- void [STL_March_injectError](#) (const [STL_March_InjectErrorHandle](#) errorHandle)
- uint16_t [STL_March_checkErrorStatus](#) (void)

4.5.1 Detailed Description

The code for this module is contained in `source/stl_march.c` and `source/stl_march_s.asm`, with `include/stl_march.h` containing the API declarations for use by applications.

Error Injection

Here are techniques to inject error:

[STL_March_testRAM\(\)](#), [STL_March_testRAMCopy\(\)](#)

- Call [STL_March_injectError\(\)](#) with testMode and xorMask values that cause errors.

4.5.2 Data Structure Documentation

4.5.2.1 struct STL_March_InjectErrorObj

Defines the March memory test inject error object.

Data Fields

uint32_t	address	Address (32-bit aligned)
uint32_t	ramSection	RAM section identifier.
uint32_t	xorMask	Mask to flip bits in test mode.
MemCfg_TestMode	testMode	Mode in which to inject error.

4.5.3 Typedef Documentation

4.5.3.1 typedef **STL_March_InjectErrorObj*** **STL_March_InjectErrorHandle**

Defines the RAM error logic test handle

4.5.4 Enumeration Type Documentation

4.5.4.1 enum **STL_March_Pattern**

Values that must be used to pass to determine the test pattern for [STL_March_testRAMCopy\(\)](#) and [STL_March_testRAM\(\)](#)

Enumerator

STL_MARCH_PATTERN_ONE Test Pattern One.

STL_MARCH_PATTERN_TWO Test Pattern Two.

STL_MARCH_PATTERN_THREE Test Pattern Three.

STL_MARCH_PATTERN_FOUR Test Pattern Four.

4.5.5 Function Documentation

4.5.5.1 void STL_March_testRAMCopy (const **STL_March_Pattern** *pattern*, const uint32_t *startAddress*, const uint32_t *length*, const uint32_t *copyAddress*)

Performs a March13N non-destructive memory test on the specified RAM memory.

Parameters

<i>pattern</i>	is the test pattern to use.
<i>startAddress</i>	is the address to start the memory test.
<i>length</i>	is the number of 32-bit words of the memory test minus 1.
<i>copyAddress</i>	is the address to copy the original contents of the memory under test. It will be used to restore the original memory at the end of the March13N memory test.

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

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

Note

If this code is running from RAM, be careful not to perform this memory test on itself, meaning do not perform the March13N memory test on the March13N program code in RAM. This will likely lead to an ITRAP. In order to test the program code for this March13N algorithm, the user can create a copy of this function in RAM or flash and run the memory test code from the copy.

This function disables global CPU interrupts (DINT) and then re-enables them after the test has completed.

length is the number of 32-bits words to test minus 1. For example, in order to test 8 32-bit words, length is 7.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.5.5.2 void STL_March_testRAM (const **STL_March_Pattern** *pattern*, const uint32_t *startAddress*, const uint32_t *length*)

Performs a March13N destructive memory test on the specified RAM memory.

Parameters

<i>pattern</i>	is the test pattern to use.
<i>startAddress</i>	is the address to start the memory test.
<i>length</i>	is the number of 32-bit words of the memory test minus 1.

This function performs a March13N memory test on RAM specified by **startAddress** and **length**. This test performs a destructive memory test, meaning the original contents will be lost by this test. The test patterns along with the March13N memory test algorithm provided, test memory for stuck-at-faults as well as boundary cases including worst case timings tailored for the C2000 RAM bank architecture.

Note

If this code is running from RAM, be careful not to perform this memory test on itself, meaning do not perform the March13N memory test on the March13N program code in RAM. This will likely lead to an ITRAP. In order to test the program code for this March13N algorithm, the user can create a copy of this function in RAM or flash and run the memory test code from the copy.

This function disables global CPU interrupts (DINT) and then re-enables them after the test has completed.

length is the number of 32-bits words to test minus 1. For example, in order to test 8 32-bit words, length is 7.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.5.5.3 void STL_March_injectError (const **STL_March_InjectErrorHandle** *errorHandle*)

Injects an error into a RAM memory address.

Parameters

<i>errorHandle</i>	the inject error handle specifying where and what type of error to inject into RAM.
--------------------	---

This function injects an error at a specific memory **address** in a specific **ramSection**. **testMode** specifies whether the error will be injected in the data or ECC/parity bits. **xorMask** specifies which bit or bits to flip in order to corrupt either the data or ECC/parity bits.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.5.5.4 uint16_t STL_March_checkErrorStatus (void)

Returns the status of the Memory Error Registers for RAM.

This function checks if there are any correctable or uncorrectable errors indicated by the Memory Error Registers and returns the status.

Returns

If the Memory Error Registers indicate a correctable error and/or an uncorrectable error in RAM, then the function returns **STL_MARCH_CORR_ERROR**, **STL_MARCH_UNC_ERROR**, or **STL_MARCH_BOTH_ERROR**. Otherwise, the function returns **STL_MARCH_PASS**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.6 Oscillator CPU Timer API Functions

Data Structures

- struct [STL_OSC_CT_Obj](#)

Macros

- #define **STL_OSC_CT_FAIL**
- #define **STL_OSC_CT_PASS**
- #define **STL_OSC_CT_PERIOD**

Typedefs

- typedef [STL_OSC_CT_Obj](#) * [STL_OSC_CT_Handle](#)

Functions

- void [STL_OSC_CT_startTest](#) (const [STL_OSC_CT_Handle](#) oscTimer2Handle)
- uint16_t [STL_OSC_CT_stopTest](#) (const [STL_OSC_CT_Handle](#) oscTimer2Handle)

4.6.1 Detailed Description

The code for this module is contained in `source/stl_osc_ct.c`, with `include/stl_osc_ct.h` containing the API declarations for use by applications.

Error Injection

Here are techniques to inject error:

STL_OSC_CT_startTest and [STL_OSC_CT_stopTest\(\)](#)

- Set maxCount and minCount to values so that the Timer 2 counter will be out of range.
- Set the prescaler so that the Timer 2 counter will not be in the minCount/maxCount range.

4.6.2 Data Structure Documentation

4.6.2.1 struct STL_OSC_CT_Obj

Defines the OSC Timer 2 test object.

Data Fields

uint32_t	minCount	Lower bound count.
uint32_t	maxCount	Upper bound count.
CPUTimer_ClockSource	clockSource	Clock source for Timer2.
CPUTimer_Prescaler	prescaler	Prescaler for selected clock source.

4.6.3 Function Documentation

4.6.3.1 void STL_OSC_CT_startTest (const **STL_OSC_CT_Handle** *oscTimer2Handle*)

Starts CPU Timer 2 to test the oscillator source.

Parameters

<i>oscTimer2Handle</i>	is a pointer to the Oscillator Timer 2 object.
------------------------	--

This function disables CPU Timer 2 interrupts, configures the CPU Timer 2 to use the specified clock source and prescaler, starts the timer, and then returns.

The user should configure the PLL and CPU Timer with independent clock sources.

The user must preserve and restore the registers modified by this function. The following CPU Timer 2 and System Control registers are modified by the function.

- TCR
- PRD
- TPRH
- TPR
- TMR2CLKCTL

Note

When **CPUTIMER_CLOCK_SOURCE_SYS** is selected as the **clockSource**, the **prescaler** is bypassed.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.6.3.2 uint16_t STL_OSC_CT_stopTest (const **STL_OSC_CT_Handle** *oscTimer2Handle*)

Stops CPU Timer 2 and checks the elapsed time.

Parameters

<code>oscTimer2Handle</code>	is a pointer to the Oscillator Timer 2 object.
------------------------------	--

This function stops CPU Timer 2 and then compares the number of ticks that have elapsed with the min and max boundaries. This function is intended to be used in combination with [STL_OSC_CT_startTest\(\)](#) to perform a periodic test of the oscillator source.

Returns

If the elapsed number of ticks is not within the min and max boundaries, the function returns **STL_OSC_CT_FAIL**. Otherwise, it returns **STL_OSC_CT_PASS**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.7 Oscillator HRPWM API Functions

Data Structures

- struct [STL_OSC_HR_Obj](#)

Macros

- #define **STL_OSC_HR_PASS**
- #define **STL_OSC_HR_FAIL**
- #define **STL_OSC_HR_SFO_INCOMPLETE**
- #define **STL_OSC_HR_SFO_COMPLETE**
- #define **STL_OSC_HR_SFO_ERROR**

Typedefs

- typedef [STL_OSC_HR_Obj](#) * [STL_OSC_HR_Handle](#)

Functions

- uint16_t [STL_OSC_HR_testSFO](#) (const [STL_OSC_HR_Handle](#) oscHRHandle)

4.7.1 Detailed Description

The code for this module is contained in `source/stl_osc_hr.c`, with `include/stl_osc_hr.h` containing the API declarations for use by applications.

Error Injection

Here are techniques to inject error:

[STL_OSC_HR_testSFO\(\)](#)

- Set mepMax and mepMin the same value so that the MEP scale factor calculated by SFO calibration will be out of range.
- Disable HRPWM clock by clearing HRPWM bit in the PCLKCR0 register.
- Set sfoDelay to 0 to cause timeout.

4.7.2 Data Structure Documentation

4.7.2.1 struct [STL_OSC_HR_Obj](#)

Defines the OSC HR test object.

Data Fields

uint32_t	ePWMBase	EPWM Base.
HRPWM_Channel	channel	HRPWM channel.
HRPWM_MEPEdgeMode	mepEdgeMode	Edge(s) controlled by MEP.
int16_t	mepMin	MEP lower bound.
int16_t	mepMax	MEP upper bound.
uint32_t	sfoDelay	SFO function delay.

4.7.3 Function Documentation

4.7.3.1 uint16_t STL_OSC_HR_testSFO (const **STL_OSC_HR_Handle** *oscHRHandle*)

Tests the HRPWM SFO library's calibration process to ensure execution completion and verify value of MEP scale factor.

Parameters

<i>oscHRHandle</i>	is a pointer to the OSC HR object.
--------------------	------------------------------------

This function runs calibration using the SFO Library Software to calculate an appropriate Micro Edge Positioner (MEP) scale factor for HRPWM-supported ePWM modules to verify this value.

The ePWM module used for the test is selected by *ePWMBase* member of the OSC HR test object. The valid range of inputs is from EPWM1_BASE to EPWMx_BASE where x is the value specified by PWM_CH - 1. PWM_CH is defined in the SFO library header file SFO_V8.h. The channel member can be configured to select between channels A and B. The *mepEdgeMode* value is configured for the PWM edges to be controlled by MEP. The values *mepMin* and *mepMax* define the minimum and maximum bounds of the expected MEP scale factor range. The expected MEP scale factor for EPWMCLK of 100 MHz, given a step size of 150 ps, is about 66 MEP steps. $(1/\text{EPWMCLK})/\text{Step size} = (1/100 \text{ MHz})/150 \text{ ps}$ is about 66.

The *sfoDelay* value is the delay count required for the SFO() to finish calibration. The typical EPWM cycles required for SFO() to complete calibration if called repeatedly without interrupts is 130, 000 EPWMCLK cycles. Please refer to **Appendix A: SFO Library Software** of the Technical Reference Manual HRPWM chapter and the device datasheet for more details. The repetition rate at which SFO() needs to be executed depends on the application. If there is not a sufficiently large interval between SFO() calls, the SFO() will return and not advance to the next stage. The execution will only advance to the next stage if the previous call's execution has completed and SFO() is called again. This interval can be experimentally calculated for a particular application.

Note

The SFO library binary is provided in the C2000Ware calibration libraries. The source code however is not publicly released. It may be made available in some cases upon request through an FAE.

Returns

If the MEP scale factor calculated by the calibration falls within the MEP min and max values, the function returns **STL_OSC_HR_PASS**. If the calibration fails to complete in the specified delay, if SFO() returns an error status, or if the calculated MEP scale factor is outside the range, this function returns **STL_OSC_HR_FAIL**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.8 PIE RAM API Functions

Macros

- #define **STL_PIE_RAM_PASS**
- #define **STL_PIE_RAM_FAIL_HANDLER**
- #define **STL_PIE_RAM_MIN_INDEX**
- #define **STL_PIE_RAM_MAX_INDEX**
- #define **STL_PIE_RAM_REDUNDANT_PIE_ADDRESS**
- #define **STL_PIE_RAM_TABLE_ROW_M**
- #define **STL_PIE_RAM_TABLE_COL_M**
- #define **STL_PIE_RAM_TABLE_COL_S**
- #define **STL_PIE_RAM_VECT_ID_M**
- #define **STL_PIE_RAM_VECT_ID_S**

Functions

- void [STL_PIE_RAM_handler](#) (void)
- void [STL_PIE_RAM_configHandler](#) (const void *handlerPtr)
- void [STL_PIE_RAM_injectFault](#) (uint16_t entry)
- void [STL_PIE_RAM_restoreVector](#) (uint16_t entry)
- void [STL_PIE_RAM_restoreTable](#) (const uint32_t *pieTableSourcePtr)
- uint16_t [STL_PIE_RAM_testRAM](#) (void)
- uint16_t [STL_PIE_RAM_testHandler](#) (uint32_t interruptNumber)

4.8.1 Detailed Description

The code for this module is contained in `source/stl_pie_ram.c`, with `include/stl_pie_ram.h` containing the API declarations for use by applications.

Error Injection

Here are techniques to inject error:

[STL_PIE_RAM_testRAM\(\)](#)

- Call [STL_PIE_RAM_injectFault\(\)](#) to bit-flip an entry of the redundant PIE RAM vector table.
- [STL_PIE_RAM_testHandler\(\)](#)
- Disable global interrupts with DINT macro.

4.8.2 Function Documentation

4.8.2.1 void [STL_PIE_RAM_handler](#) (void)

PIE RAM mismatch error handler used by [STL_PIE_RAM_testHandler\(\)](#).

This function sets a static global flag for [STL_PIE_RAM_testHandler\(\)](#) and also sets the global error flag for the STL library.

Returns

None.

4.8.2.2 void STL_PIE_RAM_configHandler (const void * *handlerPtr*)

Configures a vector mismatch handler.

Parameters

<i>handlerPtr</i>	is a pointer or address of the PIE RAM vector mismatch error exception handler.
-------------------	---

This function configures a PIE RAM mismatch error handler for the users application. On a vector fetch mismatch, an exception will be taken and the error handler will be executed.

Returns

None.

4.8.2.3 void STL_PIE_RAM_injectFault (uint16_t *entry*)

Injects a fault by bit-flipping an entry in the redundant PIE RAM vector table.

Parameters

<i>entry</i>	is an index of the PIE RAM and should be a multiple of 2 for 32-bit alignment.
--------------	--

This function bit-flips an entry of the redundant PIE RAM vector table in order to inject a fault. The **entry** parameter must be within the boundaries **STL_PIE_RAM_MIN_INDEX** and **STL_PIE_RAM_MAX_INDEX**.

Data writes to the redundant PIE vector table write only to the redundant table. This is how a fault may be injected. For more details see the device's Technical Reference Manual chapter on "Vector Address Validity Check."

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).4.8.2.4 void STL_PIE_RAM_restoreVector (uint16_t *entry*)

Restores a PIE RAM redundant vector entry from the PIE RAM vector table.

Parameters

<i>entry</i>	is an index of the PIE RAM and should be a multiple of 2 for 32-bit alignment.
--------------	--

This function restores the PIE RAM vector entry and consequently the redundant PIE RAM vector table.

Note

This function can be used to restore the fault injected by [STL_PIE_RAM_injectFault\(\)](#).

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).**4.8.2.5 void STL_PIE_RAM_restoreTable (const uint32_t * *pieTableSourcePtr*)**

Restores the PIE RAM from the initialization source.

Parameters

<i>pieTableSourcePtr</i>	is a pointer to the source of the PIE vector table in the user's application.
--------------------------	---

This function restores the PIE RAM vector table and consequently the redundant PIE RAM vector table.

Note

This function restores the entire PIE RAM table and not just a single vector entry.

Returns

None.

4.8.2.6 uint16_t STL_PIE_RAM_testRAM (void)

Tests PIE RAM integrity.

This function checks the integrity of PIE RAM vector by comparing the entries in the main PIE vector with the redundant PIE vector located at a higher memory address location. There is a redundant PIE RAM vector table starting at 0X1000D00 which is used for data integrity of the PIE. Data writes to the PIE vector table also write to the redundant PIE vector table. For more details see the device's Technical Reference Manual chapter on "PIE Vector Address Validity Check."

Note

The user may call [STL_PIE_RAM_configHandler\(\)](#) in order to register an error handler for PIE RAM mismatch.

A failure of this test will not trigger the PIE RAM mismatch handler. Only when a vector is fetched in the PIE and there is a mismatch will the handler be serviced.

Returns

If the test passes, it returns **STL_PIE_RAM_TEST_PASS**. If the test fails, it returns the address in the PIE vector table of the last failure.

Referenced by [STA_Tests_testDevice\(\)](#).**4.8.2.7 uint16_t STL_PIE_RAM_testHandler (uint32_t *interruptNumber*)**

Tests PIE RAM integrity.

Parameters

<i>interruptNumber</i>	is the 32-bit interrupt value used in the interrupt driver found in hw_ints.h.
------------------------	--

This function checks the functionality of the PIE RAM mismatch error handler. It will inject an error for the input interrupt, enable the interrupt, and force the interrupt. It will then check to see that the error handler was serviced properly. Afterward, it will restore the PIE RAM and and restore the original error handler. It will also acknowledge the PIE so that further interrupts from that group can be serviced.

Note

This test only covers PIE interrupts. This test does not cover CPU interrupts.

Returns

If the test passes, it returns **STL_PIE_RAM_PASS**. If the test fails, and the error handler was not serviced, then it returns **STL_PIE_RAM_FAIL_HANDLER**.

Referenced by [STA_Tests_testDevice\(\)](#).

4.9 Utilities API Functions

Macros

- #define **STL_UTIL_CPU_RATE**

Enumerations

- enum **STL_Util_ErrorFlag** {
[STL_UTIL_CRC](#), [STL_UTIL_CPU_REG](#), [STL_UTIL_FPU_REG](#), [STL_UTIL_VCRC_REG](#),
[STL_UTIL_PIE_RAM_MISMATCH](#), [STL_UTIL_PIE_RAM_INT](#), [STL_UTIL_OSC_TIMER2](#),
[STL_UTIL_OSC_HR_MEP_RANGE](#),
[STL_UTIL_OSC_HR_SFO](#), [STL_UTIL_OSC_HR_DELAY](#), [STL_UTIL_MARCH](#),
[STL_UTIL_CAN_RAM_PARITY](#),
[STL_UTIL_HWBIST_NMI_TEST](#), [STL_UTIL_HWBIST_FAIL](#), [STL_UTIL_HWBIST_NMI_INT](#),
[STL_UTIL_HWBIST_OVERRUN](#) }

Functions

- static void [STL_Util_delayUS](#) (uint32_t microseconds)
- void [STL_Util_setErrorFlag](#) (const [STL_Util_ErrorFlag](#) errorFlag)
- uint32_t [STL_Util_getErrorFlag](#) (void)
- void [STL_Util_clearErrorFlag](#) (const [STL_Util_ErrorFlag](#) errorFlag)

4.9.1 Detailed Description

The code for this module is contained in `source/stl_util.c` and `source/stl_util_s.asm`, with `include/stl_util.h` containing the API declarations for use by applications.

4.9.2 Enumeration Type Documentation

4.9.2.1 enum **STL_Util_ErrorFlag**

Values to be passed to [STL_Util_setErrorFlag\(\)](#) and [STL_Util_clearErrorFlag\(\)](#). These correspond to different errors in the STL.

Enumerator

- STL_UTIL_CRC** CRC check.
- STL_UTIL_CPU_REG** CPU register test.
- STL_UTIL_FPU_REG** FPU register test.
- STL_UTIL_VCRC_REG** VCRC register test.
- STL_UTIL_PIE_RAM_MISMATCH** PIE RAM.
- STL_UTIL_PIE_RAM_INT** PIE RAM handler failed to execute.
- STL_UTIL_OSC_TIMER2** OSC timer 2.
- STL_UTIL_OSC_HR_MEP_RANGE** MEP out of range.

STL_UTIL_OSC_HR_SFO SFO calibration error.
STL_UTIL_OSC_HR_DELAY SFO delay error.
STL_UTIL_MARCH March RAM test.
STL_UTIL_CAN_RAM_PARITY March test for CAN message RAM.
STL_UTIL_HWBIST_NMI_TEST HWBIST NMI test.
STL_UTIL_HWBIST_FAIL HWBIST fail.
STL_UTIL_HWBIST_NMI_INT HWBIST NMI interrupt.
STL_UTIL_HWBIST_OVERRUN HWBIST over-run.

4.9.3 Function Documentation

4.9.3.1 static void STL_Util_delayUS (uint32_t *microseconds*) [inline], [static]

Delay for a specified number of microseconds

Parameters

<i>microseconds</i>	is the number of microseconds to delay.
---------------------	---

This function calls SysCtl_delay() to achieve a delay in microseconds. The function will convert the desired delay in microseconds to the count value expected by the function. *microseconds* is the number of microseconds to delay.

Note

If this function does not get inlined (for instance, if the optimizer is turned off) the delay will be made less accurate by the overhead of the additional function call.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

4.9.3.2 void STL_Util_setErrorFlag (const **STL_Util_ErrorFlag** *errorFlag*)

Sets a global error flag.

Parameters

<i>errorFlag</i>	is a STL_Util_ErrorFlag that will be set in globalErrorFlags.
------------------	---

This function sets an error flag in the global globalErrorFlags.

Returns

None.

Referenced by [STL_CPU_REG_checkCPURegisters\(\)](#), [STL_CPU_REG_checkFPURegisters\(\)](#), and [STL_CPU_REG_checkVCRCRegisters\(\)](#).

4.9.3.3 `uint32_t STL_Util_getErrorFlag (void)`

Gets the error flag status of globalErrorFlags.

This function returns the global globalErrorFlags.

Returns

Returns the global globalErrorFlags.

Referenced by [STA_Tests_testDevice\(\)](#).

4.9.3.4 `void STL_Util_clearErrorFlag (const STL_Util_ErrorFlag errorFlag)`

Clears a flag of globalErrorFlags.

Parameters

<i>errorFlag</i>	is a STL_Util_ErrorFlag that will be cleared in globalErrorFlags.
------------------	---

This function clears a flag of globalErrorFlags.

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

5 Self-Test Application

This section will describe the Self-Test Application (or STA) example project delivered with the diagnostic library. The STA is an example which provides a demonstration of the SDL modules and diagnostic functions.

The STA is designed to act as an example application which configures the system and calls the SDL APIs to perform the safety diagnostic tests made available in the diagnostic library. The STA is not intended to limit the customer to any implementation of safety diagnostic functions. Instead, it is intended to aid integration of the SDL into the user's end application or system by providing a demonstration of how to use the library. It is the responsibility of the system integrator to meet any and all safety compliance standards in their end application.

The STA has been tested on the controlCARD and docking station.

Getting Started	44
Example Behavior	45
STA_Tests API Functions	47
STA_Timer API Functions	49
STA_Comm API Functions	51
STA_Util API Functions	52
STA_User API Functions	54

5.1 Getting Started

The Code Composer Studio project containing the STA can be imported into CCS by going to **Project -> Import CCS Projects...** and browsing to the search directory below.

```
"diagnostic\<device>\examples\test_application"
```

Right click on the newly imported project in the Project Explorer pane and select **Build Project**. This will build the default RAM configuration of the project. The project provides several other build configurations for different levels of optimization and running the application from Flash, but the RAM configuration is a good starting point for getting familiar with the STA.

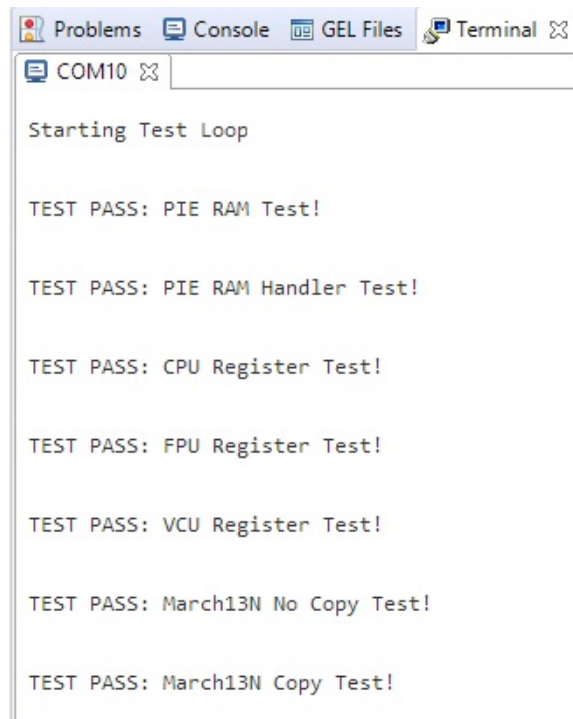
Connect and power your controlCARD according to the controlCARD info sheet in directory

```
"<C2000Ware_install>\boards\controlCARDs\"
```

The STA uses the SCIA module to communicate test progress and results through the controlCARD's USB-to-UART adapter. Before running the application, set up a terminal session with 9600 baud rate, 8-N-1. Note that CCS contains a terminal window that can be accessed by going to **View -> Terminal**.

With the test application project selected in the Project Explorer, go to **Run -> Debug**. This will launch a debug session using the .ccxml to which the project is linked. By default, the XDS100v2 is used as this is the on-board debug probe for the controlCARD.

Run the application. In the terminal, you should see a looping message indicating test cases passing.

A screenshot of a terminal window within an IDE. The terminal has tabs for 'Problems', 'Console', 'GEL Files', and 'Terminal'. The 'Terminal' tab is active, showing output from a COM10 port. The output text is as follows:

```
Starting Test Loop

TEST PASS: PIE RAM Test!

TEST PASS: PIE RAM Handler Test!

TEST PASS: CPU Register Test!

TEST PASS: FPU Register Test!

TEST PASS: VCU Register Test!

TEST PASS: March13N No Copy Test!

TEST PASS: March13N Copy Test!
```

Figure 5.1: Passing Terminal Output

5.2 Example Behavior

Most tests will display a PASS message in the terminal when they execute successfully with error injection disabled and display a FAIL message when they execute successfully when error injection is enabled. To toggle error injection, add the *enableErrorInject* variable to the CCS Expressions view. Setting its value to 1 will enable error injection. Setting it back to 0 will disable it.

The STA has built in profiling code for measuring cycle counts of the library functions as they are used by the STA. This functionality is turned on by default and uses CPU Timer 1 to capture time elapsed and records the result to an array called *STA_Tests_cycleCounts* which can be viewed in the CCS Expressions view.

The profiling can be turned off by changing the value of macro **STA_UTIL_PROFILE** to 0 and recompiling. More details about the functions used for profiling can be found in the following sections.

Note that it is not possible to inject an error for the all tests. In particular, the **STA_MARCH** and **STA_MARCH_CAN** tests are expected to return PASS even when *enableErrorInject* is set. The core API used by these tests is [STL_March_testRAM\(\)](#). This function performs the March13n algorithm on the RAM memory addresses specified without a copy and restore of their original contents unlike **STA_MARCH_COPY** and **STA_MARCH_CAN_COPY**. Since the first operation performed on the memory as part of the algorithm is a write which triggers a recalculation of the parity or ECC, any error injected prior to running the test will be overwritten before it can be detected.

Also, note that executing HWBIST can terminate the debugger connection. Therefore, the STA

provides two macros, `STA_UTIL_HWBIST_MICRO` and `STA_UTIL_HWBIST_FULL`, to include or exclude the HWBIST micro-run test and the full test respectively.

When error injection is enabled, the STA will iterate through the five error injection techniques for HWBIST. Additionally, since the HWBIST may terminate the debugger connection and access to debugger information, the STA writes some debugging information to global arrays. These global arrays are `STA_Tests_memDumpMicro` and `STA_Tests_memDumpFull`. The information includes:

1. The error injected using enumerated type `STL_HWBIST_Error`
2. Information whether or not the previous NMI vector was restored
3. The return value from the test function
4. The status of the global error flags

The global arrays are large enough to hold the information from 5 runs of the HWBIST (20 by 32-bits)—that is one for each of the 5 different types of tests executed by the STA. Some of these tests will return a passing value although an error was injected. This is due to the internal HWBIST behavior and is detailed further in a C2000 HWBIST application note (SPRACA7).

5.3 STA_Tests API Functions

Macros

- #define **STA_TESTS_PASS**
- #define **STA_TESTS_FAIL**

Enumerations

- enum **STA_TestsTypes** {
STA_TEST_START, **STA_HWBIST_MICRO**, **STA_HWBIST_FULL**, **STA_TEST_PIE_RAM**,
STA_TEST_PIE_HANDLER, **STA_CPU_REG**, **STA_FPU_REG**, **STA_VCRC_REG**,
STA_MARCH, **STA_MARCH_COPY**, **STA_MARCH_CAN**, **STA_MARCH_CAN_COPY**,
STA_CAN_RAM_PARITY, **STA_FLASH_CRC**, **STA_OSC_CT**, **STA_OSC_HR**,
STA_TEST_END, **STA_TESTS_NUMBERS** }

Functions

- unsigned char * [STA_Tests_testDevice](#) (STA_TestsTypes testItem)
- void [STA_Tests_injectErrorEnable](#) (void)
- void [STA_Tests_injectErrorDisable](#) (void)

Variables

- const STA_TestsTypes **STA_Tests_testArray** [STA_TESTS_NUMBERS]
- uint32_t **STA_Tests_cycleCounts** [STA_TESTS_NUMBERS]

5.3.1 Detailed Description

This module contains the core test execution function for this application. It defines the list of test cases to be executed and their implementations.

The code for this module is contained in `sta_tests.c`, with `sta_tests.h` containing the API declarations.

5.3.2 Function Documentation

5.3.2.1 unsigned char* STA_Tests_testDevice (STA_TestsTypes *testItem*)

Executes the safety library tests as an example.

Parameters

<i>testItem</i>	is the enumerated type <code>STA_test_types</code> test to perform.
-----------------	---

This function configures the device for each specific test. If injected errors are enabled, then this function will inject an error for the test being performed. It will then execute each STL test and return a message of characters to be printed to the terminal via SCI.

Returns

Returns a pointer to a message string indicating a pass or fail status and the test name. This string will be printed out of SCIA to the user.

References `STL_March_InjectErrorObj::address`, `STL_OSC_CT_Obj::maxCount`, `STL_OSC_HR_Obj::mepMin`, `STL_OSC_CT_Obj::minCount`, `STA_User_canParityErrorISR()`, `STA_User_initMarch()`, `STA_User_initOSCHRTTest()`, `STA_User_initOSCTimer2Test()`, `STA_Util_startProfiler()`, `STA_Util_stopProfiler()`, `STL_CAN_RAM_checkErrorStatus()`, `STL_CAN_RAM_injectError()`, `STL_CAN_RAM_NO_COPY`, `STL_CAN_RAM_testRAM()`, `STL_CPU_REG_checkCPURegisters()`, `STL_CPU_REG_checkFPURegisters()`, `STL_CPU_REG_checkVCRCRegisters()`, `STL_CRC_checkCRC()`, `STL_HWBIST_90_LOS`, `STL_HWBIST_FINAL_COMPARE`, `STL_HWBIST_init()`, `STL_HWBIST_injectError()`, `STL_HWBIST_LOGIC_FAULT`, `STL_HWBIST_NMI_TRAP`, `STL_HWBIST_NO_ERROR`, `STL_HWBIST_runFull()`, `STL_HWBIST_runMicro()`, `STL_HWBIST_TIMEOUT`, `STL_March_checkErrorStatus()`, `STL_March_injectError()`, `STL_MARCH_PATTERN_FOUR`, `STL_MARCH_PATTERN_ONE`, `STL_MARCH_PATTERN_THREE`, `STL_MARCH_PATTERN_TWO`, `STL_March_testRAM()`, `STL_March_testRAMCopy()`, `STL_OSC_CT_startTest()`, `STL_OSC_CT_stopTest()`, `STL_OSC_HR_testSFO()`, `STL_PIE_RAM_injectFault()`, `STL_PIE_RAM_restoreVector()`, `STL_PIE_RAM_testHandler()`, `STL_PIE_RAM_testRAM()`, `STL_Util_clearErrorFlag()`, `STL_Util_delayUS()`, `STL_Util_getErrorFlag()`, `STL_UTIL_HWBIST_FAIL`, `STL_UTIL_HWBIST_NMI_INT`, `STL_UTIL_HWBIST_NMI_TEST`, `STL_UTIL_HWBIST_OVERRUN`, `STL_March_InjectErrorObj::testMode`, and `STL_March_InjectErrorObj::xorMask`.

5.3.2.2 void STA_Tests_injectErrorEnable (void)

Enables errors to be injected into the tests.

This function sets the global variable **STA_Tests_injectError** to true.

Returns

None.

5.3.2.3 void STA_Tests_injectErrorDisable (void)

Disables errors to be injected into the tests.

This function sets the global variable **STA_Tests_injectError** to false.

Returns

None.

5.4 STA_Timer API Functions

Functions

- __interrupt void [STA_Timer_timer0Isr](#) (void)
- void [STA_Timer_config](#) (uint16_t msTimeOut)
- bool [STA_Timer_isTimedOut](#) (void)
- void [STA_Timer_clearTimeOut](#) (void)
- void [STA_Timer_restart](#) (void)

5.4.1 Detailed Description

This module handles the initialization and handling of the CPU timer system used to time the execution of the tests run by this application.

The code for this module is contained in `sta_timer.c`, with `sta_timer.h` containing the API declarations.

5.4.2 Function Documentation

5.4.2.1 __interrupt void STA_Timer_timer0Isr (void)

Timer 0 ISR.

Referenced by [STA_Timer_config\(\)](#).

5.4.2.2 void STA_Timer_config (uint16_t *msTimeOut*)

Configures Timer 0 to tick so that the STA will continue to make progress.

Parameters

<i>msTimeOut</i>	is the number of milliseconds between each execution of a test found in <code>sta_tests.c</code> . Another test will execute every time this STA timer runs out.
------------------	--

Returns

None.

References [STA_Timer_timer0Isr\(\)](#).

5.4.2.3 bool STA_Timer_isTimedOut (void)

Checks if Timer 0 has timed-out.

Returns

Returns true if the timer has timed-out and false if it has not.

5.4.2.4 void STA_Timer_clearTimeOut (void)

Clears Timer 0 time-out flag.

Returns

None.

5.4.2.5 void STA_Timer_restart (void)

Restarts Timer 0.

Returns

None.

5.5 STA_Comm API Functions

Macros

- #define **STA_COMM_BAUD_RATE**

Functions

- void [STA_Comm_configSCIA](#) (void)
- void [STA_Comm_transmitData](#) (unsigned char *msg)

5.5.1 Detailed Description

This module handles the configuration of the SCIA module for communication over the board USB-to-UART adapter. This path will allow the application to send test pass/fail status to a terminal program.

The code for this module is contained in `sta_comm.c`, with `sta_comm.h` containing the API declarations.

5.5.2 Function Documentation

5.5.2.1 void STA_Comm_configSCIA (void)

Configures SCIA for serial communication.

This function configures SCIA for communication to a serial communication terminal for 9600 baud rate, 8-N-1.

Returns

None.

5.5.2.2 void STA_Comm_transmitData (unsigned char * msg)

Transmits data via SCIA.

Parameters

<i>msg</i>	is a \0 terminated array of characters
------------	--

This function transmits the **msg** via SCIA to a serial communication terminal.

Returns

None.

5.6 STA_Util API Functions

Macros

- #define [STA_UTIL_PROFILE](#)

Functions

- void [STA_Util_configProfiler](#) (uint32_t base)
- static void [STA_Util_startProfiler](#) (uint32_t base)
- static uint32_t [STA_Util_stopProfiler](#) (uint32_t base)

5.6.1 Detailed Description

This module contains utility functions for this test application. This primarily includes the functions used to profile the cycle counts of the SDL functions.

The code for this module is contained in `sta_util.c`, with `sta_util.h` containing the API declarations.

5.6.2 Macro Definition Documentation

5.6.2.1 #define STA_UTIL_PROFILE

Used to enable or disable CPUTimer cycle profiling of STA tests. A value of 0 will disable cycle profiling. A value of not 0 will enable cycle profiling.

5.6.3 Function Documentation

5.6.3.1 void STA_Util_configProfiler (uint32_t *base*)

Configures the CPUTimer specified by the parameter **base** for profiling cycle counts.

Parameters

<i>base</i>	is the base address of a valid CPUTimer.
-------------	--

Returns

None.

5.6.3.2 static void STA_Util_startProfiler (uint32_t *base*) [inline], [static]

Starts the CPUTimer used for profiling cycles.

Parameters

<i>base</i>	is the base address of a valid CPUTimer.
-------------	--

Returns

None.

Referenced by [STA_Tests_testDevice\(\)](#).

5.6.3.3 `static uint32_t STA_Util_stopProfiler (uint32_t base) [inline], [static]`

Stops the CPUTimer used for profiling cycles.

Parameters

<i>base</i>	is the base address of a valid CPUTimer.
-------------	--

Returns

Returns the number of cycles elapsed since [STA_Util_startProfiler\(\)](#) was called.

Referenced by [STA_Tests_testDevice\(\)](#).

5.7 STA_User API Functions

Macros

- #define **STA_USER_MARCH_DATA_SIZE**
- #define **STA_USER_MARCH_CAN_COPY_SIZE**
- #define **STA_USER_MARCH_CANA_OBJ32_START_ADDR**
- #define **STA_USER_MARCH_CANA_OBJ4_END_ADDR**
- #define **STA_USER_MARCH_CANA_OBJ6_START_ADDR**
- #define **STA_USER_MARCH_CANA_OBJ15_END_ADDR**
- #define **STA_USER_PIE_RAM_ENTRY**
- #define **STA_USER_PIE_TEST_INT**
- #define **STA_USER_PIE_TEST_INT_GROUP_M**
- #define **STA_USER_DCSM_OTP_GOLDEN_CRC**
- #define **STA_USER_CAN_RAM_GOLDEN_CRC**
- #define **STA_USER_OSC_DELAY_US**
- #define **STA_USER_OSC_MIN_COUNT**
- #define **STA_USER_OSC_MAX_COUNT**
- #define **STA_USER_OSC_HR_MEP**
- #define **STA_USER_OSC_HR_MEP_MIN**
- #define **STA_USER_OSC_HR_MEP_MAX**

Functions

- void [STA_User_initMarch](#) (void)
- void [STA_User_initOSCTimer2Test](#) (void)
- void [STA_User_initOSCHRTTest](#) (void)
- __interrupt void [STA_User_canParityErrorISR](#) (void)

Variables

- [STL_March_InjectErrorObj](#) [STA_User_marchErrorObj](#)
- [STL_March_InjectErrorHandle](#) [STA_User_marchErrorHandle](#)
- uint16_t [STA_User_marchTestData](#) [STA_USER_MARCH_DATA_SIZE]
- uint16_t [STA_User_marchTestDataCopy](#) [STA_USER_MARCH_DATA_SIZE]
- uint16_t [STA_User_marchTestDataCANCopy](#) [STA_USER_MARCH_CAN_COPY_SIZE]
- bool [STA_User_canInterruptFlag](#)
- uint16_t [STA_User_parityErrorCode](#)
- uint16_t [STA_User_canErrorReturnVal](#)
- [STL_OSC_CT_Obj](#) [STA_User_oscTimer2Obj](#)
- [STL_OSC_CT_Handle](#) [STA_User_oscTimer2Handle](#)
- [STL_OSC_HR_Obj](#) [STA_User_oscHRObj](#)
- [STL_OSC_HR_Handle](#) [STA_User_oscHRHandle](#)

5.7.1 Detailed Description

This module contains the declaration and initialization of the various objects and handles required by SDL modules tested in this application.

The code for this module is contained in `sta_user.c`, with `sta_user.h` containing the API declarations.

5.7.2 Function Documentation

5.7.2.1 void STA_User_initMarch (void)

This function initializes the inject error object for March13N test.

Returns

None

References [STL_March_InjectErrorObj::address](#), [STL_March_InjectErrorObj::ramSection](#), [STL_March_InjectErrorObj::testMode](#), and [STL_March_InjectErrorObj::xorMask](#).

Referenced by [STA_Tests_testDevice\(\)](#).

5.7.2.2 void STA_User_initOSCTimer2Test (void)

This function initializes the Oscillator Timer2 test objects.

Returns

None

References [STL_OSC_CT_Obj::clockSource](#), [STL_OSC_CT_Obj::maxCount](#), [STL_OSC_CT_Obj::minCount](#), and [STL_OSC_CT_Obj::prescaler](#).

Referenced by [STA_Tests_testDevice\(\)](#).

5.7.2.3 void STA_User_initOSCHRTTest (void)

This function initializes the object for the Oscillator HRPWM test.

Returns

None

References [STL_OSC_HR_Obj::channel](#), [STL_OSC_HR_Obj::ePWMBase](#), [STL_OSC_HR_Obj::mepEdgeMode](#), [STL_OSC_HR_Obj::mepMax](#), [STL_OSC_HR_Obj::mepMin](#), and [STL_OSC_HR_Obj::sfoDelay](#).

Referenced by [STA_Tests_testDevice\(\)](#).

5.7.2.4 __interrupt void STA_User_canParityErrorISR (void)

CAN ISR to execute upon parity error detection in CAN message RAM.

Returns

None

References [STL_CAN_RAM_checkErrorStatus\(\)](#).

Referenced by [STA_Tests_testDevice\(\)](#).

6 Integration Notes

Profiling data is provided below for evaluation. These numbers were taken using the compiler settings and version for the RELEASE build configuration of the library project and the RAM_RELEASE and FLASH_RELEASE configurations of the STA.

Cycle Time	56
Memory Usage	57

6.1 Cycle Time

The cycle counts shown are based on the configuration of the modules used in the STA. The counts are not necessarily the worst-case cycle count and may vary depending on the module parameters. For example, some of the memory tests have configurable memory ranges over which the test should be performed, meaning the larger the memory section, the longer the test runs.

API	PASS Cycles (RAM)	FAIL Cycles (RAM)	PASS Cycles (Flash)	FAIL Cycles (Flash)
STL_HWBIST_runMicro	440	502	413	498
STL_HWBIST_runFull	305588	305588	282344	282344
STL_PIE_RAM_testRAM	5323	5328	7763	7766
STL_PIE_RAM_testHandler	263	170	299	196
STL_CPU_REG_checkCPURegisters	655	56	713	86
STL_CPU_REG_checkFPURegisters	288	122	350	163
STL_CPU_REG_checkVCRCRegisters	230	119	272	153
STL_March_testRAM	405	405	657	657
STL_March_testRAMCopy	503	503	808	808
STL_CAN_RAM_testRAM	6660	6660	8574	8874
STL_CAN_RAM_testRAM (with copy)	16234	16333	21595	21722
STL_CRC_checkCRC	500	503	279	284
STL_OSC_CT_startTest + STL_OSC_CT_stopTest	148	144	182	178
STL_OSC_HR_testSFO	132545	132546	132914	132926

6.2 Memory Usage

The following shows the size in 16-bit words of the STL APIs. These values were obtained from the STA .map file.

API	Memory (16-bit words)
STL_CAN_RAM_checkErrorStatus	14
STL_CAN_RAM_injectError	39
STL_CAN_RAM_testRAM	62
STL_CPU_REG_testCPURegisters	205
STL_CPU_REG_testFPURegisters	106
STL_CPU_REG_testVCRCRegisters	76
STL_CRC_checkCRC	40
STL_CRC_calculate	56
STL_CRC_calculateLowBytes	46
STL_CRC_reset	5
STL_HWBIST_errorNMIISR	55
STL_HWBIST_init	55
STL_HWBIST_handleReset	115
STL_HWBIST_runFull	79
STL_HWBIST_runMicro	40
STL_HWBIST_runMicroTest	126
STL_HWBIST_restoreContext	15
STL_March_checkErrorStatus	38
STL_March_injectError	16
STL_March_testRAM	35
STL_March_testRAMCopy	47
STL_OSC_CT_startTest	61
STL_OSC_CT_stopTest	29
STL_OSC_HR_testSFO	120
STL_PIE_RAM_handler	5
STL_PIE_RAM_injectFault	18
STL_PIE_RAM_restoreVector	10
STL_PIE_RAM_testHandler	101
STL_PIE_RAM_testRAM	32
STL_Util_setErrorFlag	6

There is also a contribution of 134 words from the driverlib functions used by the library. Use of the STL_OSC_HR module results in an additional 497 words from the SFO library.

7 Safety Feature and Diagnostic Examples

Many of the safety features and diagnostics documented in the device safety manual are very system-specific and not well suited to being implemented as SDL APIs. However many still require special procedures or hardware test modes that may require some demonstration beyond their documented descriptions. For these particular cases, a series of examples have been created as part of the SDL to provide such a demonstration.

Using DCSM for Freedom From Interference (FFI)	58
Test of ECC logic in Flash	59
Test of Flash Prefetch, Data Cache and Wait-States	59
Missing clock detection	60
RAM access protection violation detection	60
Test of parity/ECC logic in SRAM	61
Software test of watchdog operation	62

7.1 Using DCSM for Freedom From Interference (FFI)

sdl_ex_dcsm_ffl

This example demonstrates how to configure and use DCSM and how the DCSM blocks writes from one secured zone to another, simulating the its use to create firewalls between code of different safety requirement criticality.

It uses the default passwords to allocate one LSRAM block and one flash sector to zone 1 and another to zone 2. In this example, zoning of memories is done by programming the DCSM OTP. The values that are programmed can be found in in dcsm.asm. Secured RAM blocks are partitioned to contain data arrays used in the example, and secured Flash sectors contain the program code. Once secure, both zone1LockedArray and zone2LockedArray are immutable outside of the safety functions residing in the same zone.

!!!IMPORTANT!! By default, assignments in dcsm.asm are commented out and the sections mapped to OTP in dcsm.cmd are given the "type = DSECT" attribute to prevent programming of the OTP. In this state, this example is expected to fail (result = FAIL) because the memories won't actually be secured. If you are sure you want to permanently update the zones' link pointers and zone select blocks, uncomment the code in dcsm.asm and remove "type = DSECT" from dcsm.cmd.

Parts of this example where generated with assistance from the DCSM Tool. We strongly recommend using this tool to generate the code for your DCSM configuration. Refer to the following guide for more information: <http://www.ti.com/lit/pdf/spracp8>

Note

This example assumes you have not written to the DCSM OTP before and that the first zone select block is available for use. If this is not the case, you will need to update the link pointer, zone select block address, and passwords to use the next available zone select block.

External Connections

- None.

Watch Variables

- **result** - Status indicating success of blocking writes to secured zones.
- **errorZone1NotChanged**, **errorZone2NotChanged**, **errorZone1Changed**, **errorZone2Changed** - Count of errors found during execution.
- **zone1LockedArray** - Array located in zone 1 secured memory.
- **zone2LockedArray** - Array located in zone 2 secured memory.

7.2 Test of ECC logic in Flash

`sdl_ex_flash_ecc_test`

This example demonstrates how to test the Flash ECC logic functionality.

A software test of the Flash ECC logic can be performed with the help of ECC test registers. Using the test registers, you can generate both single bit errors and uncorrectable errors. For additional details on the implementation of this diagnostic, see the "SECEDED Logic Correctness Check" section in the device technical reference manual.

External Connections

- None.

Watch Variables

- **nmiISRFlag** - Indicates that the NMI was triggered and called the ISR.
- **nmiStatus** - NMI status flags read in the ISR.
- **errorISRFlag** - Indicates that the correctable ECC error interrupt was triggered and called the ISR.
- **errorStatus** - Error status flags read in the ISR, indicating whether an error was single-bit or uncorrectable.
- **errorCount** - Number of correctable errors detected read in the ISR.
- **errorType** - For a single-bit error, indicates whether it was in the data or ECC bits.
- **dataOut** - For a single-bit error, displays the corrected data.
- **result** - Status of a successful detection and handling of ECC errors.

7.3 Test of Flash Prefetch, Data Cache and Wait-States

`sdl_ex_flash_prefetch_test`

This example demonstrates how to test the behavior of the flash prefetch, data cache, and wait-states. Each of these configurations are altered one by one and the impact they have on a test function's execution time is measured. It is expected that an increase in the execution time will be observed in each case.

Note that as mentioned in the safety manual description of this diagnostic, a golden value may be used to check the execution time. This wasn't used for this example because of possible compiler differences between users' environments, but can be used in your actual application.

External Connections

- None.

Watch Variables

- **baseCycles** - Execution time of runPrefetchTestCode();
- **failCount** - Number of tests that returned failures.
- **result** - Status of flash configuration logic test.

7.4 Missing clock detection

sdl_ex_mcd_test

This example demonstrates how to check the missing clock detection functionality.

Once the MCD is simulated by disconnecting the clock to the MCD module an NMI is generated. This NMI determines that an MCD error was generated due to a clock failure which is handled in the ISR.

Before the MCD the clock frequency will be as per device initialization. Post MCD error, the clock will be running at the speed of INTOSC1 with the PLL disabled (10 MHz).

The example also shows how to lock the PLL after missing clock detection, by first explicitly switching the clock source to INTOSC1, resetting the missing clock detect circuit and then re-locking the PLL. After that the clock frequency will be restored to the original configuration.

External Connections

- None.

Watch Variables

- **fail** - Indicates that a missing clock was either not detected or was not handled correctly.
- **nmilSRFlag** - Indicates that the missing clock failure caused an NMI to be triggered and called the ISR to handle it.
- **mcdDetect** - Indicates that a missing clock was detected.
- **result** - Status of a successful handling of missing clock detection.

7.5 RAM access protection violation detection

sdl_ex_ram_access_protect

This example demonstrates a functional test of the RAM access protection violation detection and handling for several different types accesses. The example generates a number of master or

non-master fetch, write, or read violations from the CPU, CLA, and DMA as applicable to the device.

External Connections

- None.

Watch Variables

- **violationISRFlag** - Indicates that the access violation interrupt was triggered and called the ISR.
- **illegalISRFlag** - Indicates that a CPU fetch violation interrupt was generated and the ITRAP ISR ran.
- **violationStatus** - Access protection violation status flags read in the ISR.
- **violationAddr** - Address at which the access violation occurred.
- **result** - Status of a successful detection and handling of RAM access protection violations.

7.6 Test of parity/ECC logic in SRAM

sdl_ex_ram_ecc_parity_test

This example demonstrates how to check the SRAM ECC or parity logic functionality.

Three types of SRAM errors are generated to demonstrate the use of the SRAM test modes and the checking of the related flag and information registers. These types are parity errors (LSx, GSx, and CLAMSG RAMs) and correctable and uncorrectable ECC errors (M0 and M1 RAMs).

External Connections

- None.

Watch Variables

- **nmiISRFlag** - Indicates that the NMI was triggered and called the ISR.
- **nmiStatus** - NMI status flags read in the ISR.
- **errorISRFlag** - Indicates that the correctable RAM error interrupt was triggered and called the ISR.
- **errorStatus** - Correctable RAM error status flags read in the ISR.
- **errorCount** - Number of correctable errors detected read in the ISR.
- **errorAddr** - Address at which the error was detected.
- **result** - Status of a successful detection and handling of RAM errors.

7.7 Software test of watchdog operation

sdl_ex_watchdog

This example shows how to use the wakeup interrupt to test the operation of the watchdog. The example generates two watchdog interrupts. The first is caused by delaying the reset of the watchdog too long, causing it to overflow. The second uses the windowing functionality to reset the counter before the minimum threshold can be reached.

External Connections

- None.

Watch Variables

- **wakeISRFlag** - Indicates the watchdog ISR was executed.
- **result** - Status of successful generation of expected watchdog pulse.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information 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.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Medical	www.ti.com/medical
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2021, Texas Instruments Incorporated