Motorola Built-In Test (MBIT) Diagnostic Software

User's Manual

MBITA/UM1

June 2002 Edition

© Copyright 2002 Motorola Inc.

All rights reserved.

Printed in the United States of America.

Motorola and the Motorola symbol are registered trademarks of Motorola, Inc.

Tornado and VxWorks are registered trademarks of Wind River Systems, Inc.

Windows is a registered trademark of Microsoft in the U.S. and other countries.

All other product or service names mentioned in this document are trademarks or registered trademarks of their respective holders.

Notice

While reasonable efforts have been made to assure the accuracy of this document, Motorola, Inc. assumes no liability resulting from any omissions in this document, or from the use of the information obtained therein. Motorola reserves the right to revise this document and to make changes from time to time in the content hereof without obligation of Motorola to notify any person of such revision or changes.

Electronic versions of this material may be read online, downloaded for personal use, or referenced in another document as a URL to the Motorola Computer Group Web site. The text itself may not be published commercially in print or electronic form, edited, translated, or otherwise altered without the permission of Motorola, Inc.

It is possible that this publication may contain reference to or information about Motorola products (machines and programs), programming, or services that are not available in your country. Such references or information must not be construed to mean that Motorola intends to announce such Motorola products, programming, or services in your country.

Limited and Restricted Rights Legend

If the documentation contained herein is supplied, directly or indirectly, to the U.S. Government, the following notice shall apply unless otherwise agreed to in writing by Motorola, Inc.

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (b)(3) of the Rights in Technical Data clause at DFARS 252.227-7013 (Nov. 1995) and of the Rights in Noncommercial Computer Software and Documentation clause at DFARS 252.227-7014 (Jun. 1995).

Motorola, Inc.
Computer Group
2900 South Diablo Way
Tempe, Arizona 85282

Contents

CHAPTER 3 Integrating Custom Diagnostics

CHAPTER 4 Utility Methods

CHAPTER 5 MBIT Faults

APPENDIX A API Method's Reference Pages

APPENDIX B Integrating Custom Diagnostics' Reference Pages

APPENDIX C Utility Methods' Reference Pages

APPENDIX D Installing MBIT with Tornado 2.1 and VxWorks

APPENDIX E Known Issues

APPENDIX F Related Documentation

List of Tables

About This Manual

This manual explains how to install and use the Motorola Built-In Test (MBIT) 1.01 diagnostic software for MVME51*xx* family boards running the Wind River Systems, Inc. VxWorks® real-time operating system. MBIT also depends on the use of the Tornado® 2.1 development environment.

This manual is a companion to the *Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide* listed in Appendix F, *Related Documentation*. The *Test Reference Guide* identifies and describes the supported devices and subtests needed to create test lists as part of a diagnostic application.

This *User's Manual* supports both the board level version of MBIT (PN: MBIT-BRD-51XX) and the system level version of MBIT (PN: MBIT-SYS-51XX). Refer to Chapter 1, *MBIT Overview* for a description of each version.

This manual is intended for use by software programmers or individuals with experience in the C programming language.

As of the printing date of this manual, MBIT supports the MVME51*xx* models listed below.

Model Number	**Description**
All models of the MVME51*xx* are available with either a VME Scanbe front panel (-xxx1) or a IEEE 1101 compatible front panel (-xxx3).	
450 MHz MPC750 Class Commercial Models	
MVME5100-013*x*	450 MHz MPC750 class, 64MB ECC SDRAM, 17MB Flash and 1MB L2 cache
MVME5100-016*x*	450 MHz MPC750 class, 512MB ECC SDRAM, 17MB Flash and 1MB L2 cache
400 MHz MPC755 Class Extended Temperature Models	
MVME5106-114*x*	400 MHz MPC755 class, 128MB ECC SDRAM and 1MB L2 cache

Model Number	Description
MVME5106-115*x*	400 MHz MPC755 class, 256MB ECC SDRAM and 1MB L2 cache
MVME5106-116*x*	400 MHz MPC755 class, 512MB ECC SDRAM and 1MB L2 cache
400 MHz MPC7400 Commercial Models	
MVME5101-013*x*	400 MHz MPC7400, 64MB ECC SDRAM, 17MB Flash and 1MB L2 cache
MVME5101-016*x*	400 MHz MPC7400, 512MB ECC SDRAM, 17MB Flash and 1MB L2 cache
MVME5101-213*x*	400 MHz MPC7400, 64MB ECC SDRAM, 17MB Flash and 2MB L2 cache
MVME5101-214*x*	400 MHz MPC7400, 128MB ECC SDRAM, 17MB Flash and 2MB L2 cache
MVME5101-216*x*	400 MHz MPC7400, 512MB ECC SDRAM, 17MB Flash and 2MB L2 cache
400 and 500 MHz MPC7410 Commercial Models	
MVME5110-213*x*	400 MHz MPC7410, 64MB ECC SDRAM and 2MB L2 cache
MVME5110-214*x*	400 MHz MPC7410, 128MB ECC SDRAM and 2MB L2 cache
MVME5110-215*x*	400 MHz MPC7410, 256MB ECC SDRAM and 2MB L2 cache
MVME5110-216*x*	400 MHz MPC7410, 512MB ECC SDRAM and 2MB L2 cache
MVME5110-223*x*	500 MHz MPC7410, 64MB ECC SDRAM and 2MB L2 cache
MVME5110-224*x*	500 MHz MPC7410, 128MB ECC SDRAM and 2MB L2 cache
MVME5110-225*x*	500 MHz MPC7410, 256MB ECC SDRAM and 2MB L2 cache
MVME5110-226*x*	500 MHz MPC7410, 512MB ECC SDRAM and 2MB L2 cache

Model Number	Description
500 MHz MPC7410 Extended Temperature Models	
MVME5107-214*x*	500 MHz MPC7410, 128MB ECC SDRAM and 2MB L2 cache
MVME5107-215*x*	500 MHz MPC7410, 256MB ECC SDRAM and 2MB L2 cache
MVME5107-216*x*	500 MHz MPC7410, 512MB ECC SDRAM and 2MB L2 cache

Overview of Contents

This manual is divided into the following chapters and appendices:

Chapter 1, *MBIT Overview*, provides a high-level overview of the system level and board level versions of MBIT.

Chapter 2, *Using MBIT*, provides the MBIT application programming interface, as well as an example of how to use MBIT.

Chapter 3, *Integrating Custom Diagnostics*, provides instructions on how to integrate custom diagnostics available with the system level version of MBIT.

Chapter 4, *Utility Methods*, provides methods to complete various activities while integrating custom diagnostics in the system level version of MBIT.

Chapter 5, *MBIT Faults*, provides the faults built into the MBIT API or pre-defined by the MVME51*xx* diagnostics.

Appendix A, *API Method's Reference Pages*, provides detailed information about the MBIT API methods mentioned in Chapter 2, *Using MBIT*.

Appendix B, *Integrating Custom Diagnostics' Reference Pages*, provides detailed information about the diagnostic integration methods, generic device driver methods, device driver methods, and the device read/write utility methods mentioned in Chapter 3, *Integrating Custom Diagnostics*.

Appendix C, *Utility Methods' Reference Pages*, provides detailed information about the utility methods mentioned in Chapter 4, *Utility Methods*.

Appendix D, *Installing MBIT with Tornado 2.1 and VxWorks*, provides instructions on how to install MBIT with the Tornado 2.1 development system.

Appendix E, *Known Issues*, provides known issues with the MBIT diagnostic software.

Appendix F, *Related Documentation*, provides a list of related documentation for the MBIT software.

Comments and Suggestions

Motorola welcomes and appreciates your comments on its documentation. We want to know what you think about our manuals and how we can make them better. Mail comments to:

Motorola Computer Group
Reader Comments DW164
2900 S. Diablo Way
Tempe, Arizona 85282

You can also submit comments to the following e-mail address:
reader-comments@mcg.mot.com

In all your correspondence, please list your name, position, and company. Be sure to include the title and part number of the manual and tell how you used it. Then tell us your feelings about its strengths and weaknesses and any recommendations for improvements.

Conventions Used in This Manual

The following typographical conventions are used in this document:

bold

is used for user input that you type just as it appears; it is also used for commands, options and arguments to commands, and names of programs, directories and files.

italic

is used for names of variables to which you assign values, for function parameters, and for structure names and fields. Italic is also used for comments in screen displays and examples, and to introduce new terms.

`courier`

is used for system output (for example, screen displays, reports), examples, and system prompts.

<Enter>, **<Return>** or **<CR>**

represents the carriage return or Enter key.

Ctrl

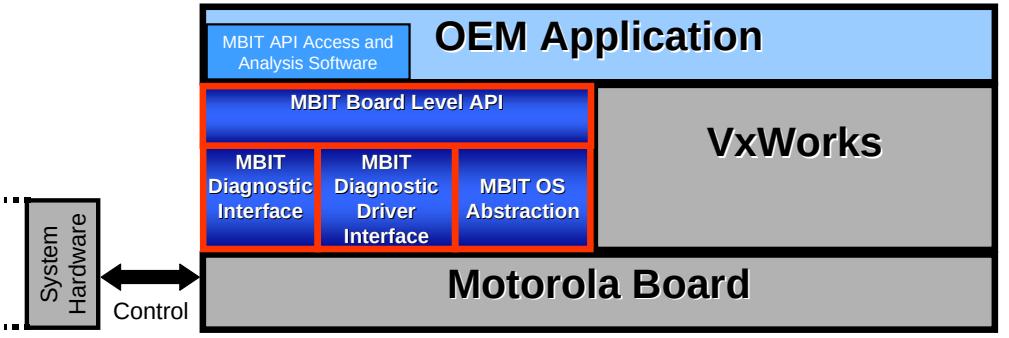
represents the Control key. Execute control characters by pressing the Ctrl key and the letter simultaneously, for example, Ctrl-d.

MBIT Overview 1

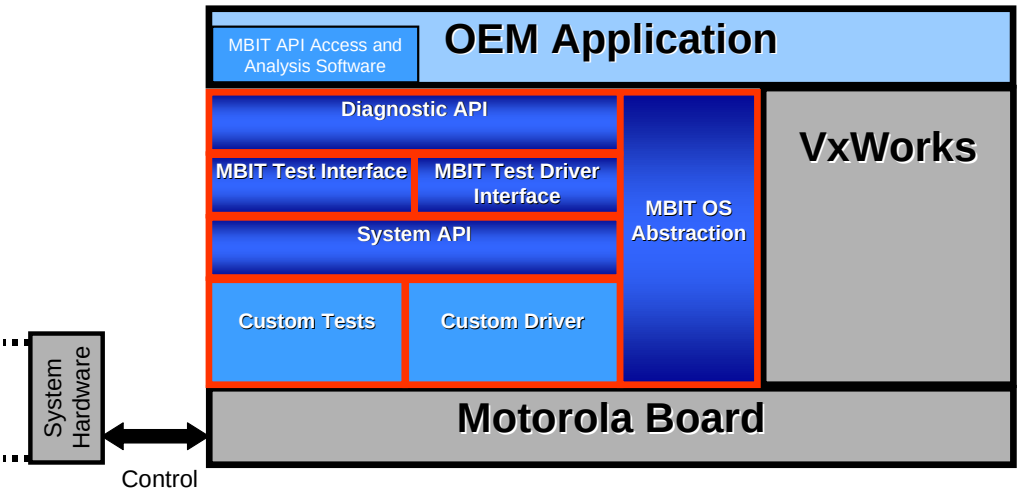
Introduction

MBIT is an off-the-shelf software infrastructure designed to verify the correct operation of Motorola Computer Group hardware. MBIT is available in two versions—board level MBIT and system level MBIT.

- Board level MBIT (PN: MBIT-BRD-51XX)—a comprehensive diagnostic software package designed to verify the correct operation of board mounted logical devices. All tests can execute at boot-up and selected tests can run continuously in the background of user applications. An application programming interface (API) is included to provide access to test results and to control the operation of device tests. Chapter 2, *Using MBIT* and Appendix A, *API Method's Reference Pages* are specifically for the board level version of MBIT.



- System level MBIT (PN: MBIT-SYS-51XX)—includes all of the functionality and API function calls of the board level version and enables system-wide testing. It provides a framework and additional API methods to support the inclusion of software designed to test custom hardware and/or system components. Chapter 3, *Integrating Custom Diagnostics*, Chapter 4, *Utility Methods*, Appendix B, *Integrating Custom Diagnostics' Reference Pages*, and Appendix C, *Utility Methods' Reference Pages* are specifically for the system level version of MBIT.



Before using the MBIT diagnostic software, connect and configure the board and other hardware according to the respective installation guide.

System Requirements

In order to successfully install and use this diagnostic software, you need the following items:

- ✓ Wind River Systems, Inc. Tornado 2.1 development environment
- ✓ Wind River Systems, Inc. VxWorks® real-time operating system
- ✓ Motorola's MVME51*xx* VME processor module
- ✓ Wind River Systems, Inc. board support package (BSP) for Motorola's MVME51*xx* VME processor module (PN: TDK-14498-ZC)

Refer to the Help pull down menu in your Tornado 2.1 environment for more information on system requirements.

Installation

Refer to Appendix D, *Installing MBIT with Tornado 2.1 and VxWorks*, for instructions on how to install the MBIT diagnostic software.

MBIT Features

MBIT provides PowerPC™ architecture-compatible single-board computers with an API that allows the user's application to control subtest execution and sequencing. The following list summarizes general features and functions of the API and associated diagnostic.

- ❏ The user's application controls diagnostic execution and sequencing. The application dictates the execution of each diagnostic.
- ❏ The user's application can extend MBIT to add custom diagnostics in the system level product.
- ❏ The user's application can invoke lists of diagnostics.
- ❏ Each diagnostic executes independently of all other diagnostic.

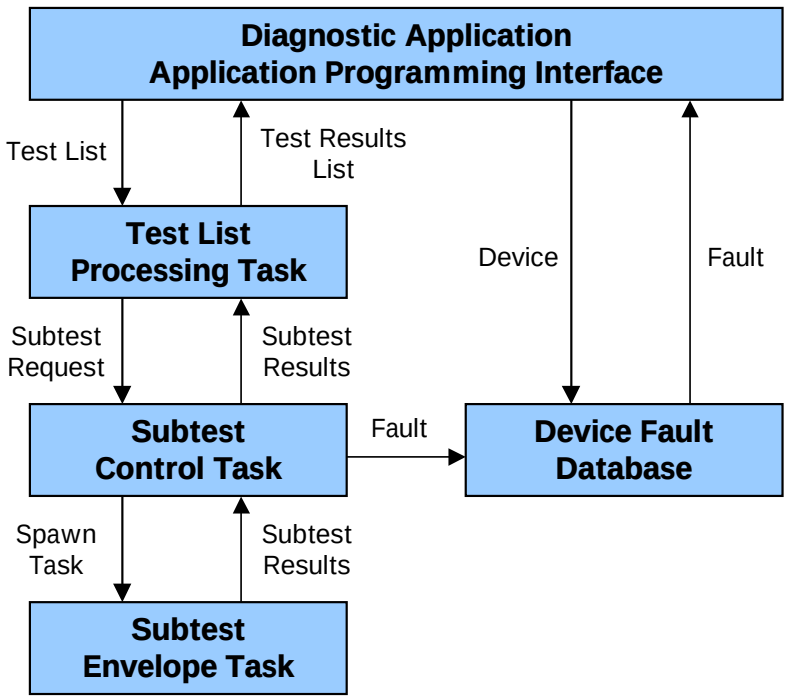
- ❏ Each diagnostic provides its own initialization, resource allocation, and resource de-allocation.
- ❏ The diagnostic application can control whether a subtest halts on the first error detected or runs until subtest completion.
- ❏ The diagnostic application can clear the results of previously executed subtests.
- ❏ The diagnostic application can halt a subtest in progress.
- ❏ MBIT uses a time-out mechanism on diagnostics to prevent them from waiting for an event that may never occur due to a hardware malfunction.
- ❏ Exception handling mechanisms help prevent the diagnostic application from failing due to hardware faults during subtest execution.
- ❏ MBIT reports the status for each executed subtest.
- ❏ MBIT diagnostics cover 95 percent of stuck-at faults for board address and data nodes.
- ❏ Total subtest execution time for all built-in subtests on all devices, with the exception of the memory subtests, does not exceed five minutes.

The MBIT diagnostics cover more than 95 percent of address and data nodes on MVME51*xx* family boards alone, and in combination with an attached IPMC712. To calculate the coverage, the analysis process classified each node on the board (a trace connecting one or more components) as an address, data, control or other. For each node classified as address or data, the analysis process verified that each diagnostic could detect stuck-at zero or stuck-at one faults. This coverage calculation assumes executing diagnostics for each device, executing all the subtests for a device, and using the default parameters for the subtests.

MBIT Process

The MBIT diagnostic software is explained in a five-part process, ending with a summary of subtest results. The process executes in four separate threads and communicates by message passing.

1. *Diagnostic Application*
2. *Test List Processing Task*
3. *Subtest Control Task*
4. *Subtest Envelope Task*
5. *Device Fault Database*



Diagnostic Application

The diagnostic application submits a test list for execution via the API. A test list contains a set of subtests and only a single list can be submitted at any one time. Further test list requests are ignored until the current test list is processed. Once a test list is submitted, a message is queued to the *Test List Processing Task* and the API returns control to the user's application to await future commands.

The MBIT API methods are described in greater detail in Chapter 2, *Using MBIT* and in Appendix A, *API Method's Reference Pages*.

Test List Processing Task

Upon receipt of the test list message, the test list processing task buffers the test list and queues a subtest for processing by the *Subtest Control Task*. A subtest is sent and the test list processing task waits for test completion. Upon receipt of the test results, the test list processing task buffers the completion data in a test list results message, which returns to the diagnostic application upon completion of all subtests contained in the test list message. The next subtest in the test list is then queued to the subtest control task. The results are returned through a response message queue.

Subtest Control Task

Upon receipt of a subtest response message, the subtest control task calls the configured subtest installation method. A task is spawned to run the configured subtest execution method. A timer is set to limit execution time and the subtest control task waits for subtest completion. Refer to the *Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide*, listed in Appendix F, *Related Documentation*, for default subtest time limits.

Upon receipt of the subtest completion message, the subtest control task stores the completion data for the logical device being tested in the results message and device fault database. The subtest control task also invokes the subtest de-installation method and forwards the results message to the test list processing task. Subtest installation and de-installation methods are responsible for allocating and freeing resources required by the subtest.

If the test results are not received before the timer expires, it is assumed the subtest is unable to complete its testing. In such a case, the spawned subtest task is deleted, a time-out indication is stored in the fault database, and the time-out indication is queued to the test list processing task.

If subtest control receives an abort directive while a test list is executing, the spawned subtest task is deleted. The value **BIT_TEST_ABORTED** is placed in a subtest result message, which is then queued to the test list processing task. The fault database is not updated, since an operator abort directive does not represent a device failure.

Subtest Envelope Task

The test execution task initializes a subtest results message and calls the configured subtest repeatedly based on the iteration count. Upon completion of all required test iterations, a summary of the results is placed in the results message and is then queued to the subtest control task. The test execution task then exits.

Device Fault Database

The device fault database contains the results of subtest execution for each supported device. The results of subtest execution begin to accumulate in the fault database after API initialization. The results may indicate unexecuted tests on the device. If there are executed tests on the device, the result indicates either success or the first fault detected for the device. The user's application can obtain fault information for logical devices by invoking the **getBitDeviceFault()** method (see Chapter 2, *Using MBIT* or Appendix A, *API Method's Reference Pages* for details on this method).

Note The device fault database clears when the **reinitBit()** method is invoked.

Using MBIT

The MBIT software provides the methods contained in the MBIT API. These methods, which are listed below, initialize the software, control subtest execution and sequencing, terminate the software, and perform several other functions necessary for a diagnostic application. Appendix A, *API Method's Reference Pages* describes each of these methods in greater detail.

Method	Description
initBit()	Initiates the MBIT software.
reinitBit()	Clears the device fault database.
isBitInitializationComplete()	Returns the MBIT initialization status.
executeBitTests()	Executes a list of subtests.
buildBitDefaultTestList()	Fills in a test list with default test entries for each subtest associated with the given device.
buildBitDefaultTestEntry()	Fills in a test list with a single default test entry for the associated subtest and device.
getBitResponse()	Obtains a list of test results.
getNumBitResponses()	Provides the number of MBIT test results lists in the response queue.
abortBitTests()	Aborts a subtest or group of subtests.
getBitDeviceFault()	Gets a fault for the specified logical device from the device fault database.
getBitSubtestDesc()	Gets a subtest description.
getBitDeviceDesc()	Gets a logical device description.
getBitFaultDesc()	Gets a fault description.
getBitSubtestIdByName()	Gets the subtest ID for the corresponding subtest name.

Method	Description
getBitDeviceIdByName()	Gets the device ID for the corresponding device name.
getBitFaultIdByName()	Gets the fault ID for the corresponding fault name.
getBitNumberOfSubtests()	Gets the number of subtests.
getBitNumberOfDevices()	Gets the number of devices.
getBitNumberOfFaults()	Gets the number of faults.
getBitMaxTestListEntries()	Gets the maximum number of test list entries supported by executeBitTests() and getBitResponse().
terminateBit()	Terminates the MBIT software.

Initializing MBIT

The **initBit()** method initializes MBIT, **reinitBit()** clears the device fault database, and **terminateBit()** terminates MBIT. The **isBitInitializationComplete()** method returns **TRUE** or **FALSE**, depending on whether or not the MBIT initialization is complete.

initBit()

initBit() performs MBIT initialization and must be invoked prior to any other method. This method creates the test list processing task and the subtest control task.

Here's a synopsis of the **initBit()** method:

```
#include <api/bitApi.h>
BIT_FAULT initBit(BIT_FAULT (*pConfigRoutines[])(),
                              int numConfigRoutines)
```

where *pConfigRoutines[]* is an array of function pointers to custom subtest and device configuration methods and *numConfigRoutines* is the number of custom configuration methods.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

reinitBit()

reinitBit() clears the device fault database and extinguishes the Fail LED.

Here's a synopsis of the **reinitBit()** method:

```
#include <api/bitApi.h>
BIT_FAULT reinitBit(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

isBitInitializationComplete()

isBitInitializationComplete() returns the MBIT initialization status.

Here's a synopsis of the **isBitInitializationComplete()** method:

```
#include <api/bitApi.h>
BOOLEAN isBitInitializationComplete(void)
```

This method returns **TRUE** if the MBIT initialization is complete, **FALSE** if it is not.

Executing Subtests in MBIT

Executing subtests begins with creating test lists and submitting them for execution by calling **executeBitTests()**. Follow this method up with **getBitResponse()**, which returns test results. **abortBitTests()** allows aborting test execution any time outside critical sections during tests. See the section on subtest attributes in the *Test Reference Guide* for a list of subtests with protected critical sections.

A user may obtain a test list filled with default test entries for a given device by calling either **buildBitDefaultTestList()** or **buildBitDefaultTestEntry()**.

An MBIT application may occupy two threads of execution. All API methods, except **getBitResponse()**, must be called from the thread

initBit() is called from. **getBitResponse()** may be called from another thread.

executeBitTests()

executeBitTests() submits and executes a list of subtests. The test list processing task buffers the test list and processes it in the background.

Here's a synopsis of the **executeBitTests()** method:

```
#include <api/bitApi.h>
BIT_FAULT executeBitTests(BIT_TEST_CONTROL listControl,
                          unsigned int testCount,
                          TEST_ENTRY testList[])
```

where *listControl* specifies **HALT_ON_ERROR** or **RUN_TILL_COMPLETION** (see **config/bitCommonDefs.h**), *testCount* is the number of entries in the test list, and *testList[]* is an array of tests to execute.

For each successfully submitted test list, MBIT places a single test results list in the response queue.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

Creating Test Lists

In creating a test list, the following applies:

- ❏ Any subtest can be included in a single test list and a test list may contain a single entry.
- ❏ The number of subtest entries in a test list is limited to the number of entries returned by the **getBitMaxTestListEntries()** method.
- ❏ Only one subtest is executed at any one time.

MBIT provides two methods to create test lists with default test entries. These methods can be used as a starting point for configuring subtest parameters. The **buildBitDefaultTestList()** method fills a test

list with subtests associated with a given device. The **buildBitDefaultTestEntry()** method fills a test list with a single subtest associated with a given device. These test lists may then be submitted to the **executeBitTests()** method for processing.

Executing a Test List

To execute a test list, the user must specify a list control with one of the values, **HALT_ON_ERROR** or **RUN_TILL_COMPLETION**. If **HALT_ON_ERROR** is specified, processing of the test list terminates with the detection of the first failed subtest. Otherwise, all subtests in the test list are executed. If a subtest fails, the board fail LED illuminates.

For more information on executing a test list, refer to Appendix A, *API Method's Reference Pages*.

buildBitDefaultTestList()

buildBitDefaultTestList() fills in a test list with default test entries for each subtest associated with the given device. The list of test entries must be allocated before this method is called. The maximum number of test entries returned is no more than the value returned by the **getBitMaxTestListEntries()** method. Refer to the *Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide* for a list of subtests and the associated devices.

Here's a synopsis of the **buildBitDefaultTestList()** method:

```
#include <api/bitApi.h>
BIT_FAULT buildBitDefaultTestList(
                                BIT_LOGICAL_DEVICE deviceId,
                                unsigned int *numTests,
                                TEST_ENTRY testEntryList[])
```

where *deviceId* is an ID specifying a unique device. The number of test entries returned is placed in the integer at *numTests* and the test entries for the device are placed in the buffer starting at *testEntryList[]*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the

return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

buildBitDefaultTestEntry()

buildBitDefaultTestEntry() fills in a single default test entry for the associated subtest and device. The test entry must be allocated before this method is called.

Here's a synopsis of the **buildBitDefaultTestEntry()** method:

```
#include <api/bitApi.h>
BIT_FAULT buildBitDefaultTestEntry(BIT_SUBTEST subtestId,
                                   BIT_LOGICAL_DEVICE deviceId,
                                   TEST_ENTRY *testEntry)
```

where *subtestId* is an ID specifying a unique subtest, *deviceID* is an ID specifying a unique device, and the test entry for subtest and device is placed in the *TEST_ENTRY* as *testEntry*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

getBitResponse()

getBitResponse() provides the results of a test list submitted by invoking **executeBitTests()**. **getBitResponse()** blocks until the executing subtest completes, times out, or aborts. If this method is called when no subtests are executing, it blocks until a call is made to **executeBitTests()** and all subtests in the list complete execution. **getBitResponse()** returns a single test results list and removes it from the response queue.

Here's a synopsis of the **getBitResponse()** method:

```
#include <api/bitApi.h>
BIT_FAULT getBitResponse(TEST_RESULTS_ENTRY testResults[],
                                   unsigned int *numberOfResults)
```

where *testResults[]* is a user allocated array for the test results and *numberOfResults* will receive the number of entries in the test results list.

If the number of results exceeds the value returned by **getBitMaxTestListEntries()**, an error is returned.

The number of *testResults* entries allocated must be greater than or equal to the number of test entries submitted with **executeBitTests()**. The number of *testResults* returned will be less than or equal to the number of test entries submitted with **executeBitTests()**.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

getNumBitResponses()

getNumBitResponses() provides the number of MBIT test results lists in the MBIT response queue.

Here's a synopsis of the **getNumBitResponses()** method:

```
#include <api/bitApi.h>
BIT_FAULT getNumBitResponses(int *msgCount)
```

where *msgCount* will contain the number of test results lists in the MBIT response queue.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

abortBitTests()

abortBitTests() terminates current test list processing and aborts any subtest in progress. This method has no effect if a subtest is not executing or has already completed. Tests with protected critical sections are not aborted until the critical section is exited. Test results for those tests already complete are made available in response to the submitted test list. For each successfully submitted test list, a single test results list is placed in the response queue.

Invoking **abortBitTests()**, then **reinitBit()**, is sufficient to place MBIT in an initial state.

Here's a synopsis of the **abortBitTests()** method:

```
#include <api/bitApi.h>
BIT_FAULT abortBitTests(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

Obtaining IDs in MBIT

The **getBitSubtestIdByName()**, **getBitDeviceIdByName()**, and **getBitFaultIdByName()** methods return an ID representing the string identifier.

getBitSubtestIdByName()

getBitSubtestIdByName() returns the ID representing the subtest string identifier.

Here's a synopsis of the **getBitSubtestIdByName()** method:

```
#include <api/bitApi.h>
BIT_SUBTEST getBitSubtestIdByName(const char* const subtest)
```

where *subtest* is a string identifier specifying the subtest.

Upon successful completion, a **BIT_SUBTEST** is returned, which is an ID representing the subtest identifier. Otherwise, **–1** is returned if the subtest is not found. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitDeviceIdByName()

getBitDeviceIdByName() returns the ID representing the device string identifier.

Here's a synopsis of the **getBitDeviceIdByName()** method:

```
#include <api/bitApi.h>
BIT_LOGICAL_DEVICE getBitDeviceIdByName(const char* const
device)
```

where *device* is a string identifier specifying the device.

Upon successful completion, a **BIT_LOGICAL_DEVICE** is returned, which is an ID representing the device identifier. Otherwise, **–1** is returned if the device is not found. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitFaultIdByName()

getBitFaultIdByName() returns the ID representing the fault string identifier.

Here's a synopsis of the **getBitFaultIdByName()** method:

```
#include <api/bitApi.h>
BIT_FAULT getBitFaultIdByName(const char* const fault)
```

where *fault* is a string identifier specifying the fault

Upon successful completion, a **BIT_FAULT** is returned, which is an ID representing the fault identifier. Otherwise, **–1** is returned if the fault is not found. Refer to Chapter 5, *MBIT Faults* for more return values.

Obtaining Faults in MBIT

The **getBitDeviceFault()** method obtains fault information for a specified logical device.

getBitDeviceFault()

getBitDeviceFault() obtains the fault data for the device specified by the input device enumeration value. The first fault detected for the specified logical device returns to the caller.

Invoking **reinitBit()** clears the collected fault data.

Here's a synopsis of the **getBitDeviceFault()** method:

```
#include <api/bitApi.h>
BIT_FAULT getBitDeviceFault(BIT_LOGICAL_DEVICE device,
                            BIT_FAULT *deviceFault)
```

where *device* is the logical device for the requested fault data and the returned fault code will be place in the **BIT_FAULT** as *deviceFault*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

Obtaining String Descriptions in MBIT

The **getBitSubtestDesc()**, **getBitDeviceDesc()**, and **getBitFaultDesc()** methods obtain strings describing subtests, logical devices and diagnostic faults, respectively. The IDs mentioned in these methods are obtained from the corresponding methods: **getBitSubtestIdByName()**, **getBitDeviceIdByName()**, and **getBitFaultIdByName()**.

The string descriptions returned should not be modified or freed.

getBitSubtestDesc()

getBitSubtestDesc() returns a string describing the subtest.

Here's a synopsis of the **getBitSubtestDesc()** method:

```
#include <api/bitApi.h>
const char* getBitSubtestDesc(BIT_SUBTEST subtestId)
```

where *subtestId* is an ID specifying a unique subtest.

Upon successful completion, **getBitSubtestDesc()** returns a string containing the subtest description. If it fails, it returns an empty string. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitDeviceDesc()

getBitDeviceDesc() returns a string describing the logical device.

Here's a synopsis of the **getBitDeviceDesc()** method:

```
#include <api/bitApi.h>
const char* getBitDeviceDesc(BIT_LOGICAL_DEVICE deviceId)
```

where *deviceId* is an ID specifying a unique device.

Upon successful completion, **getBitDeviceDesc()** returns a string containing the device description. If it fails, it returns an empty string. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitFaultDesc()

getBitFaultDesc() returns a string describing the fault.

Here's a synopsis of the **getBitFaultDesc()** method:

```
#include <api/bitApi.h>
const char* getBitFaultDesc(BIT_FAULT faultId)
```

where *faultId* is an ID specifying a unique fault.

Upon successful completion, **getBitFaultDesc()** returns a string containing the fault description. If it fails, it returns "No description supplied for fault." Refer to Chapter 5, *MBIT Faults* for more return values.

Obtaining Counts in MBIT

getBitNumberOfSubtests(), **getBitNumberOfDevices()**, **getBitNumberOfFaults()**, and **getBitMaxTestListEntries()** return a count of subtests, devices, faults, and test list entries.

getBitNumberOfSubtests()

getBitNumberOfSubtests() returns the number of subtests.

Here's a synopsis of the **getBitNumberOfSubtests()** method:

```
#include <api/bitApi.h>
int getBitNumberOfSubtests(void)
```

Upon successful completion, **getBitNumberOfSubtests()** returns the number of MBIT configured subtests. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitNumberOfDevices()

getBitNumberOfDevices() returns the number of devices.

Here's a synopsis of the **getBitNumberOfDevices()** method:

```
#include <api/bitApi.h>
int getBitNumberOfDevices(void)
```

Upon successful completion, **getBitNumberOfDevices()** returns the number of MBIT configured devices. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitNumberOfFaults()

getBitNumberOfFaults() returns the number of faults.

Here's a synopsis of the **getBitNumberOfFaults()** method:

```
#include <api/bitApi.h>
int getBitNumberOfFaults(void)
```

Upon successful completion, **getBitNumberOfFaults()** returns the number of MBIT configured faults. Refer to Chapter 5, *MBIT Faults* for more return values.

getBitMaxTestListEntries()

getBitMaxTestListEntries() returns the maximum number of test list entries.

Here's a synopsis of the **getBitMaxTestListEntries()** method:

```
#include <api/bitApi.h>
int getBitMaxTestListEntries(void)
```

Upon successful completion, **getBitMaxTestListEntries()** returns the maximum number of MBIT configured test list entries. Refer to Chapter 5, *MBIT Faults* for more return values.

Terminating MBIT

The **terminateBit()** method performs an orderly termination of MBIT and releases all allocated resources.

terminateBit()

terminateBit() terminates MBIT, including releasing allocated resources and the termination of all spawned child tasks.

After invoking **terminateBit()**, you may call **initBit()**. There are no restrictions placed on the number of times you can call **initBit()** and **terminateBit()**, as long as each call to **initBit()** is followed by a call to **terminateBit()** prior to the next invocation of **initBit()**.

Note: Calling **initBit()** and **terminateBit()** an excessive number of times may cause memory fragmentation.

Here's a synopsis of the **terminateBit()** method:

```
#include <api/bitApi.h>
BIT_FAULT terminateBit(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix A, *API Method's Reference Pages* and Chapter 5, *MBIT Faults*.

Example: Using MBIT

The following is a simplified example, or sample application, of how to use MBIT. The example includes how to initialize MBIT, run a test, and then terminate the software.

```
#include <vxWorks.h>
#include <stdio.h>
#include <stdlib.h>
#include <api/bitApi.h>
#include <config/testDefaults.h>
#include <config/bitCommonDefs.h>
/*
bitSampleApplication

Calling this routine will initialize bit, run a test, and then
terminate bit when the test is finished. There is no error
checking performed in this application.
*/
void bitSampleApplication(void)
{
    BIT_FAULT status;
    BIT_SUBTEST subtest;
    BIT_LOGICAL_DEVICE device;
    TEST_ENTRY test[1];
    TEST_RESULTS_ENTRY testResults[1];
    unsigned int numResults;
    /*
    Run initBit so that all of MBIT is initialized. This needs
    to be done before running a subtest.
    */
    status = initBit(NULL,0);
    subtest = getBitSubtestIdByName("BIT_RAM_BIT_WALK");
    device = getBitDeviceIdByName("BIT_ECC_SDRAM");
    /*
    To pass parameters to the test, use the testParamPtr field
    in the TEST_ENTRY test structure. Then pass the test to
    the buildBitDefaultTestEntry. Otherwise, pass nothing in
    the test field and the defaults will be used.
    */
    status = buildBitDefaultTestEntry(subtest, device, test);
    status = executeBitTests(HALT_ON_ERROR, 1, test);
    status = getBitResponse(testResults, &numResults);
    status = terminateBit();
}
```

Integrating Custom Diagnostics 3

Introduction

This chapter explains integrating custom diagnostics for a developer. The diagnostic integration methods are used for configuring devices and subtests, and are used throughout this chapter.

Diagnostic Integration Methods

MBIT provides the following methods for integrating diagnostics:

Method	Description
addBitSubtestIdent()	Adds a subtest entry.
addBitDeviceIdent()	Adds a device entry.
addBitFaultIdent()	Adds a fault entry.
createBitTestAssociations()	Creates an association between a device, subtest, and a driver.
installBitDriver()	Installs the methods for the driver.
installBitSubtestEntries()	Installs the methods for the test and sets the test defaults.
getBitNumberOfAssociations()	Obtains the number of associations.

These methods may only be used during the initialization of MBIT, with the exception of **getBitNumberOfAssociations()**. These methods are used by the configuration methods that setup devices and/or subtests and may not be called during the diagnostic test.

addBitSubtestIdent()

This method adds a subtest entry with the provided identifier and description.

Here's a synopsis of the **addBitSubtestIdent()** method:

```
#include <config/bitTestUtils.h>
BIT_FAULT addBitSubtestIdent(const char *subtest,
                             const char *description,
                             BIT_SUBTEST *id)
```

where *subtest* is the unique subtest name, *description* is the description of the *subtest* name, and *id* is the unique ID being returned that represents the *subtest*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED** and places a new unique ID in *id*, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

addBitDeviceIdent()

This method adds a device entry with the provided identifier and description.

Here's a synopsis of the **addBitDeviceIdent()** method:

```
#include <config/bitTestUtils.h>
BIT_FAULT addBitDeviceIdent(const char *device,
                            const char *description,
                            BIT_LOGICAL_DEVICE *id)
```

where *device* is the unique device name, *description* is the description of the *device* name, and *id* is the unique ID being returned that represents the *device*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED** and places a new unique ID in *id*, otherwise it may return any of the return values listed in Appendix B,

Integrating Custom Diagnostics' Reference Pages and Chapter 5, *MBIT Faults*.

addBitFaultIdent()

This method adds a fault entry with the provided identifier and description.

Here's a synopsis of the **addBitFaultIdent()** method:

```
#include <config/bitTestUtils.h>
BIT_FAULT addBitFaultIdent(const char *fault,
                           const char *description,
                           BIT_FAULT_TYPE type,
                           BIT_FAULT *id)
```

where *fault* is the unique fault name, *description* is the description of the *fault*, *type* is the type of *fault* (for example, hardware, software; see **config/bitCommonDefs.h**), and *id* is the unique ID being returned that represents the *fault*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED** and places a new unique ID in *id*, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

createBitTestAssociations()

This method creates an association between subtests, devices, and a driver. If a driver is not available for the associated subtest and device, the *pDriveDesc* parameter should be **NULL**.

Here's a synopsis of the **createBitTestAssociations()** method:

```
#include <config/bitTestUtils.h>
BIT_FAULT createBitTestAssociations(BIT_SUBTEST subtestId[],
                           int numSubtestIds,
                           BIT_LOGICAL_DEVICE deviceId[],
                           int numDeviceIds,
                           DRV_DESC *pDriveDesc)
```

where *subtestId[]* is an array of subtest IDs, *numSubtestIds* is the number of subtest IDs, *deviceId[]* is an array of device IDs, *numDeviceIds* is the number of device IDs, and *pDriveDesc* is the pointer to the driver being associated with the subtests and devices.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

installBitDriver()

This method installs the driver entry points. When you add a driver you must implement all of the driver entry points.

Here's a synopsis of the **installBitDriver()** method:

```
#include <config/bitTestUtils.h>
DRV_DESC* installBitDriver(
     BIT_FAULT (*drvInstall) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvDeinstall) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvOpen) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvClose) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvRead) (DEV_DESC *devDescPtr,
                           unsigned int bufferSize,
                           char *buffer,
                           unsigned int *bytesRead),
     BIT_FAULT (*drvWrite) (DEV_DESC *devDescPtr,
                           unsigned int bufferSize,
                           char *buffer,
                           unsigned int *bytesWritten),
     BIT_FAULT (*drvIoctl) (DEV_DESC *devDescPtr
                           int function,
                           int argument))
```

where *drvInstall* is the driver install entry point, *drvDeinstall* is the driver deinstall entry point, *drvOpen* is the driver open entry point, *drvClose* is the driver close entry point, *drvRead* is the driver read entry point, *drvWrite* is the driver write entry point, and *drvIoctl* is the driver ioctl entry point. Refer to the *Device Driver Interface* section for more information on the device driver methods.

Upon successful completion, **DRV_DESC** is returned, which is the pointer to the driver descriptor. If an error occurs, **NULL** is returned. Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more return values.

installBitSubtestEntries()

This method installs the required subtest entry points and sets the default parameters, which are used by **buildBitDefaultTestList()** and **buildBitDefaultTestEntry()**.

Here's a synopsis of the **installBitSubtestEntries()** method:

```
#include <config/bitTestUtils.h>
BIT_FAULT installBitSubtestEntries(
     BIT_SUBTEST subtest,
     BIT_FAULT (*installTest) (BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device,
                               void *testParamPtr),
     BIT_FAULT (*deinstallTest) (BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device,
                               void *testParamPtr),
     BIT_FAULT (*runTest) (BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device,
                               void *testParamPtr),
     int interations,
     int durations
     BIT_TEST_CONTROL control,
     BIT_FAULT (*freeParamPtr) (BIT_SUBTEST subtest,
                               void *testParamPtr),
     BIT_FAULT (*initParamPtr) (BIT_SUBTEST subtest,
                               void *testParamPtr),
     int paramSize)
```

where *subtest* is the subtest ID the methods and default parameters are associated with, *installTest* is the subtest installation method, *deinstallTest* is the subtest de-installation method, *runTest* is the actual test method, *iterations* is the default number of times to run the test, *duration* is the default maximum number of milliseconds the test is allowed to run, *control* is the default test control to halt on the first error detected or to run until test completion, *freeParamPtr* is the pointer to the free parameter method,

initParamPtr is the pointer to the method that initializes the default parameter structure, and *paramSize* is the size of the parameter structure used by the subtest

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

getBitNumberOfAssociations()

This method returns the number of associations and should only be used after initializing MBIT. If called during initialization, this method will return an invalid number of associations.

Here's a synopsis of the **getBitNumberOfAssociations()** method:

```
#include <config/bitTestUtils.h>
int getBitNumberOfAssociations(void)
```

Upon successful completion, **getBitNumberOfAssociations()** returns the number of associations. If MBIT has not been initialized, it returns **0**.

Implementing Subtests

The MBIT API provides a method that allows the operator to abort an executing subtest. This is achieved by providing a separate thread of execution for the subtest and a method of terminating this thread of execution. Any resource allocation must be viewed as a critical region. During execution of such a critical region, termination must not be allowed to take place or resources may not be capable of being reclaimed. These public methods are installed in *installBitSubtestEntries()*.

Subtest Structure

MBIT requires that each subtest implements the following public methods: test installation, test execution, and test de-installation. These methods

must perform all the resource allocation and de-allocation required by the subtest, including a device driver installation and any memory allocation.

Test Method	Definition
Test installation	Allows resource allocation and driver installation.
Test execution	Implements the bulk of the test and is subject to unexpected termination. Resource allocation is discouraged.
Test de-installation	Allows removing drivers, freeing allocated resources, and placing the device in a known state.

All methods return a **BIT_FAULT**, as defined in **config/bitCommonDefs.h**. The faults returned may either be pre-defined (refer to Chapter 5, *MBIT Faults*) or added during subtest configuration. Each method passes in a logical device and a subtest, which are **BIT_LOGICAL_DEVICE** and **BIT_SUBTEST**. The test execution method contains a control which directs the test to halt on the first error detected or to run until test completion, if possible. The *BIT_TEST_CONTROL* definition is located in **bitCommonDefs.h**. The last parameter is passed to all methods and is a pointer to a structure containing test parameters. This parameter structure is unique to the subtest and contains items to control test execution such as patterns, buffer sizes, etc.

Example: Subtest Structure

The following is an example of a subtest implementing the three public test methods:

```
BIT_FAULT installL2CacheTest(BIT_LOGICAL_DEVICE device,
                             BIT_SUBTEST subtest,
                             void *parameters)
{
     return(getBitFaultIdByName("BIT_NO_FAULT_DETECTED"));
}
BIT_FAULT runL2CacheTest(BIT_LOGICAL_DEVICE device,
                             BIT_SUBTEST subtest,
                             BIT_SUBTEST_CONTROL control,
                             void *parameters)
{
     return(getBitFaultIdByName("BIT_NO_FAULT_DETECTED"));
}
BIT_FAULT deinstallL2CacheTest(BIT_LOGICAL_DEVICE device,
                             BIT_SUBTEST subtest,
                             void *parameters)
{
     return(getBitFaultIdByName("BIT_NO_FAULT_DETECTED"));
}
```

Subtest Parameters

Each diagnostic method includes a void pointer parameter. This parameter allows passing a subtest-specific parameter structure to the diagnostic methods. The parameter structure is optional and, if used, an initialization and free method must be provided to *installBitSubtestEntries()*. See *Example: Subtest Parameter Configuration* for steps on how to configure the optional parameter structure.

If the subtest allows parameters, the allocation for the default parameter structure takes place after **executeBitTests()** is called. The parameter initialization methods are called if the parameter pointer (*testParamPtr* in the *TEST_ENTRY* structure) is set to **NULL**. These are the two methods setup by **installBitSubtestEntries()** during the diagnostic configuration method. The de-allocation of the parameter structure occurs after the test runs. Also, if the user wants to pass in his own parameters for the subtests,

the *testParamPtr* field in the *TEST_ENTRY* structure holds this information. Once **executeBitTests()** is called, the parameter pointer is copied over to an internal MBIT buffer so that the user does not have to keep the pointer around. The parameter structure does not get returned back to the API.

Example: Subtest Parameter Configuration

The steps below outline the method to configure new subtest parameters.

1. Declare the parameter structure used by the diagnostics.

```
typedef struct {
     int length
     int *data
} XXX_LOOPBACK_PARAMS
```

2. Create the method to initialize parameter defaults.

```
BIT_FAULT initLoopbackParams(BIT_SUBTEST subtest,
     void **testParamPtr)
{
     BIT_FAULT faultCode;
     XXX_LOOPBACK_PARAMS *loopParams;
     int defaultLength = 10;

     loopParams = malloc (sizeof
     (XXX_LOOPBACK_PARAMS));

     loopParams.length = defaultLength;
     loopParams.data = malloc (defaultLength * sizeof
     (int));

     for (i = 0; i < defaultLength; i++)
        {
        data[i] = 0xFF00FF00;
        }
     *testParamPtr = loopParams;
     return(
     getBitFaultIdByName("BIT_NO_FAULT_DETECTED"));
}
```

3. Create the method to free parameter defaults. Any memory allocated during initialization is free'd and the parameter pointer is set to **NULL**.

```
BIT_FAULT freeLoopbackParams(
     BIT_SUBTEST subtest,
     void **testParamPtr,)
{
     XXX_LOOPBACK_PARAMS *loopParams =
     (XXX_LOOPBACK_PARAMS *)*testParamPtr;

     free(loopParams->data);
     loopParams->data = NULL;

     free(loopParams);
     loopParams = NULL;

     *testParamPtr = NULL;
     return
     (getBitFaultIdByName("BIT_NO_FAULT_DETECTED"));
}
```

Subtest Configuration

A subtest is integrated using a configuration method that includes the following:

1. *Subtest Addition*
2. *Subtest Installation*
3. *Addition of Subtest-Specific MBIT Faults*

Subtest Addition

Subtests are added by providing a name and description to **addBitSubtestIdent()**. The number of subtests is obtained by calling **getBitNumberOfSubtests()**.

Subtest Installation

Subtest installation is performed with a call to **installBitSubtestEntries()** for each subtest added to the MBIT system. This method associates the subtest methods (that is, installation, execution, and de-installation) with the subtest and sets the default values. See the *Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide* for each subtest's default values.

Addition of Subtest-Specific MBIT Faults

A fault identifier is added by providing a name, description, and fault type to **addBitFaultIdent()**. The fault type is used to specify whether the fault is a system or hardware fault. System faults are *not* saved in the device fault database.

Example: Subtest Configuration

```
faultCode = addBitSubtestIdent ("LOOPBACK_SUBTEST_ONE",
                                "Loopback subtest number one",
                                &loopSubtests[0]);

faultCode = installBitSubtestEntries (loopbackSubtest[0],
                                installLoopbackOneTest,
                                deinstallLoopbackOneTest,
                                runLoopbackOneTest,
                                1,
                                1000,
                                RUN_TILL_COMPLETION,
                                freeLoopbackParams,
                                initLoopbackParams,
                                sizeof(XXX_LOOPBACK_PARAMS));
```

Implementing an MBIT Device Driver

The MBIT system utilizes its own device drivers to increase functionality and enhance operating system (OS) independence. However, MBIT software does not preclude the use of the drivers supplied by a given OS.

MBIT drivers operate independently and do not use the OS supplied I/O system. If a generic driver method is called by a configured subtest, the associated device driver method is located and called. The generic driver method also looks up the device descriptor for the logical device passed in and passes it to the device's driver method.

The **drvInstall()** method must be called before any other generic driver methods can be accessed. After **drvDeinstall()** is called by a configured subtest method, the generic driver methods listed in *Generic Device Driver Interface* can no longer be called.

Note: The generic device driver methods in this section are in the **api/bitGenericDriver.h** header file and are also described later in this section.

Generic Device Driver Interface

The MBIT generic driver interface utilizes subtest and logical device input parameters to select and install the appropriate device driver methods. Such a scheme allows different drivers to be used for different tests on the same device.

Each application's call to a generic driver must provide a subtest and a device. The subtest and device are mapped to an appropriate device descriptor, which is provided to the driver. This re-mapping occurs for all of the generic driver methods. Depending on the application's generic driver call, additional information may be passed to the driver. For instance, in the case of a **read** or **write** operation, the driver receives a pointer to the buffer address, the buffer length, and a pointer for returning the number of bytes that are transferred.

The generic interface allows a single method to use different drivers to test different logical devices. A single driver can be installed to support many separate logical devices. For example, a given driver may support all serial

devices on a given board or separate drivers may be required for some serial devices. Use **installBitDriver()** to specify all of the methods supported by the driver. Refer to *Creating Diagnostic Associations* on page 3-43 on how to associate the driver with supported subtests and devices.

MBIT provides the following generic driver methods:

Method	Description
drvInstall()	Installs a device driver.
drvDeinstall()	Removes an installed device driver.
drvOpen()	Opens a device for I/O operations.
drvClose()	Closes a device.
drvRead()	Reads data from a device.
drvWrite()	Writes data to a device.
drvIoctl()	Controls the operation of a device.

drvInstall()

drvInstall() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

It is suggested that this method be invoked by the install test method for the specified subtest.

Here's a synopsis of the **drvInstall()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvInstall(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device);
```

where *subtest* is the current subtest and *device* is the device to operate on.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

drvDeinstall()

drvDeinstall() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

This method must be invoked by the de-install test method for the specified subtest.

Here's a synopsis of the **drvDeinstall()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvDeinstall(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device);
```

where *subtest* is the current subtest and *device* is the device to operate on.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

drvOpen()

drvOpen() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

Here's a synopsis of the **drvOpen()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvOpen(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device);
```

where *subtest* is the current subtest and *device* is the device to operate on.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

drvClose()

drvClose() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

This method must be called before the **drvDeinstall()** is called.

Here's a synopsis of the **drvClose()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvClose(BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device);
```

where *subtest* is the current subtest and *device* is the device to operate on.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

drvRead()

drvRead() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

This method must be called after the **drvOpen()** has been called.

Here's a synopsis of the **drvRead()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvRead(BIT_SUBTEST subtest,
                              BIT_LOGICAL_DEVICE device,
                              unsigned int bufferSize,
                              char *buffer,
                              unsigned int *bytesRead);
```

where *subtest* is the current subtest, *device* is the device to operate on, *bufferSize* is the size of *buffer* in bytes, *buffer* is the buffer to place data in, and *bytesRead* is a pointer to the number of bytes read.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

drvWrite()

drvWrite() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

This method must be called after **drvOpen()** has been called.

Here's a synopsis of the **drvWrite()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvWrite(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device,
                                unsigned int bufferSize,
                                char *buffer,
                                unsigned int *bytesWritten);
```

where *subtest* is the current subtest, *device* is the device to operate on, *bufferSize* is the number of bytes from *buffer* to write, *buffer* is the buffer to write data from, and *bytesWritten* is the pointer to the number of bytes written.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

drvIoctl()

drvIoctl() finds the associated driver method based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver method is then called.

This method must be called after the **drvOpen()** has been called.

Here's a synopsis of the **drvIoctl()** method:

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvIoctl(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device,
                                int function,
                                int argument);
```

where *subtest* is the current subtest, *device* is the device to operate on, *function* is the driver-specific operation to perform on the device or driver, and *argument* is a driver-specific argument for the *function*.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults*.

Device Driver Interface

The MBIT device driver interface is similar to most device driver implementations. However, there are certain differences that the developer should observe to ensure a successful implementation with the MBIT environment.

To integrate a device driver with the MBIT application, the developer must provide seven driver methods when the driver is configured with *installBitDriver()*. These driver methods provide the capability of communicating between the MBIT application and the device. The driver must provide the standard **install**, **deinstall**, **open**, **close**, **read**, **write**, and **ioctl** methods to implement the necessary interface. There may be cases when a device does not need to implement one of these driver methods. In this case, the developer can stub the driver method by simply providing a method that returns a successful status.

MBIT provides the following driver methods:

Method	Description
devXXXInstall()	Allows MBIT to prepare the driver for subsequent access of the device.
devXXXDeInstall()	Allows the user to terminate the use of the driver.
devXXXOpen()	Allows the user to prepare the device for testing.
devXXXClose()	Allows the user to close the device in preparation for terminating use of the device.
devXXXRead()	Allows the user to read information from the device.
devXXXWrite()	Allows the driver to write information to the device.
devXXXIoctl()	Allows the driver to perform special operations with the device.

dev*XXX*Install()

dev*XXX*Install() allows MBIT to prepare the driver for subsequent access of the device. The driver should save all of the necessary device registers for restoration when the test is complete. This method receives a pointer to the device descriptor of the device being tested. All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

This method is responsible for allocating required resources (that is, buffers, semaphores, etc.) and saving the state of the device. It is also responsible for installing any required interrupt service methods. This method may also disable the device driver supplied by the underlying OS if such a capability is supported.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Install()** method:

```
devXXXInstall(DEV_DESC *pDevDesc);
```

where *pDevDesc* is a pointer to a structure that contains all of the registers that are needed to perform an install operation on the device. Using the

register definitions in the device descriptor guarantees that the correct device is being accessed.

The return values for this method are determined by the developer.

dev*XXX*DeInstall()

dev*XXX*Deinstall() allows the user to terminate the use of the driver. The driver de-installation method is responsible for resource reclamation, restoring the device state, and de-installing the interrupt service methods. The driver should restore all of the device registers that were saved when the driver was installed. This method receives a pointer to a device descriptor of the device being tested. All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Deinstall()** method:

```
devXXXDeinstall(DEV_DESC *pDevDesc)
```

where *pDevDesc* is a pointer to a structure that contains all of the registers that are needed to perform a de-install operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

The return values for this method are determined by the developer.

dev*XXX*Open()

dev*XXX*Open() allows the user to prepare the device for testing. It receives a pointer to the device descriptor of the device being tested. Also, driver variables may be initialized in preparation for subsequent driver use.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Open()** method:

```
devXXXOpen(DEV_DESC *pDevDesc)
```

where *pDevDesc* is a pointer to a structure that contains all of the registers that are needed to perform an open operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

The return values for this method are determined by the developer.

dev*XXX*Close()

dev*XXX*Close() allows the user to close the device in preparation for terminating use of the device. It receives a pointer to the device descriptor of the device being tested.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Close()** method:

```
devXXXClose(DEV_DESC *pDevDesc)
```

where *pDevDesc* is a pointer to a structure that contains all of the registers that are needed to perform a close operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

The return values for this method are determined by the developer.

dev*XXX*Read()

dev*XXX*Read() allows the user to read information from the device. It receives a pointer to the device descriptor of the device being tested, the requested buffer size, a pointer to the buffer address, and a pointer to a variable that holds the number of bytes read. The developer should also update the variable pointed to by the *bytesRead* parameter before returning to the caller.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Read()** method:

```
devXXXRead(DEV_DESC *pDevDesc,
                              UINT32 bufferSize,
                              INT8 *bufferAddr,
                              UINT32 *bytesRead)
```

where *pDevDesc*, *bufferSize*, *bufferAddr* and *bytesRead* are the input parameters.

pDevDesc is a pointer to a structure that contains all of the registers that are needed to perform a read operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

bufferSize contains the number of bytes that the user expects to read from the device. If the device supports transfers wider than a byte, the driver should adjust the count appropriately.

bufferAddr points to the first element of the data buffer that the driver stores the data that is read from the device. The caller must provide a data buffer sufficiently large enough to accept the requested number of bytes defined in the buffer size parameter.

bytesRead points to the variable that the driver returns as the number of bytes read. In the event of an error, the byte count should reflect the actual byte count of the received data.

The return values for this method are determined by the developer.

dev*XXX*Write()

dev*XXX*Write() allows the driver to write information to the device. It receives a pointer to the device descriptor of the device being tested, the requested buffer size, a pointer to the buffer address, and a pointer to a variable that holds the number of bytes written. The developer should also update the variable pointed to by the *bytesWritten* parameter before returning to the caller.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Write()** method:

```
devXXXWrite(DEV_DESC *pDevDesc,
                              UINT32 bufferSize,
                              INT8 *bufferAddr,
                              UINT32 *bytesWritten)
```

where *pDevDesc*, *bufferSize*, *bufferAddr*, and *bytesWritten* are the input parameters.

pDevDesc is a pointer to a structure that contains all of the registers that are needed to perform a write operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

bufferSize contains the number of bytes that the user expects to write to the device. If the device supports transfers wider than a byte, the driver should adjust the count appropriately.

bufferAddr points to the first element of the data buffer that the driver reads data that is to be written to the device. The caller should provide a data buffer sufficiently large enough to reflect the requested number of bytes defined in the *bufferSize* parameter.

bytesWritten points to the variable that the driver returns as the number of bytes written. In the event of an error, the *bytesWritten* variable should reflect the actual byte count of the output data.

The return values for this method are determined by the developer.

dev*XXX*Ioctl()

devXXXIoctl() allows the driver to perform special operations with the device. It receives a pointer to the device descriptor of the device being tested, the requested function, and an argument.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* for more information on this method.

Here's a synopsis of the **dev*XXX*Ioctl()** method:

```
devXXXIoctl(DEV_DESC *pDevDesc,
                              INT32 function,
                              INT32 argument)
```

where *pDevDesc*, *function*, and *argument* are the input parameters.

pDevDesc is a pointer to a structure that contains all of the registers that are needed to perform an ioctl operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

function contains the special operation that is to be performed. The actual function value and implementation is device-specific.

argument contains special information that is required by the method being performed. The actual value of the argument is method- and device-specific. It should be noted that the user is not limited to an integer value as an argument. The argument can be a pointer that is cast as an integer when the method is called. Being a pointer, the user can pass a large amount of information to the ioctl method.

The return values for this method are determined by the developer.

Installing a Device Driver into the MBIT Environment

Before a device driver can be used, it must be installed in the MBIT environment. To accomplish this, the developer must call the method *installBitDriver()*. The driver entry points are the **install**, **deinstall**, **open**, **close**, **read**, **write**, and **ioctl** operations. A device driver may support multiple devices and/or subtests. To associate the devices and subtests that the driver supports, the developer makes a separate call to **createBitTestAssociations()** from within a diagnostic configuration method.

Here's one example of the **installBitDriver()** call:

```
drvPtr = installBitDriver(installFuncPtr, deinstallFuncPtr,
                              openFuncPtr, closeFuncPtr,
                              readFuncPtr, writeFuncPtr,
                              ioctlFuncPtr);
```

Upon successful installation, the returned value will be a pointer to a driver descriptor. A driver descriptor is defined by the *DRV_DESC* structure located in **api/bitGenericDriver.h**. If the install fails, **NULL** is returned. Refer to *Using the Diagnostic Configuration Method* on page 3-43 for more information.

Once configured, the developer may use the generic driver interface to access the diagnostic device driver.

Initializing the Diagnostic Devices

Initialization of diagnostic devices requires adding the device to MBIT and setting up a default device descriptor for the device. A device initialization method is created for each unique device.

A diagnostic may support multiple instances of a device and each instance of a device does not require a separate device initialization method.

Device Initialization Method

The device initialization method is a configuration method passed to **initBit()**. Its purpose is to add the device and setup default device descriptor values.

A device initialization method describes the hardware including register and device types by providing the following categories of information:

- *Device Descriptor Structure (DEV_DESC)*
- *Device Address Table Array (part of DEV_DESC)*
- *Generic Device Address Table Structure (part of DEV_DESC)*
- *Address Type (ADDR_TYPE)*
- *Device Type (DEV_TYPE)*
- *Device Read and Write Utility Methods*

Device Descriptor Structure (*DEV_DESC*)

The device descriptor structure describes the device to MBIT and Table 3-1 provides a description of each field. All types and enumerations are located in **utilities/bitDeviceUtils.h**.

Table 3-1. Device Descriptor (*DEV_DESC*)

Field	Description
logicalDev	The logical device number. This value is a **BIT_LOGICAL_DEVICE** representing the device.
devType	General device descriptor information. Valid values are described in the *DEV_TYPE* structure.
devName	The device class name that the logical device belongs to.
pci	The **devVend** method contains the PCI device and vendor ID, defined by the *BIT_PCI_INFO* structure. The device initialization method must initialize **devVend** for PCI devices. **devVend** should be initialized as (device << 16) | vendor. All other structure elements are initialized internally by MBIT.
reg	A pointer to the address table structure. This array defines each hardware address for a device.
genReg	A pointer to the generic table structure. This structure provides an association between generic register names and specific registers for a particular class of devices. The members of the generic address table are *ADDR_INFO* pointers.
initStat	Device-specific initialization status. Valid values are described in the *INIT_STAT* structure.
totalRegCnt	Total number of device addresses. This is the actual address count, not the number of bytes.
barRegCnt[]	Number of registers per BAR. MBIT supports up to six BARs. The **barRegCnt[]** contains one field for each BAR supported by MBIT (that is, **barRegCnt[0]** – **barRegCnt[5]**). Each field is initialized to the number of hardware addresses for the BAR. If the BAR does not contain any hardware addresses, the **barRegCnt** []should be set to **0**.

Table 3-1. Device Descriptor (*DEV_DESC*) (continued)

Field	Description
intVec	The interrupt vector may be the actual vector value, **BIT_AUTO_CONFIG** or **BIT_NONE_CONFIG**. **BIT_AUTO_CONFIG** causes the device descriptor initialization to assign a vector at run time. **BIT_NONE_CONFIG** indicates that the device does not use interrupts.
intLvl	The interrupt level may be the actual level value, **BIT_AUTO_CONFIG** or **BIT_NONE_CONFIG**. **BIT_AUTO_CONFIG** and **BIT_NONE_CONFIG** behave similarly to **IntVec** above.
baseAddr	For PCI devices, **BIT_AUTO_BASE_ADDR** causes the base address to be configured at run time.
The first three items listed below are method pointers to methods that enable, disable, or check enable status for the device. Devices that must be enabled before reading or writing require these methods. If the device does not need to be enabled, set each method pointer to **NULL**.	
isEnabled	A method to check if the device is enabled. The method should return **TRUE** if enabled, **FALSE** otherwise.
enableDev	A method to enable the device. This method is responsible for enabling read and/or write accesses to the device.
disableDev	A method to disable the device. This method is responsible for disabling read and/or write accesses to the device.
data	Device instance data. This is a general-purpose element that may be used by the developer to contain additional device-specific data.
The following are method pointers for reading and writing registers (refer to *Device Read and Write Utility Methods* on page 3-33). Utility Method Parameters: Parameter 1: 32-bit address to access. Parameter 2: (Output) Data to write to the address. (Input) Address to store the data read from Parameter 1 above.	
inReg8	A method to read an 8-bit address.

Table 3-1. Device Descriptor (*DEV_DESC*) (continued)

Field	Description
outReg8	A method to write an 8-bit address.
inReg16	A method to read a 16-bit address.
outReg16	A method to write a 16-bit address.
inReg32	A method to read a 32-bit address.
outReg32	A method to write a 32-bit address.

Device Address Table Array (part of *DEV_DESC*)

The address table array defines each hardware address for a device. The hardware addresses are the complete addresses, which are calculated by adding the address offset with the base address of the device.

The address information structure contains the information for one particular device hardware address. The structure is used by **bitIn()**/**bitOut()** (see Chapter 4, *Utility Methods*) when accessing a

hardware address. In addition, it provides a means to keep track of bits toggled during read and/or write accesses to the hardware address.

Table 3-2. Address Information (ADDR_INFO)

Field	Description
addr	A hardware address on a device. The address is calculated by adding the device base address with the offset of the hardware address (refer to the *Device Address Table Array (part of DEV_DESC)*).
mask	A mask used to mark read/write bits. A mask of 0x0F for an 8-bit hardware address indicates the four high-order bits are read-only.
val	A value used to store data read from or written to a hardware address. To write the data to a hardware address, assign the data to this field and then use **bitOut()** to write the value.
save	The location to allow a developer to save a value read from the hardware address.
size	The number of addressable bytes at the hardware address.
type	A type of hardware address (refer to *Address Type (ADDR_TYPE)*).
dev	An integer ID representing the device associated with the hardware address. This must match **logicalDev** from the *DEV_DESC* containing the *ADDR_INFO*.
changes	A structure keeps track of bits toggled when accessing the hardware address through **bitIn()**/**bitOut()**.

If a device's addressable memory is not continuous, the offset is multiplied by an address interval to get to the address needed (for example, address = base address + (offset * address interval).

Devices that support auto configuration of the base address, specified by **BIT_AUTO_BASE_ADDR**, will only require the offset for each hardware address. Run time initialization code will calculate the complete base address by adding the address offset to the base address for each hardware address. PCI devices are the only devices that support **BIT_AUTO_BASE_ADDR**.

Devices that map different register sets using different base addresses may require several address tables. Each address table for the device will be combined into the same 'C' array and only the names and address calculations will differ. This is specifically useful for PCI devices that map different registers using different PCI base address registers (that is, BAR0, BAR1, etc.).

Below is an example of the device address table 'C' structure.

```
typedef struct
{
     // BAR 0 (I/O mapped register set)
     ADDR_INFO ioscntl0;
     ADDR_INFO ioscntl1;
     ADDR_INFO ioscid;
     ADDR_INFO iosxfer;
     ...
     // BAR 1 (MEMORY mapped register set)
     ADDR_INFO memscntl0;
     ADDR_INFO memscntl1;
     ADDR_INFO memscid;
     ADDR_INFO memsxfer;
     ...
     // BAR 2 (SCRIPTS RAM)
     ADDR_INFO scriptsRam;
} BIT_SYM895A_REG_T;
```

This structure is accessed through the device descriptor member **reg**. The members of the address table are *ADDR_INFO* structures.

The device address table structure must remain accessible throughout the execution of MBIT. This structure is *not* free'd by MBIT during termination.

Generic Device Address Table Structure (part of *DEV_DESC*)

The generic device address table structure provides a method of associating generic register names with specific registers for a particular class of devices. For example, all memory controllers have registers that define and enable banks of SDRAM. However, there is no standard for these registers. By using a generic register name and generic register bit masks, software can be written to determine if a bank is enabled, and the size of the bank.

Example: Generic Device Address Table 'C' Structure

Below is an example of the generic device address table 'C' structure.

```
typedef struct
{
     ADDR_INFO* allRam;
     ADDR_INFO* ramBankA;
     ADDR_INFO* ramBankB;
     ADDR_INFO* ramBankC;
     ADDR_INFO* ramBankD;
     ADDR_INFO* ramBankE;
     ADDR_INFO* ramBankF;
     ADDR_INFO* ramBankG;
     ADDR_INFO* ramBankH;
} BIT_RAM_GENREG_T;
```

This structure is accessed through the device descriptor member **genReg**. The members of the generic address table are *ADDR_INFO* structure pointers.

The generic register references *must* point to hardware addresses with the same format for all devices the developer designs the code to support. Also, the generic device address table structure *must* be allocated by the device initialization method. This structure is free'd by MBIT during termination.

Address Type (ADDR_TYPE)

The address type provides MBIT with additional information about the hardware address. This type may indicate a register, memory, or a serial ROM hardware address. In addition, the behavior (that is, readable, writeable, etc.) of the hardware address may be described.

Table 3-3. Address Type (ADDR_TYPE)

Field	Description
Register Hardware Address Types	
REG_RD_ONLY	Register read-only
REG_WR_ONLY	Register write-only
REG_RDWR	Register read/write
REG_RD_ONLY_CLRS_BITS	Register read-only, reading clears bits
REG_WR_ONLY_CLRS_BITS	Register write-only, writing clears bits
REG_RDWR_RD_CLRS_BITS	Register read/write, reading clears bits
REG_RDWR_WR_CLRS_BITS	Register read/write, writing clears bits
REG_RDWR_CLRS_BITS	Register read/write, reading or writing clears bits
REG_RD_SIDE_EFFECT	Register read-only, reading adversely effects device
REG_WR_SIDE_EFFECT	Register write-only, writing adversely effects device
REG_RDWR_SIDE_EFFECT	Register read/write, reading or writing adversely effects device
REG_RD_VISIBLE_TEST	Register read-only, location designated for read visibility testing
REG_WR_VISIBLE_TEST	Register write-only, location designated for write visibility testing
REG_RDWR_VISIBLE_TEST	Register read/write, location designated for read/write visibility testing
Memory Hardware Address Types	
MEM_RD_ONLY	Memory read-only
MEM_WR_ONLY	Memory write-only

Table 3-3. Address Type (ADDR_TYPE) (continued)

Field	Description
MEM_RDWR	Memory read/write
MEM_RD_VISIBLE_TEST	Memory read-only, location designated for read visibility testing
MEM_WR_VISIBLE_TEST	Memory write-only, location designated for write visibility testing
MEM_RDWR_VISIBLE_TEST	Memory read/write, location designated for read/write visibility testing
Serial ROM Hardware Address Types	
SROM_RD_ONLY	Serial ROM read-only
SROM_WR_ONLY	Serial ROM write-only
SROM_RDWR_ONLY	Serial ROM read/write
SROM_RD_VISIBLE_TEST	Serial ROM read-only, location designated for read visibility testing
SROM_WR_VISIBLE_TEST	Serial ROM write-only, location designated for write visibility testing
SROM_RDWR_VISIBLE_TEST	Serial ROM read/write, location designated for read/write visibility testing

Device Type (*DEV_TYPE*)

The device type provides MBIT with a general description of the device. MBIT currently supports PCI, ISA, memory, and general device types. PCI device hardware addresses are configured internally by MBIT during initialization if *baseAddr* is set to **BIT_AUTO_BASE_ADDR**. MBIT does not perform any additional device type configurations for other supported device types.

Table 3-4. Device Type (*DEV_TYPE*)

Field	Description
DEV_TYPE_NONE	Device type invalid.
DEV_TYPE_PCI	Device type PCI.
DEV_TYPE_ISA	Device type ISA.
DEV_TYPE_MEM	Device type memory.
DEV_TYPE_GEN	Device type general.

Device Read and Write Utility Methods

The device read and write utility methods provided by MBIT are added to the device descriptor during device initialization. These are then used by **bitIn()/bitOut()** to access the device hardware addresses.

MBIT provides the following device read and write utility methods:

Method	Description
bitProbeIn8/16/32()	Reads 8/16/32-bit data from the designated address.
bitProbeOut8/16/32()	Writes 8/16/32-bit data to the designated address.
bitProbeInSwap16/32()	Reads and byte swaps 16/32-bit data from the designated address.
bitProbeOutSwap16/32()	Writes and byte swaps 32-bit data to the designated address.

Method	Description
bitIn8/16/32()	Reads 8/16/32-bit data from the designated address.
bitOut8/16/32()	Writes 8/16/32-bit data to the designated address.
bitInSwap16/32()	Reads and byte swaps 16/32-bit data from the designated address.
bitOutSwap16/32()	Writes and byte swaps 16/32-bit data to the designated address.
bitPciWrite32()	Writes 32-bit data to PCI (I/O or memory) space in little-endian mode.
bitPciRead32()	Reads 32-bit data from PCI (I/O or memory) space.

bitProbeIn8/16/32()

bitProbeIn8/16/32() reads 8/16/32-bit data from the designated address.

The data read is written into a 8/16/32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()/bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitIn8()/bitIn16()/bitIn32()** should be used.

Here's a synopsis of the **bitProbeIn8/16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeIn8(ULONG addr, UINT32 *pdata);
STATUS bitProbeIn16(ULONG addr, UINT32 *pdata);
STATUS bitProbeIn32(ULONG addr, UINT32 *pdata);
```

where *addr* is the address to read data from and *pdata* is the pointer to the 32-bit location to store data.

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitProbeOut8/16/32()

bitProbeOut8/16/32() writes 8/16/32-bit data to the designated address.

The data is written into an 8/16/32-bit memory location. The MBIT exception handler is enabled during the data write. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitOut8()/bitOut16()/bitOut32()** should be used.

Here's a synopsis of the **bitProbeOut8/16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeOut8(ULONG addr, UINT8 *data);
STATUS bitProbeOut16(ULONG addr, UINT16 *data);
STATUS bitProbeOut32(ULONG addr, UINT32 *data);
```

where *addr* is the address to write data to and *data* is the data to write out.

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitProbeInSwap16/32()

bitProbeInSwap16/32() reads and byte swaps 16/32-bit data from the designated address. The data is read as little/big-endian and loaded as big/little-endian into a 32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()/bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitInSwap16()/bitInSwap32()** should be used.

Here's a synopsis of the **bitProbeInSwap16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeInSwap16(ULONG addr, UINT32 *pdata);
STATUS bitProbeInSwap32(ULONG addr, UINT32 *pdata);
```

where *addr* is the address to read data from and *pdata* is the pointer to a 32-bit location to store data.

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitProbeOutSwap16/32()

bitProbeOutSwap16/32() writes and byte swaps 16/32-bit data to the designated address. The big/little-endian data is written into a 16/32-bit data location as little/big-endian.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitOutSwap16()/bitOutSwap32()** should be used.

Here's a synopsis of the **bitProbeOutSwap16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeOutSwap16(ULONG addr, UINT16 *data);
STATUS bitProbeOutSwap32(ULONG addr, UINT32 *data);
```

where *addr* is the address to write data to and *data* is the data to write out.

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitIn8/16/32()

bitIn8/16/32() reads 8/16/32-bit data from the designated address. The data read is written into a 32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()**/**bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is *not* enabled during the data read. If the MBIT exception handler is needed to "catch" exceptions caused by the access to the device, then **bitProbeIn8()**/**bitProbeIn16()**/**bitProbeIn32()** should be used.

Here's a synopsis of the **bitIn8/16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitIn8(ULONG addr, UINT32 *pdata);
STATUS bitIn16(ULONG addr, UINT32 *pdata);
STATUS bitIn32(ULONG addr, UINT32 *pdata);
```

where *addr* is the address to read data from and *pdata* is the pointer to a 32-bit location to store data.

This method always returns **0**.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitOut8/16/32()

bitOut8/16/32() writes 8/16/32-bit data to the designated address. The data is written into an 8/16/32-bit memory location.

The MBIT exception handler is *not* enabled during the data write. If the MBIT exception handler is needed to "catch" exceptions caused by the access to the device, then **bitProbeOut8()**/**bitProbeOut16()**/**bitProbeOut32()** should be used.

Here's a synopsis of the **bitOut8/16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitOut8(ULONG addr, UINT8 *data);
STATUS bitOut16(ULONG addr, UINT16 *data);
STATUS bitOut32(ULONG addr, UINT32 *data);
```

where *addr* is the address to write data to and *data* is the data to write out.

This method always returns **0**.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitInSwap16/32()

bitInSwap16/32() reads and byte swaps 16/32-bit data from the designated address. The data is read as little/big-endian and loaded as big/little-endian into a 32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()**/**bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is *not* enabled during the data read. If the MBIT exception handler is needed to "catch" exceptions caused by the access to the device, then **bitProbeInSwap16()**/**bitProbeInSwap32()** should be used.

Here's a synopsis of the **bitInSwap16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitInSwap16(ULONG addr, UINT32 *pdata);
STATUS bitInSwap32(ULONG addr, UINT32 *pdata);
```

where *addr* is the address to read data from and *pdata* is the pointer to a 32-bit location to store data.

This method always returns **0**.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitOutSwap16/32()

bitOutSwap16/32() writes and byte swaps 16/32-bit data to the designated address. The big/little-endian data is written into a 16/32-bit data location as little/big-endian.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitProbeOutSwap16()**/**bitProbeOutSwap32()** should be used.

Here's a synopsis of the **bitOutSwap16/32()** method:

```
#include <utilities/bitDeviceUtils.h>
STATUS bitOutSwap16(ULONG addr, UINT16 *data);
STATUS bitOutSwap32(ULONG addr, UINT32 *data);
```

where *addr* is the address to write data to and *data* is the data to write out.

This method always returns **0**.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitPciWrite32()

bitPciWrite32() writes 32-bit data to PCI (I/O or memory) space in little-endian mode.

Here's a synopsis of the **bitPciWrite32()** method:

```
#include <utilities/bitDeviceUtils.h>
void bitPciWrite32(ULONG addr, UINT32 data);
```

where *addr* is the PCI address to write data to and *data* is the data to write out.

This method has no return values.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitPciRead32()

bitPciRead32() reads 32-bit data from PCI (I/O or memory) space.

Here's a synopsis of the **bitPciRead32()** method:

```
#include <utilities/bitDeviceUtils.h>
void bitPciRead32(ULONG addr, UINT32 *pdata);
```

where *addr* is the PCI address to read data from and *pdata* is the pointer to store data to.

This method has no return values.

Refer to Appendix B, *Integrating Custom Diagnostics' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

Creating a Device Initialization Method

This section provides an example of the required steps to create a device initialization method. The method is responsible for adding the device to the device list and setting device default information.

The method begins by making a call to **addBitDeviceIdent()** to add the device. A call to the **getDevTablePtr()** method is then made to get a pointer to the device descriptor associated with the added device. The device descriptor is used to fill in all the data for that particular device. Refer to *Device Descriptor Structure (DEV_DESC)* on page 3-25 for a description of the device descriptor structure.

The following is an outline of a device initialization method for a device on an MVME51*xx* board. The device being added is the system memory controller (SMC).

```
BIT_FAULT initMemoryController(void)
{
BIT_FAULT faultCode;
BIT_LOGICAL_DEVICE memDev; // Used to catch the return
                           from addBitDeviceIdent().
DEV_DESC *pDevDesc;       // Device descriptor pointer.
```

1. (Optional) Declare and allocate the generic device address table structure (structure is free'd during termination of MBIT).

   ```
   BIT_SMC_GENREG_T *genReg = malloc
                              (sizeof(BIT_SMC_GENREG_T));
   ```

2. Declare a static device address table structure.

   ```
   static BIT_HAWK_SMC_REG_T smc;
   ```

3. Add the device to the device list.

   ```
   faultCode =
   addBitDeviceIdent("BIT_MEMORY_CONTROLLER",
                              "Memory Controller",
                              &memDev);
   ```

4. Set the logical device to return from **addBitDeviceIdent()**.

   ```
   pDevDesc->logicalDev = memDev;
   ```

5. Get the device descriptor for this device.

   ```
   pDevDesc = getDevTablePtr(memIdent);
   ```

6. Set the default initialization status.

   ```
   pDevDesc->initStat = INIT_NOT_INITIALIZED;
   ```

7. Set the default 8-, 16-, and 32-bit read and write methods.

   ```
   pDevDesc->inReg8 = (INREG8_FUNCPTR) bitProbeIn8;
   pDevDesc->outReg8 = (OUTREG8_FUNCPTR) bitProbeOut8;
   pDevDesc->inReg16 = (INREG16_FUNCPTR) bitProbeIn16;
   pDevDesc->outReg16 = (OUTREG16_FUNCPTR) bitProbeOut16;
   pDevDesc->inReg32 = (INREG32_FUNCPTR) bitProbeIn32;
   pDevDesc->outReg32 = (OUTREG32_FUNCPTR) bitProbeOut32;
   ```

8. Set the device type.

```
pDevDesc->devType = DEV_TYPE_GEN;
```

9. Set the device class.

```
pDevDesc->devName = "HAWK_SMC";
```

10. Set the interrupt level and vector.

```
pDevDesc->intVec = BIT_NONE_CONFIG;
pDevDesc->intLvl = BIT_NONE_CONFIG;
pDevDesc->baseAddr = HAWK_SMC_BASE_ADDR;
```

11. Setup the generic hardware addresses, if applicable.

```
genReg->sdramEnSzA = &smc.sdramEnSzA;
genReg->sdramEnSzB = &smc.sdramEnSzB;
genReg->sdramEnSzC = &smc.sdramEnSzC;
genReg->sdramEnSzD = &smc.sdramEnSzD;
genReg->sdramEnSzE = &smc.sdramEnSzE;
genReg->sdramEnSzF = &smc.sdramEnSzF;
```

...etc...

12. Setup the hardware addresses.

```
smc.venDevId.addr = pDevDesc->baseAddr +
                      (0x00*addrInterval);
smc.venDevId.mask = 0x00000000;
smc.venDevId.size = 4;
smc.venDevId.type = REG_RD_ONLY;
smc.venDevId.dev = memDev;

smc.revIdGcr.addr = pDevDesc->baseAddr +
                      (0x08*addrInterval);
smc.revIdGcr.mask = 0x01000100;
smc.revIdGcr.size = 4;
smc.revIdGcr.type = REG_RDWR;
smc.revIdGcr.dev = memDev;
```

...etc...

13. Assign the pointer to the address table and generic address table structures in the device descriptor. The structures must be cast to void pointers.

```
pDevDesc->genReg = (void*)genReg;
pDevDesc->reg = (void*)&smc;
```

14. (Non-PCI devices only) If setup was successful, indicate device descriptor initialization is complete.

```
pDevDesc->initStat = INIT_OK;

return (faultCode);
} /* end of the routine */
```


Creating Diagnostic Associations

To bring the elements (that is, subtests, devices, and drivers) of a diagnostic together, an association must be made between the related elements. The elements of a diagnostic are associated by invoking the method **createBitTestAssociations()**. The method creates an association for each subtest and device provided. If a driver is also provided, each subtest and device association will include an association to the driver.

With multiple subtests and devices, several calls to **createBitTestAssociations()** may be required to achieve the desired result. For example, if a particular device is not supported by all subtests, an additional call to **createBitTestAssociations()** is required. If, for example, a driver supports loopback on two logical devices but only supports modem controls on one, multiple calls to **createBitTestAssociations()** may be required. Otherwise, the API may indicate a subtest requiring modem controls can be run on a device that does not support them. A properly configured system should return an error indicating that the test was not supported.

Calls to the **createBitTestAssociations()** method are placed in the diagnostic configuration method (refer to *Using the Diagnostic Configuration Method*).

Using the Diagnostic Configuration Method

A diagnostic is configured into MBIT using a diagnostic configuration method. The configuration method contains all of the necessary calls to integrate custom diagnostics. This configuration method is passed to **initBit()** and then called during MBIT initialization.

The following steps demonstrate the methods involved in configuring diagnostics and the suggested order of their use.

1. Make a call to the **addBitFaultIdent()** method for each subtest-specific MBIT fault.
2. Make a call to the **addBitSubtestIdent()** method for each subtest being configured. The subtest ID returned identifies the subtest to all methods with a **BIT_SUBTEST** parameter.
3. Make a call to **installBitSubtestEntries()** for each subtest that was added. This method uses the subtest ID returned by **addBitSubtestIdent()** as its **BIT_SUBTEST** parameter.
4. Make a call to **installBitDrivers()** and pass the driver entry points for the driver being installed. This method returns a driver descriptor pointer (*DRV_DESC**), which is used when creating an association to a driver with the method **createBitTestAssociations()**.
5. Make a call to **getBitDeviceIdByName()** for each device that is associated with the installed subtests. Use the return value from each call to **getBitDeviceIdByName()** to provide as a parameter to the method **createBitTestAssociations()**.
6. Make a call to **createBitTestAssociations()** using the subtest IDs, device IDs, and a driver descriptor pointer. This call may be made several times to create different associations for specific diagnostic configurations.

Example: Diagnostic Configuration Method

The steps below show an example of the diagnostic configuration method. This example configures a diagnostic with two subtests, two fault, one device, and one driver.

```
BIT_FAULT xxxLoopbackConfigTest()
{
```

1. Declare and initialize local variables.

```
BIT_FAULT faultCode = NO_FAULT_DETECTED;

int loopbackSubtests[2];
int loopbackDevices[1];
int loopbackFaults[2];

DRV_DESC *pDrvDesc;
```

2. Add the fault and subtest identifiers.

```
faultCode = addBitFaultIdent("LOOPBACK_FAULT_ONE",
              "Loopback fault number one",
              BIT_HARDWARE_FAULT,
              &loopbackFaults[0]);

faultCode = addBitFaultIdent("LOOPBACK_FAULT_TWO",
              "Loopback fault number two",
              BIT_HARDWARE_FAULT,
              &loopbackFaults[1]);

faultCode = addBitSubtestIdent ("LOOPBACK_SUBTEST_ONE",
              "Loopback subtest number one",
              &loopSubtests[0]);

faultCode = addBitSubtestIdent ("LOOPBACK_SUBTEST_TWO",
              "Loopback subtest number two",
              &loopSubtests[1]);
```

3. Install each of the subtests.

```
faultCode = installBitSubtestEntries (loopbackSubtest[0],
              installLoopbackOneTest,
              deinstallLoopbackOneTest,
              runLoopbackOneTest,
              1,
              1000,
              RUN_TILL_COMPLETION,
              freeLoopbackParams,
              initLoopbackParams,
              sizeof(XXX_LOOPBACK_PARAMS));

faultCode = installBitSubtestEntries (loopbackSubtest[1],
              installLoobackTwoTest,
              deinstallLoopbackTwoTest,
              runLoopbackTwoTest,
              1,
              5000,
              HALT_ON_ERROR,
              freeLoopbackParams,
              initLoopbackParams,
              sizeof(XXX_LOOPBACK_PARAMS);
```

4. Install the diagnostic device drivers.

```
pDrvDesc = installBitDriver(loopbackDrvInstall,
              loopbackDrvDeinstall,
              loopbackDrvOpen,
              loopbackDrvClose,
              loopbackDrvRead,
              loopbackDrvWrite,
              loopbackDrvIoctl);
```

5. Get the device IDs to use when creating an association to the device.

```
loopbackDevices[0] =
   getBitDeviceIdByName("LOOPBACK_DEVICE_ONE");
```

6. Create an association between the subtests, devices, and diagnostic device drivers.

```
status = createBitTestAssociations(loopbackSubtests,
              2,
              loopbackDevices,
              1,
              pDrvDesc)
return (status);
}
```

Utility Methods 4

Introduction

MBIT provides methods to complete various activities while integrating custom diagnostics. These methods are grouped into the following categories:

Refer to Appendix C, *Utility Methods' Reference Pages* for more information on these methods.

Cache Utility Methods

MBIT provides the following L1 and L2 cache utility methods. The header file, **utilities/bitCacheUtils.h**, provides the prototypes for these methods.

Method	Description
bitDataCacheEnable()	Invalidates and then enables the L1 data cache.
bitDataCacheDisable()	Flushes and then disables the L1 data cache.
bitDataCacheIsEnabled()	Returns the boolean enable state of L1 data cache.
bitDataCacheFlush()	Flushes the entire L1 data cache.
bitDataCacheFlushInvalidate()	Flushes and invalidates the entire L1 data cache.
bitDataCacheInvalidate()	Invalidates the L1 data cache for a range of memory.
bitDataCacheLock()	Locks the L1 data cache.
bitDataCacheUnlock()	Unlocks the L1 data cache.

Method	Description
bitInstCacheEnable()	Invalidates and then enables the L1 instruction cache.
bitInstCacheDisable()	Disables the L1 instruction cache.
bitInstCacheIsEnabled()	Returns the boolean enable state of the L1 instruction cache.
bitInstCacheLock()	Locks the L1 instruction cache.
bitInstCacheUnlock()	Unlocks the L1 instruction cache.
bitL2CacheSizeGet()	Returns the size of the L2 cache configured by the hardware.
bitL2CacheEnable()	Enables the L2 cache.
bitL2CacheDisable()	Flushes, invalidates and then disables the L2 cache.
bitL2CacheOn()	Enables the L2 cache without any flushing or invalidation.
bitL2CacheOff()	Disables the L2 cache without any flushing or invalidation.
bitL2CacheIsEnabled()	Returns the boolean enable state of the L2 cache.
bitL2CacheFlush()	Flushes the entire L2 cache.
bitL2CacheFlushInvalidate()	Flushes and invalidates the entire L2 cache.
bitL2CacheInvalidate()	Invalidates the entire L2 cache.
bitL2CacheLock()	Locks the L2 cache if the L2 cache supports it.
bitL2CacheUnlock()	Unlocks the L2 cache if the L2 cache supports it.
bitL2CacheIsLockable()	Returns the boolean lock capability of the L2 cache.
bitL2CacheFill()	Fills the specified number of 32-bit words in the L2 cache with the specified pattern.
bitL2CacheIsWritebackCapable()	Returns the boolean write-back capability of the L2 cache.

bitDataCacheEnable()

This method invalidates and then enables the L1 data cache.

Here's a synopsis of the **bitDataCacheEnable()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheEnable(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitDataCacheDisable()

This method flushes and then disables the L1 data cache.

Here's a synopsis of the **bitDataCacheDisable()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheDisable(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitDataCacheIsEnabled()

This method returns the boolean enable state of the L1 data cache.

Here's a synopsis of the **bitDataCacheIsEnabled()** method:

```
<utilities/bitCacheUtils.h>
BOOL bitDataCacheIsEnabled(void)
```

Upon successful completion, this method returns **TRUE** if the L1 data cache is enabled or **FALSE** if the L1 data cache is *not* enabled.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheFlush()

This method flushes the entire L1 data cache.

Here's a synopsis of the **bitDataCacheFlush()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheFlush(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitDataCacheFlushInvalidate()

This method flushes and invalidates the entire L1 data cache.

Here's a synopsis of the **bitDataCacheFlushInvalidate()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheFlushInvalidate(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitDataCacheInvalidate()

This method invalidates the L1 data cache for a range of memory.

Here's a synopsis of the **bitDataCacheInvalidate()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheInvalidate(void *address, UINT bytes)
```

where *address* is the virtual address to begin invalidation and *bytes* is the number of bytes to invalidate.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitDataCacheLock()

This method locks the L1 data cache.

Here's a synopsis of the **bitDataCacheLock()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheLock(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitDataCacheUnlock()

This method unlocks the L1 data cache.

Here's a synopsis of the **bitDataCacheUnlock()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheUnlock(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitInstCacheEnable()

This method invalidates and then enables the L1 instruction cache.

Here's a synopsis of the **bitInstCacheEnable()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheEnable(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitInstCacheDisable()

This method disables the L1 instruction cache.

Here's a synopsis of the **bitInstCacheDisable()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheDisable(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitInstCacheIsEnabled()

This method returns the boolean enable state of the L1 instruction cache.

Here's a synopsis of the **bitInstCacheIsEnabled()** method:

```
<utilities/bitCacheUtils.h>
BOOL bitInstCacheIsEnabled(void)
```

Upon successful completion, this method returns **TRUE** if the L1 instruction cache is enabled or **FALSE** if the L1 instruction cache is *not* enabled.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInstCacheLock()

This method locks the L1 instruction cache.

Here's a synopsis of the **bitInstCacheLock()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheLock(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitInstCacheUnlock()

This method unlocks the L1 instruction cache.

Here's a synopsis of the **bitInstCacheUnlock()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheUnlock(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheSizeGet()

This method returns the size (in bytes) of the L2 cache configured by the hardware (the L2 cache controller).

Here's a synopsis of the **bitL2CacheSizeGet()** method:

```
<utilities/bitCacheUtils.h>
UINT bitL2CacheSizeGet(void)
```

Upon successful completion, this method returns *numBytes* (L2 cache size in bytes) or **0** if the size cannot be determined.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheEnable()

This method enables the L2 cache.

Here's a synopsis of the **bitL2CacheEnable()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheEnable(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheDisable()

This method flushes, invalidates and then disables the L2 cache. If a **NULL** buffer is provided, then a local buffer is allocated for use and free'd before return. If the size of a given non-**NULL** buffer is not equal to twice the L2 cache size or is not naturally aligned, then no action is taken by the method.

Here's a synopsis of the **bitL2CacheDisable()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheDisable(void *pFlushBuffer, int buffSize)
```

where *pFlushBuffer* and *buffSize* are the input parameters.

pFlushBuffer is a pointer to a cacheable memory block twice the size of the L2 cache and aligned naturally. If **NULL**, a local buffer is used and *buffSize* is ignored. The local buffer will not be guaranteed to be cacheable if BATs or page tables have been altered prior to calling.

buffSize is the size of the flush buffer in bytes. It must be equal to twice the L2 cache size.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheOn()

This method enables the L2 cache without any flushing or invalidation.

Here's a synopsis of the **bitL2CacheOn()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheOn(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheOff()

This method disables the L2 cache without any flushing or invalidation.

Here's a synopsis of the **bitL2CacheOff()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheOff(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheIsEnabled()

This method returns the boolean enable state of the L2 cache.

Here's a synopsis of the **bitL2CacheIsEnabled()** method:

```
<utilities/bitCacheUtils.h>
BOOL bitL2CacheIsEnabled(void)
```

Upon successful completion, this method returns **TRUE** if the L2 cache is enabled or **FALSE** if the L2 cache is *not* enabled.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheFlush()

This method flushes the entire L2 cache. If a **NULL** buffer is provided, then a local buffer is allocated for use and free'd before return. If the size of a given non-**NULL** buffer is not equal to twice the L2 cache size or is not naturally aligned, then no action is taken by the method.

Here's a synopsis of the **bitL2CacheFlush()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheFlush(void pFlushBuffer, int buffSize)
```

where *pFlushBuffer* and *buffSize* are the input parameters.

pFlushBuffer is a pointer to a cacheable memory block twice the size of the L2 cache and aligned naturally. If **NULL**, a local buffer is used and *buffSize* is ignored. The local buffer will not be guaranteed to be cacheable if BATs or page tables have been altered prior to calling.

buffSize is the size of the flush buffer in bytes. It must be equal to twice the L2 cache size.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheFlushInvalidate()

This method flushes and invalidates the entire L2 cache.

Here's a synopsis of the **bitL2CacheFlushInvalidate()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheFlushInvalidate(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheInvalidate()

This method invalidates the entire L2 cache. Any modified data in the L2 cache is lost unless it is flushed first.

Here's a synopsis of the **bitL2CacheInvalidate()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheInvalidate(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheLock()

This method locks the L2 cache if the L2 cache supports it.

Here's a synopsis of the **bitL2CacheLock()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheLock(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheUnlock()

This method unlocks the L2 cache if the L2 cache supports it.

Here's a synopsis of the **bitL2CacheUnlock()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheUnlock(void)
```

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheIsLockable()

This method returns the boolean lock capability of the L2 cache.

Here's a synopsis of the **bitL2CacheIsLockable()** method:

```
<utilities/bitCacheUtils.h>
BOOL bitL2CacheIsLockable(void)
```

Upon successful completion, this method returns **TRUE** if the L2 cache is lockable or **FALSE** if the L2 cache is *not* lockable.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheFill()

This method fills the specified number of 32-bit words in the L2 cache with the specified pattern. The pattern is incremented by the *modifier* after every write.

Here's a synopsis of the **bitL2CacheFill()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheFill(UINT *bufPtr, UINT *castOutBuf,
                         UINT wordCount,
                         UINT pattern,
                         int modifier)
```

where *bufPtr* is a pointer to a cacheable buffer to fill, *castOutBuf* is a pointer to a cacheable buffer to fill that causes data to be cast-out from the L1 data cache, *wordCount* is the number of 32-bit words to fill, *pattern* is the pattern to fill the buffer with, and *modifier* is the value with which to modify the pattern after each write to the buffer.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitL2CacheIsWritebackCapable()

This method returns the boolean write-back capability of the L2 cache.

Here's a synopsis of the **bitL2CacheIsWritebackCapable()** method:

```
<utilities/bitCacheUtils.h>
BOOL bitL2CacheIsWritebackCapable(void)
```

Upon successful completion, this method returns **TRUE** if the L2 cache supports write-back or **FALSE** if the L2 cache *does not* support write-back.

Refer to Chapter 5, *MBIT Faults* for more faults.

Diagnostic Device Utility Methods

MBIT provides the following methods to obtain device descriptors, access devices, and monitor what each device accesses. The utility methods **bitIn()** and **bitOut()** may read/write hardware locations defined by *ADDR_INFO* structures. The header file **utilities/bitDeviceUtils.h** provides the prototypes for these methods.

Method	Description
getDeviceDescriptor()	Takes a logical device number and returns a pointer to the device descriptor.
getDevTablePtr()	Takes a logical device number and returns a pointer to the device descriptor.
bitTrackChanges()	Starts or stops tracking register bit changes during hardware access.
bitIn()	Reads from the location described by the *ADDR_INFO* structure passed to the method.
bitOut()	Writes to the location described by the *ADDR_INFO* structure passed to the method.

getDeviceDescriptor()

This method takes a logical device number and returns a pointer to the device descriptor. The device descriptor contains all the information needed to interface with the device.

Here's a synopsis of the **getDeviceDescriptor()** method:

```
<utilities/bitCacheUtils.h>
DEV_DESC* getDeviceDescriptor(BIT_LOGICAL_DEVICE device)
```

where *device* is the logical device to retrieve.

Upon successful completion, this method returns **DEV_DESC** (a pointer to the device descriptor) or **NULL** if the device descriptor is invalid.

Refer to Chapter 5, *MBIT Faults* for more faults.

getDevTablePtr()

This method takes a logical device number and returns a pointer to the device descriptor. The device descriptor contains all the information needed to interface with the device. This method should only be used by methods initializing device descriptors.

Here's a synopsis of the **getDevTablePtr()** method:

```
<utilities/bitCacheUtils.h>
DEV_DESC* getDevTablePtr(BIT_LOGICAL_DEVICE device)
```

where *device* is the logical device to retrieve.

Upon successful completion, this method returns *DEV_DESC* (a pointer to the device descriptor) or **NULL** if the device descriptor is invalid.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitTrackChanges()

This method starts or stops tracking register bit changes during hardware access.

Here's a synopsis of the **bitTrackChanges()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitTrackChanges(ADDR_INFO *reg, UINT on)
```

where *reg* is the pointer to the location's *ADDR_INFO* structure and *on* is a boolean value to indicate starting or stopping change tracking. **TRUE** starts tracking, **FALSE** stops tracking.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitIn()

This method reads from the location described by the *ADDR_INFO* structure passed to the method. If an exception is caused by the read, **BIT_BUS_ERROR** is returned to indicate an exception occurred. If a device is not enabled and enable/disable methods are defined, then the device is enabled, written to, and then disabled. The value read from the location is put into the *val* field of the *ADDR_INFO* structure passed to the method.

Here's a synopsis of the **bitIn()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitIn(ADDR_INFO *reg)
```

where *reg* is the pointer to the location's *ADDR_INFO* structure.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

bitOut()

This method writes to the location described by the *ADDR_INFO* structure passed to the method. If an exception is caused by the write, **BIT_BUS_ERROR** is returned to indicate an exception occurred. If a device is not enabled and enable/disable methods are defined, then the device is enabled, written to, and then disabled. The value actually written to the location is the value in the *val* field of the *ADDR_INFO* structure passed to the method.

Here's a synopsis of the **bitOut()** method:

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitOut(ADDR_INFO *reg)
```

where *reg* is the pointer to the location's *ADDR_INFO* structure.

Upon successful completion, this method returns **BIT_NO_FAULT_DETECTED**, otherwise it may return any of the

return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

Interrupt Utility Methods

MBIT provides methods to lock/unlock interrupts, and to connect methods to exceptions/interrupts for use during diagnostics. The header file, **utilities/bitCacheUtils.h**, provides the prototypes for these methods

Method	Description
bitIntLock()	Has the capability to lock all interrupts.
bitIntUnlock()	Has the capability to unlock all interrupts.
bitForceIntUnlock()	Has the capability to unlock all interrupts.
bitIntConnect()	Connects an MBIT interrupt handler to the MBIT interrupt table.
isBitIntEnabled()	Checks if an interrupt is enabled on a specified interrupt level.
bitIntVectorSet()	Sets a vector entry in the MBIT interrupt table.
bitIntEnable()	Enables the interrupt level.
bitIntDisable()	Disables the interrupt level.

bitIntLock()

This method increments an interrupts-locked reference count and if interrupts are not locked, it locks all interrupts.

Here's a synopsis of the **bitIntLock()** method:

```
<utilities/bitCacheUtils.h>
void bitIntLock(void)
```

There are no return values for this method.

bitIntUnlock()

This method decrements the interrupts-locked reference count incremented by **bitIntLock()** and if the reference count is **0**, it unlocks all interrupts.

Here's a synopsis of the **bitIntUnlock()** method:

```
<utilities/bitCacheUtils.h>
void bitIntUnlock(void)
```

There are no return values for this method.

bitForceIntUnlock()

This method sets the interrupts-locked reference count to **0** and unlocks all the interrupts locked by **bitIntLock()**.

Here's a synopsis of the **bitForceIntUnlock()** method:

```
<utilities/bitCacheUtils.h>
void bitForceIntUnlock(void)
```

There are no return values for this method.

bitIntConnect()

This method connects an MBIT interrupt handler to the MBIT interrupt table. In software, there may be up to 256 interrupts connected, however, hardware may limit the actual number available. This method only connects one handler to any interrupt vector at any one time. All interrupt handlers not connected, that use this method, remain as they were installed by the operating system (chained handlers remain chained). To disconnect the handler from the vector, use **bitIntVectorSet()** with a **NULL** entry.

Here's a synopsis of the **bitIntConnect()** method:

```
<utilities/bitCacheUtils.h>
STATUS bitIntConnect(VOIDFUNCPTR *vector,
                             VOIDFUNCPTR routine,
                             int param)
```

where *vector* is the interrupt vector to connect, *routine* is the routine to connect to the specified vector, and *param* is the parameter provided to the specified routine (when an interrupt occurs).

Upon successful completion, this method returns **OK** if no fault is detected, otherwise it may return any of the return values listed in Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults*.

isBitIntEnabled()

This method checks if an interrupt is enabled on a specified interrupt level.

Here's a synopsis of the **isBitIntEnabled()** method:

```
<utilities/bitCacheUtils.h>
STATUS isBitIntEnabled(int level)
```

where *level* is the interrupt level to be tested.

Upon successful completion, this method returns **TRUE** if enabled, **FALSE** if not enabled, and **–1** if the interrupt level could not be resolved. Refer to Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitIntVectorSet()

This method sets a vector entry in the MBIT interrupt table.

Here's a synopsis of the **bitIntVectorSet()** method:

```
<utilities/bitCacheUtils.h>
STATUS bitIntVectorSet(VOIDFUNCPTR *vector, INT32 *entry)
```

where *vector* is the interrupt vector to connect and *entry* is the method to connect to the specified vector.

There are no return values for this method.

bitIntEnable()

This method enables the interrupt level.

Here's a synopsis of the **bitIntEnable()** method:

```
<utilities/bitCacheUtils.h>
INT32 bitIntEnable(INT32 level)
```

where *level* is the interrupt level to enable.

Upon successful completion, this method returns **OK** if no fault is detected or **–1** if the interrupt level could not be resolved. Refer to Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

bitIntDisable()

This method disables the interrupt level.

Here's a synopsis of the **bitIntDisable()** method:

```
<utilities/bitCacheUtils.h>
INT32 bitIntDisable(INT32 level)
```

where *level* is the interrupt level to be disabled.

Upon successful completion, this method returns **OK** if no fault is detected or **–1** if the interrupt level could not be resolved. Refer to Appendix C, *Utility Methods' Reference Pages* and Chapter 5, *MBIT Faults* for more information.

Time Utility Methods

MBIT provides two time-related utility methods. The header file, **utilities/bitCacheUtils.h**, provides the prototypes for these methods

Method	Description
bitUsDelay()	Delays *micro* number of microseconds.
bitMsDelay()	Delays for *milli* number of milliseconds.

bitUsDelay()

This method delays *micro* number of microseconds. Note that the resolution may be more than a microsecond, so this call gives the smallest timeout possible in those cases.

Here's a synopsis of the **bitUsDelay()** method:

```
<utilities/bitCacheUtils.h>
void bitUsDelay(UINT32 micro)
```

where *micro* is the number of microseconds to delay.

There are no return values for this method.

bitMsDelay()

This method delays for *milli* number of milliseconds.

Here's a synopsis of the **bitMsDelay()** method:

```
<utilities/bitCacheUtils.h>
void bitMsDelay(UINT32 milli)
```

where *milli* is the number of milliseconds to delay.

There are no return values for this method.

MBIT Faults

5

The following tables list the faults that are either built into the MBIT API or pre-defined by the MVME51*xx* diagnostics.

Built-In MBIT Faults

The built-in faults may be used during any stage from initialization to termination of the MBIT API. H=hardware and S=software.

Table 5-1. Built-In MBIT Faults

Type	MBIT Fault String	Description
H	BIT_NO_FAULT_DETECTED	No fault detected, successful.
S	BIT_INIT_NOT_PERFORMED	MBIT initialization was already performed.
S	BIT_INIT_ALLOCATION_ERROR	Required resources for initialization are unavailable.
S	BIT_INIT_ALREADY_PERFORMED	Attempted to perform initialization twice.
S	BIT_DEV_INSTALL_NOT_DEFINED	Device installation is not configured.
S	BIT_TST_INSTALL_NOT_DEFINED	Test installation is not configured.
S	BIT_DEVICE_NOT_PRESENT	Device is not present.
S	BIT_DEVICE_NOT_SUPPORTED	Device is not supported.
S	BIT_SUBTEST_NOT_SUPPORTED	Selected subtest is not supported on this device.
S	BIT_INVALID_TEST_PARAM	Invalid test parameter was supplied.
S	BIT_TEST_NOT_EXECUTED	No results available for this test.
S	BIT_TEST_ABORTED	The test was aborted by the operator.

Table 5-1. Built-In MBIT Faults (continued)

Type	MBIT Fault String	Description
S	BIT_OPERATION_IN_PROGRESS	A previous operation (test or abort) is in progress.
S	BIT_INVALID_MSG_ERROR	An invalid message was received.
S	BIT_MESSAGE_QUEUE_ERROR	Unable to queue or de-queue a message.
S	BIT_RESOURCE_MGMT_FAULT	A memory management error occurred.
S	BIT_INSTALL_TEST_FAILED	Test installation failed.
S	BIT_INSTALL_DEV_FAILED	Device installation failed.
S	BIT_SYS_RESTORE_FAILED	Unable to restore pre-test system state.
S	BIT_TEST_SEQUENCE_ERROR	Tests in list completed out of order.
S	BIT_DRIVER_SEQUENCE_ERROR	Driver methods invoked out of order.
S	BIT_TESTING_TERMINATED	Testing terminated.
S	BIT_INVALID_DEVICE_DESC	Device descriptor has invalid field (configuration error).
H	BIT_DEVICE_ENABLE_FAULT	Failed to enable a disabled device.
H	BIT_DEVICE_DISABLE_FAULT	Failed to disable an enabled device.
H	BIT_BUS_ERROR	Device did not respond to transfer.
S	BIT_FUNCTION_NOT_SUPPORTED	Function call is not supported.
H	BIT_INTERRUPT_FAULT	Initialization of interrupt failed.
H	BIT_TEST_TIMED_OUT	The test timed out prior to completion.
H	BIT_DATA_MISCOMPARE	Data miscompare on write and read sequence.
S	BIT_INVALID_SUBTEST_ID	The subtest ID is invalid.
S	BIT_INVALID_DEVICE_ID	The device ID is invalid.
S	BIT_INVALID_FAULT_ID	The fault ID is invalid.
S	BIT_DUPLICATE_ASSOCIATION	The association already exists.
S	BIT_INVALID_LIST_CONTENT	Data in the list is invalid.

Table 5-1. Built-In MBIT Faults (continued)

Type	MBIT Fault String	Description
S	BIT_ADD_IDENT_ERROR	An error occurred while adding identifier.
S	BIT_DUPLICATE_IDENT	The identifier has already been added to the list.
S	BIT_DEVICE_NOT_ASSOCIATED	The device is not associated with any tests.
S	BIT_CACHE_ROUTINE_NOT_ SUPPORTED	Cache routine is not defined or supported.
S	BIT_PROCESSOR_NOT_SUPPORTED	Processor type is unknown and not supported.
H	BIT_DATA_CACHE_NOT_ENABLED	Data cache must be enabled before calling a function.
H	BIT_CACHE_NOT_ENABLED	Cache must be enabled before calling a cache function.
H	BIT_CACHE_LOCK_NOT_SUPPORTED	Cache locking is not supported.
H	BIT_SPD_CHECKSUM_FAULT	SPD checksum fault.
H	BIT_SPD_SROM_FAULT	SPD access fault.
H	BIT_VPD_SROM_FAULT	VPD error accessing the VPD SROM.
H	BIT_VPD_CONTAINS_NO_CRC	VPD contains no valid CRC packet.
H	BIT_VPD_HEADER_INVALID	VPD header is invalid.
H	BIT_VPD_DATA_INVALID	VPD data is invalid.

Pre-Defined MBIT Faults

The pre-defined faults may only be used once initialization of MBIT has successfully completed and prior to termination. H=hardware and S=software.

Table 5-2. Pre-Defined MBIT Faults

Type	MBIT Fault String	Description
H	BIT_ECC_DETECT_ERROR	Memory error correction failed.
H	BIT_BATTERY_LOW_POWER	NVRAM battery power is low. Replace battery.
H	BIT_CPU_ALU_FAULT	CPU arithmetic logic unit fault.
H	BIT_CPU_SCIU_FAULT	CPU single-cycle integer unit fault.
H	BIT_CPU_MCIU_FAULT	CPU multi-cycle integer unit fault.
H	BIT_CPU_FPU_FAULT	CPU floating point unit fault.
H	BIT_CPU_LSU_FAULT	CPU load/store unit fault.
H	BIT_CPU_BIU_FAULT	CPU bus interface unit fault.
H	BIT_CPU_BPU_FAULT	CPU branch processing unit fault.
H	BIT_CPU_MMU_FAULT	CPU memory management unit fault.
H	BIT_CPU_VPU_FAULT	CPU vector processing unit fault.
H	BIT_VISIBILITY_FAULT	Device failed visibility test.
H	BIT_NO_VISIBILITY_LOCATION	No location on device designated for visibility testing.
H	BIT_DEVICE_ENABLE_FAULT	Failed to enable a disabled device.
H	BIT_DEVICE_DISABLE_FAULT	Failed to disable an enabled device.
H	BIT_VPD_CRC_FAULT	VPD CRC did not equal the calculated CRC.
H	BIT_RTC_CLOCK_FAULT	RTC clock read fault.
H	BIT_RTC_CLOCK_SET_FAULT	RTC clock set fault.
H	BIT_RTC_CLOCK_ACCURACY_FAULT	RTC clock accuracy fault.

Table 5-2. Pre-Defined MBIT Faults (continued)

Type	MBIT Fault String	Description
H	BIT_RTC_ALARM_FAULT	RTC alarm timer fault.
H	BIT_RTC_ALARM_ACCURACY_FAULT	RTC alarm time accuracy fault.
H	BIT_RTC_WATCHDOG_FAULT	RTC watchdog timer fault.
H	BIT_RTC_WATCHDOG_EARLY_FAULT	RTC watchdog timer early fault.
H	BIT_SERIAL_REGISTER_FAULT	Serial register test fault.
H	BIT_SERIAL_RECV_FAULT	Serial receiver fault (parity, frame, overrun).
H	BIT_PARALLEL_REGISTER_FAULT	Parallel register test fault.
H	BIT_PARALLEL_FIFO_FAULT	Parallel FIFO test fault.
H	BIT_MPIC_INTERRUPT_FAULT	MPIC interrupt controller fault.
H	BIT_ISA_INTERRUPT_FAULT	ISA interrupt controller fault.
H	BIT_MPIC_INTERRUPT_MARGINAL	MPIC interrupt controller marginal.
H	BIT_ISA_INTERRUPT_MARGINAL	ISA interrupt controller marginal.
H	BIT_SCSI_SELECTION_FAULT	SCSI target selection failed.
H	BIT_SCSI_FIFO_CLEAR_FAULT	SCSI and/or DMA FIFOs failed to clear.
H	BIT_SCSI_ARBITRATE_FAULT	SCSI target arbitration fault.
H	BIT_SCSI_TIMER_FAULT	SCSI bus timer(s) indication of timeout failed.
H	BIT_SCSI_BMOVE_TIMEOUT	SCSI block move timed out.
H	BIT_SCSI_PARITY_FAULT	SCSI parity error interrupt.
H	BIT_SCSI_NO_PARITY_FAULT	SCSI parity error interrupt failed to assert.
H	BIT_SCSI_ILLEGAL_FAULT	SCSI illegal instruction interrupt.
H	BIT_SCSI_NO_ILLEGAL_FAULT	SCSI illegal instruction interrupt failed to assert.
H	BIT_PCI_BUS_FAULT	PCI bus fault indicated.
H	BIT_NO_PCI_BUS_FAULT	PCI bus fault not indicated.

Table 5-2. Pre-Defined MBIT Faults (continued)

Type	MBIT Fault String	Description
H	BIT_SCSI_INTERRUPT_TIMEOUT	SCSI timeout interrupt received.
H	BIT_SCSI_INTERRUPT_FAULT	SCSI interrupt failed to assert.
H	BIT_SCSI_INIT_ERROR	SCSI bus not free for initialization.
H	BIT_SCSI_CONTROL_LINE_FAULT	SCSI bus control lines contain bad data.
H	BIT_SCSI_DATA_LINE_FAULT	SCSI bus data lines contain bad data.
H	BIT_USER_DATA_SROM_FAULT	SROM user configuration data access error.
H	BIT_VME_REGISTER_IO_FAULT	VME bridge register write/read access fault.
H	BIT_VME_REGISTER_DATA_FAULT	VME bridge register write/read data miscompare.
H	BIT_VME_REGISTER_VALUE_FAULT	VME bridge register read value fault.
H	BIT_VME_RW_TARGET_IO_FAULT	VME write/read target access fault.
H	BIT_VME_RW_TARGET_DATA_FAULT	VME write/read target data miscompare.
H	BIT_VME_RW_TARGET_INT_FAULT	VME write/read target interrupt fault.
H	BIT_VME_NO_TARGET_RW_IO_FAULT	VME target write/read to non-existent address succeeded.
H	BIT_VME_LOCMON_FAULT	VME bridge location monitor (mailbox) interrupt fault.
H	BIT_VME_LOCMON_IO_FAULT	VME bridge location monitor (mailbox) access fault.
H	BIT_ETHERNET_SROM_ACCESS_ ERROR	An error occurred while accessing the Serial EEPROM (SROM) connected to the Ethernet device.
H	BIT_ETHERNET_SROM_CHECKSUM_ FAULT	The checksum of the Serial EEPROM (SROM) does not match the calculated value.

Table 5-2. Pre-Defined MBIT Faults (continued)

Type	MBIT Fault String	Description
H	BIT_ETHERNET_SROM_VERIFY_ FAULT	The contents of the Serial EEPROM (SROM) do not make sense or do not match the expected values.
H	BIT_ETHERNET_REGISTER_ ACCESS_FAULT	An error occurred while accessing the Ethernet device registers.
H	BIT_ETHERNET_LOOPBACK_DATA_ FAULT	The transmitted and received data sent in loopback do not match.
H	BIT_ETHERNET_DRIVER_FAULT	A driver fault occurred.
H	BIT_ETHERNET_TRANSMIT_ERROR	An error occurred during transmission.
H	BIT_ETHERNET_TRANSMIT_BLOCK	The transmission would block.
H	BIT_FLASH_STUCK_ERROR	Possible stuck bit was detected, or insufficient variability in Flash data.
H	BIT_FLASH_FLOAT_ERROR	Possible floating bit was detected.
H	BIT_THERMOMETER_REGISTER_ FAULT	Thermometer register fault.
H	BIT_THERMOMETER_ALARM_FAULT	Thermometer thermal alarm fault.
H	BIT_THERMOMETER_RANGE_FAULT	Thermometer thermal range fault.
H	BIT_THERMOMETER_I2C_RW_FAULT	Thermometer I^2C read/write fault.
H	BIT_THERMOMETER_ACTIVE_FAULT	Thermometer active alarm fault.
H	BIT_THERMOMETER_INACTIVE_ FAULT	Thermometer inactive alarm fault.

API Method's Reference Pages

A

This appendix provides detailed information about the MBIT API methods mentioned in Chapter 2, *Using MBIT*.

initBit()

Name

initBit()—initializes the MBIT software

Synopsis

```
#include <api/bit.Api.h>
BIT_FAULT initBit(BIT_FAULT (*pConfigRoutines[]()),
                                int numConfigRoutines)
```

Parameters

pConfigRoutines[]

is an array of function pointers to custom subtest and device configuration methods.

numConfigRoutines

is the number of custom configuration methods.

Description

This method performs MBIT system initialization and must be invoked prior to any other method in MBIT. This method creates the test list processing task and the subtest control task (refer to Chapter 1, *MBIT Overview*).

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_ALREADY_PERFORMED—MBIT initialization was already performed
BIT_INIT_ALLOCATION_ERROR—required resources for initialization are unavailable
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_OPERATION_IN_PROGRESS—MBIT test or abort in progress
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message

Refer to Chapter 5, *MBIT Faults* for more faults.

reinitBit()

Name

reinitBit()—clears the device fault database

Synopsis

```
#include <api/bitApi.h>
BIT_FAULT reinitBit(void)
```

Parameters

No input parameters are required by this method.

Description

This method places the MBIT API in an initial state after executing tests. As a result, the fault database clears and the Fail LED extinguishes.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_OPERATION_IN_PROGRESS—MBIT test or abort in progress
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message

Refer to Chapter 5, *MBIT Faults* for more faults.

isBitInitializationComplete()

Name

isBitInitializationComplete()—returns the MBIT initialization status

Synopsis

```
#include <api/bitApi.h>
BOOLEAN isBitInitializationComplete(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the MBIT initialization status.

Return Values

TRUE—MBIT initialization is complete
FALSE—MBIT initialization is *not* complete

Refer to Chapter 5, *MBIT Faults* for more faults.

executeBitTests()

Name

executeBitTests()—submits and executes a list of subtests

Synopsis

```
#include <api/bitApi.h>
BIT_FAULT executeBitTests(BIT_TEST_CONTROL listControl,
                          unsigned int testCount,
                          TEST_ENTRY testList[])
```

Parameters

listControl

specifies **HALT_ON_ERROR** or **RUN_TILL_COMPLETION** (see **config/bitCommonDefs.h**).

testCount

is the number of entries in the test list.

testList[]

is an array of tests to execute.

Description

Executing subtests begins with creating test lists and submitting them for execution by calling **executeBitTests()**. Follow this method up with **getBitResponse()**, which returns test results. **abortBitTests()** allows aborting test execution any time outside critical sections during tests. See the section on subtest attributes in the *Test Reference Guide* for a list of subtests with protected critical sections.

An MBIT application may occupy two threads of execution. All API methods, except **getBitResponse()**, must be called from the thread **initBit()** is called from. **getBitResponse()** may be called from another thread.

Note A user may submit a test list filled with default test entries for a given device by calling either **buildBitDefaultTestList()** or **buildBitDefaultTestEntry()**.

This method submits and executes MBIT tests. In creating a test list, the following applies:

- ❏ Any subtest can be included in a single test list and a test list may contain a single entry.
- ❏ The number of subtest entries in a test list is limited to the **getBitMaxTestListEntrie()s** method.
- ❏ Only one subtest is executed at any one time.

To execute a test list, specify the *listControl* parameter with one of the available values, **HALT_ON_ERROR** or **RUN_TILL_COMPLETION**. If **HALT_ON_ERROR** is specified, processing of the test list terminates with the detection of the first failed test. Otherwise, all tests in the test list execute without stopping for failed tests.

Value	Definition
HALT_ON_ERROR	Test list terminates with the detection of the first failed test.
RUN_TILL_COMPLETION	Test list executes without stopping for failed tests.

A count of the subtests contained in the test list must be supplied along with the address of an array of test entries. The *TEST_ENTRY* definition is contained in the configuration file, **config/bitCommonDefs.h**. Each test entry specifies a test to execute, the logical device to be tested, a test duration, an iteration count and a test control specifying one of the values, **HALT_ON_ERROR** or **RUN_TILL_COMPLETION**.

The test duration parameter determines the maximum allowable time in which a test must complete. If the test does not complete within the specified time, it aborts and a failure is declared. The test duration may be affected by optional test parameters.

The test iteration count determines the number of times the test executes in succession. The test duration parameter is automatically multiplied by the iteration count to determine total allowable execution time. Required driver installation, exception handler installation, and test initialization are performed only once; the test must provide its own re-initialization prior to returning status. Once the test executes the required number of times, the results return to test control and invoke the test de-installation method.

A subtest has two options for control when it runs. The subtest can use **RUN_TILL_COMPLETION** or **HALT_ON_ERROR** as the control. If the test uses **RUN_TILL_COMPLETION**, the test may run till the end even if a fault occurs. If **HALT_ON_ERROR** is the control parameter, then the subtest allows the execution to halt after the first fault occurs and returns to the API. For clarification, the following paragraph is an example of a test that uses both types of control as the control parameter.

Example: a subtest wants to test a list of five different bit patterns for reading and writing to a register. If the subtest uses **HALT_ON_ERROR** for its control parameter, the test returns when one of the patterns receives an error during reading or writing. The test does not continue to read or write the other patterns. If the subtest uses **RUN_TILL_COMPLETION** for its control parameter, the test runs all of the bit patterns even if an error occurs during the reading or writing to the register.

In addition to the parameters already described, many subtests allow additional control over the nature of the testing. For example, memory tests allow the user to specify the patterns to be used and the memory locations to be tested. To determine if a subtest allows additional parameters, reference the *Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide*, listed in Appendix F, *Related Documentation*, and the associated include file for the package that implements the test. To support such controls, a pointer to a user-supplied structure passes to the subtest upon invocation. The pointer is set to **NULL** if the test does not support subtest-specific controls.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_OPERATION_IN_PROGRESS—MBIT test or abort in progress
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_TST_INSTALL_NOT_DEFINED—test installation is not configured
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message

Refer to Chapter 5, *MBIT Faults* for more faults.

buildBitDefaultTestList()

Name

buildBitDefaultTestList()—fills in a list of default test entries

Synopsis

```
#include <config/testDefaults.h>
BIT_FAULT buildBitDefaultTestList(
                                BIT_LOGICAL_DEVICE deviceId,
                                unsigned int *numTests,
                                TEST_ENTRY testEntryList[])
```

Parameters

deviceId

is an ID specifying a unique device.

numTests

is the number of test entries being returned.

testEntryList[]

is the list of test entries being returned for a given device.

Description

This method fills in a test list with default test entries for each subtest associated with the given device. The list of test entries must be allocated before this method is called. The maximum number of test entries returned is no more than the value returned by the **getBitMaxTestListEntries()** method. Refer to the *Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide* for a list of subtests and the associated devices.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_INVALID_TEST_PARAM—invalid parameter was supplied

Refer to Chapter 5, *MBIT Faults* for more faults.

buildBitDefaultTestEntry()

Name

buildBitDefaultTestEntry()—fills in a single default test entry

Synopsis

```
#include <config/testDefaults.h>
BIT_FAULT buildBitDefaultTestEntry(BIT_SUBTEST subtestId,
                                   BIT_LOGICAL_DEVICE deviceId,
                                   TEST_ENTRY *testEntry)
```

Parameters

subtestId

is an ID specifying a unique subtest.

deviceId

is an ID specifying a unique device.

testEntry

is a test entry being returned for the associated subtest and device.

Description

This method fills in a single default test entry for the associated subtest and device. The test entry must be allocated before this method is called.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_INVALID_TEST_PARAM—invalid parameter was supplied

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitResponse()

Name

getBitResponse()—obtains a list of test results

Synopsis

```
#include <api/bitApi.h>
BIT_FAULT getBitResponse(TEST_RESULTS_ENTRY testResults[],
                         unsigned int *numberOfResults)
```

Parameters

testResults[]

is a user allocated array for the test results.

numberOfResults

receives the number of entries in the test results list. If the number of results exceeds the value returned by **getBitMaxTestListEntries()**, an error is returned.

The number of *testResults* entries allocated must be greater than or equal to the number of test entries submitted with **executeBitTests()**. The number of *testResults* returned will be less than or equal to the number of test entries submitted with **executeBitTests()**.

Description

This method obtains the results of a test list submitted by invoking **executeBitTests()**. This method blocks until the executing test list completes, times out, or aborts. If this method is called when no tests are executing, it blocks the viewing of test results until a call is made to **executeBitTests()** and all tests in the list complete execution. **getBitResponse()** returns a single test results list and removes it from the response queue.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message
BIT_INVALID_MSG_ERROR—invalid message was received

Refer to Chapter 5, *MBIT Faults* for more faults.

getNumBitResponses()

Name

getNumBitResponses()—provides the number of MBIT test results lists

Synopsis

```
#include <api/bitApi.h>
BIT_FAULT getNumBitResponses(int *msgCount)
```

Parameters

msgCount

will contain the number of test results lists in the MBIT response queue.

Description

This method provides the number of MBIT test results lists in the MBIT response queue.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message
BIT_RESOURCE_MGMT_FAULT—memory management error occurred

Refer to Chapter 5, *MBIT Faults* for more faults.

abortBitTests()

Name

abortBitTests()—aborts any subtest in progress

Synopsis

```
#include <api/bitApi.h>
BIT_FAULT abortBitTests(void)
```

Parameters

No input parameters are required by this method.

Description

This method terminates current test list processing and aborts any subtest in progress. This method has no effect if a subtest is not executing or has already completed. Tests with protected critical sections will not be aborted until the critical section is exited. Test results for those tests already complete are made available in response to the submitted test list. For each successfully submitted test list, a single test results list is placed in the response queue.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitDeviceFault()

Name

getBitDeviceFault()—obtains fault information for a specific logical device

Synopsis

```
#include <config/bitCommonDefs.h>
BIT_FAULT getBitDeviceFault(BIT_LOGICAL_DEVICE device,
                            BIT_FAULT *deviceFault)
```

Parameters

device

is the logical device for which fault data is being requested.

deviceFault

is the fault code.

Description

This method obtains the fault data for the device specified by the input device value. The first fault detected for the specified logical device returns to the caller.

Invoking **reinitBit()** clears the collected fault data and places MBIT in an initial state. This action also extinguishes the Fail LED when it illuminates from the detection of a fault by a previous test.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_RESOURCE_MGMT_FAULT—memory management error occurred
BIT_INVALID_TEST_PARAM—invalid parameter was supplied

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitSubtestDesc()

Name

getBitSubtestDesc()—obtains a string describing a subtest

Synopsis

```
#include <config/bitCommonDefs.h>
const char* getBitSubtestDesc(BIT_SUBTEST subtestId)
```

Parameters

subtestId

is an ID specifying a unique subtest.

Description

This method returns a string describing the subtest. The string descriptions returned should not be modified or free'd.

Return Values

Upon successful completion, **getBitSubtestDesc()** returns a string containing the subtest description. If it fails, it returns an empty string.

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitDeviceDesc()

Name

getBitDeviceDesc()—obtains a string describing a logical device

Synopsis

```
#include <config/bitCommonDefs.h>
const char* getBitDeviceDesc(BIT_LOGICAL_DEVICE deviceId)
```

Parameters

deviceId

is an ID specifying a unique device.

Description

This method returns a string describing the logical device.

Return Values

Upon successful completion, **getBitDeviceDesc()** returns a string containing the device description. If it fails, it returns an empty string.

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitFaultDesc()

Name

getBitFaultDesc()—obtains a string describing a subtest fault

Synopsis

```
#include <config/bitCommonDefs.h>
const char* getBitFaultDesc(BIT_FAULT faultId)
```

Parameters

faultId

is an ID specifying a unique fault.

Description

This method returns a string describing the subtest fault.

Return Values

Upon successful completion, **getBitFaultDesc()** returns a string containing the fault description. If it fails, it returns "No description supplied for fault."

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitSubtestIdByName()

Name

getBitSubtestIdByName()—returns the subtest string identifier ID

Synopsis

```
#include <config/bitCommonDefs.h>
BIT_SUBTEST getBitSubtestIdByName(const char* const subtest)
```

Parameters

subtest

is a string identifier specifying the subtest.

Description

This method returns the ID representing the subtest string identifier.

Return Values

BIT_SUBTEST—an ID representing the subtest identifier or
–1—the subtest is not found

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitDeviceIdByName()

Name

getBitDeviceIdByName()—returns the device string identifier ID

Synopsis

```
#include <config/bitCommonDefs.h>
BIT_LOGICAL_DEVICE getBitDeviceIdByName(
                              const char* const device)
```

Parameters

device

is a string identifier specifying the device.

Description

This method returns the ID representing the device string identifier.

Return Values

BIT_LOGICAL_DEVICE—an ID representing the device identifier
–1—the device is not found

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitFaultIdByName()

Name

getBitFaultIdByName()—returns the device string identifier ID

Synopsis

```
#include <config/bitCommonDefs.h>
BIT_FAULT getBitFaultIdByName(const char* const fault)
```

Parameters

fault

is a string identifier specifying the fault.

Description

This method returns the ID representing the fault string identifier.

Return Values

BIT_FAULT—an ID representing the fault identifier
–1—the fault is not found

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitNumberOfSubtests()

Name

getBitNumberOfSubtests()—returns the number of subtests

Synopsis

```
#include <config/bitCommonDefs.h>
int getBitNumberOfSubtests(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the number of subtests.

Return Values

Upon successful completion, **getBitNumberOfSubtests()** returns the number of MBIT configured subtests.

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitNumberOfDevices()

Name

getBitNumberOfDevices()—returns the number of devices

Synopsis

```
#include <config/bitCommonDefs.h>
int getBitNumberOfDevices(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the number of devices.

Return Values

Upon successful completion, **getBitNumberOfDevices()** returns the number of MBIT configured devices.

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitNumberOfFaults()

Name

getBitNumberOfFaults()—returns the number of faults

Synopsis

```
#include <config/bitCommonDefs.h>
int getBitNumberOfFaults(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the number of faults.

Return Values

Upon successful completion, **getBitNumberOfFaults()** returns the number of MBIT configured faults.

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitMaxTestListEntries()

Name

getBitMaxTestListEntries()—returns the maximum number of test list entries

Synopsis

```
#include <config/bitCommonDefs.h>
int getBitMaxTestListEntries(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the maximum number of test list entries.

Return Values

Upon successful completion, **getBitMaxTestListEntries()** returns the maximum number of MBIT configured test list entries.

Refer to Chapter 5, *MBIT Faults* for more faults.

terminateBit()

Name

terminateBit()—terminates the MBIT software

Synopsis

```
#include <api/bitApi.h>
BIT_FAULT terminateBit(void)
```

Parameters

No input parameters are required by this method.

Description

This method performs an orderly termination of the MBIT software, including releasing allocated resources and the termination of all spawned child tasks.

After invoking **terminateBit()**, you may call **initBit()**. There are no restrictions placed on the number of times you can call **initBit()** and **terminateBit()**, as long as each call to **initBit()** is followed by a call to **terminateBit()** prior to the next invocation of **initBit()**.

Note: Calling **initBit()** and **terminateBit()** an excessive number of times may cause memory fragmentation.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_NOT_PERFORMED—MBIT initialization was already performed
BIT_MESSAGE_QUEUE_ERROR—could not queue MBIT message
BIT_RESOURCE_MGMT_FAULT—memory management error occurred

Refer to Chapter 5, *MBIT Faults* for more faults.

Integrating Custom Diagnostics' Reference Pages

B

This appendix provides detailed information about the methods mentioned in Chapter 3, *Integrating Custom Diagnostics*. These methods are categorized into the following four sections.

Diagnostic Integration Methods

MBIT provides the following methods for integrating diagnostics:

B

addBitSubtestIdent()

Name

addBitSubtestIdent()—adds a subtest entry

Synopsis

```
#include <config/bitTestUtils.h>
BIT_FAULT addBitSubtestIdent(const char *subtest,
                             const char *description,
                             BIT_SUBTEST *id)
```

Parameters

subtest

is the unique subtest name.

description

is the description of the *subtest* name.

id

is the unique ID being returned that represents the *subtest*.

Description

This method adds a subtest entry with the provided identifier and description.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_ALREADY_PERFORMED—MBIT initialization was already performed
BIT_INIT_ALLOCATION_ERROR—required resources for initialization are unavailable
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_DUPLICATE_IDENT—name of the identifier already exists
BIT_RESOURCE_MGMT_FAULT— memory management error occurred
BIT_INVALID_LIST_CONTENT—content of list is invalid

Refer to Chapter 5, *MBIT Faults* for more faults.

B

addBitDeviceIdent()

Name

addBitDeviceIdent()—adds a device entry

Synopsis

```
#include <config/bitTestUtils.h>
BIT_FAULT addBitDeviceIdent(const char *device,
                            const char *description,
                            BIT_LOGICAL_DEVICE *id)
```

Parameters

device

is the unique device name.

description

is the description of the *device* name.

id

is the unique ID being returned that represents the *device*.

Description

This method adds a device entry with the provided identifier and description.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_ALREADY_PERFORMED—MBIT initialization was already performed
BIT_INIT_ALLOCATION_ERROR—required resources for initialization are unavailable
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_DUPLICATE_IDENT—name of the identifier already exists
BIT_RESOURCE_MGMT_FAULT—memory management error occurred
BIT_INVALID_LIST_CONTENT—content of list is invalid

Refer to Chapter 5, *MBIT Faults* for more faults.

B

addBitFaultIdent()

Name

addBitFaultIdent()—adds a fault entry

Synopsis

```
#include <config/bitTestUtils.h>
BIT_FAULT addBitFaultIdent(const char *fault,
                           const char *description,
                           BIT_FAULT_TYPE type,
                           BIT_FAULT *id)
```

Parameters

fault

is the unique fault name.

description

is the description of the *fault*.

type

is the type of *fault* (for example, hardware, software; see **config/bitCommonDefs.h**).

id

is the unique ID being returned that represents the *fault*.

Description

This method adds a fault entry with the provided identifier and description.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_ALREADY_PERFORMED—MBIT initialization was already performed
BIT_INIT_ALLOCATION_ERROR—required resources for initialization are unavailable
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_DUPLICATE_IDENT—name of the identifier already exists
BIT_RESOURCE_MGMT_FAULT—memory management error occurred
BIT_INVALID_LIST_CONTENT—content of list is invalid

Refer to Chapter 5, *MBIT Faults* for more faults.

B

createBitTestAssociations()

Name

createBitTestAssociations()—creates an association between devices, subtests, and a driver

Synopsis

```
#include <config/bitTestUtils.h>
BIT_FAULT createBitTestAssociations(BIT_SUBTEST subtestId[],
                                    int numSubtestIds,
                                    BIT_LOGICAL_DEVICE deviceId[],
                                    int numDeviceIds,
                                    DRV_DESC *pDriveDesc)
```

Parameters

subtestId[]

is an array of subtest IDs.

numSubtestIds

is the number of subtest IDs.

deviceId[]

is an array of device IDs.

numDeviceIds

is the number of device IDs.

pDriveDesc

is the pointer to the driver being associated with the subtests and devices.

Description

This method creates an association between subtests, devices, and a driver. If a driver is not available for the associated subtest and device, the *pDriveDesc* parameter should be **NULL**.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_ALREADY_PERFORMED—MBIT initialization was already performed
BIT_INIT_ALLOCATION_ERROR—required resources for initialization are unavailable
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_INVALID_SUBTEST_ID—subtest ID is invalid
BIT_INVALID_DEVICE_ID—device ID is invalid
BIT_DUPLICATE_ASSOCIATION—association has already been created
BIT_RESOURCE_MGMT_FAULT—memory management error occurred

Refer to Chapter 5, *MBIT Faults* for more faults.

B

installBitDriver()

Name

installBitDriver()—installs the driver entry points

Synopsis

```
#include <api/bitGenericDriver.h>
DRV_DESC* installBitDriver(
     BIT_FAULT (*drvInstall) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvDeinstall) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvOpen) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvClose) (DEV_DESC *devDescPtr),
     BIT_FAULT (*drvRead) (DEV_DESC *devDescPtr,
                           unsigned int bufferSize,
                           char *buffer,
                           unsigned int *bytesRead),
     BIT_FAULT (*drvWrite) (DEV_DESC *devDescPtr,
                           unsigned int bufferSize,
                           char *buffer,
                           unsigned int *bytesWritten),
     BIT_FAULT (*drvIoctl) (DEV_DESC *devDescPtr
                           int function,
                           int argument))
```

Parameters

drvInstall

is the driver install entry point. Also refer to *devXXXInstall()* on page B-29.

drvDeinstall

is the driver deinstall entry point. Also refer to *devXXXDeinstall()* on page B-31.

drvOpen

is the driver open entry point. Also refer to *devXXXOpen()* on page B-33.

drvClose

is the driver close entry point. Also refer to *devXXXClose()* on page B-34.

drvRead

is the driver read entry point. Also refer to *devXXXRead()* on page B-35.

drvWrite

is the driver write entry point. Also refer to *devXXXWrite()* on page B-37.

drvIoctl

is the driver ioctl entry point. Also refer to *devXXXIoctl()* on page B-39.

Description

This method installs the driver entry points. When you add a driver, you must implement all of the driver entry points.

Return Values

DRV_DESC—pointer to the driver descriptor
NULL—an error occurred

Refer to Chapter 5, *MBIT Faults* for more faults.

B

installBitSubtestEntries()

Name

installBitSubtestEntries()—installs the required subtest entry points and sets the default parameters

Synopsis

```
#include <config/bitTestUtils.h>
BIT_FAULT installBitSubtestEntries(
     BIT_SUBTEST subtest,
     BIT_FAULT (*installTest) (BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device,
                               void *testParamPtr),
     BIT_FAULT (*deinstallTest) (BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device,
                               void *testParamPtr),
     BIT_FAULT (*runTest) (BIT_SUBTEST subtest,
                               BIT_LOGICAL_DEVICE device,
                               void *testParamPtr),
     int interations,
     int duration
     BIT_TEST_CONTROL control,
     BIT_FAULT (*freeParamPtr) (BIT_SUBTEST subtest,
                               void *testParamPtr),
     BIT_FAULT (*initParamPtr) (BIT_SUBTEST subtest,
                               void *testParamPtr),
     int paramSize)
```

Parameters

subtest

is the subtest ID the methods and default parameters are associated with.

installTest

is the subtest installation method.

deinstallTest

is the subtest de-installation method.

runTest

is the actual test method.

iterations

is the default number of times to run the test.

duration

is the default maximum number of milliseconds the test is allowed to run.

control

is the default test control to halt on the first error detected or to run until test completion, if possible.

freeParamPtr

is the pointer to the free parameter method. This method is responsible for memory de-allocation of the parameter structure.

initParamPtr

is the pointer to the method that initializes the default parameter structure. This method is responsible for any memory allocation and initialization of the default parameter structure.

paramSize

is the size of the parameter structure used by the subtest.

Description

This method installs the required subtest entry points and sets the default parameters, which are used by **buildBitDefaultTestList** and **buildBitDefaultTestEntry**.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_INIT_ALREADY_PERFORMED—MBIT initialization was already performed
BIT_INIT_ALLOCATION_ERROR—required resources for initialization are unavailable
BIT_INVALID_TEST_PARAM—invalid parameter was supplied
BIT_RESOURCE_MGMT_FAULT—memory management error occurred

Refer to Chapter 5, *MBIT Faults* for more faults.

getBitNumberOfAssociations()

Name

getBitNumberOfAssociations()—obtains the number of associations

Synopsis

```
#include <config/bitTestUtils.h>
int getBitNumberOfAssociations(void)
```

Parameters

None

Description

This method returns the number of associations.

Return Values

Upon successful completion, **getBitNumberOfAssociations()** returns the number of associations. If MBIT has not been initialized, it returns **0**.

Refer to Chapter 5, *MBIT Faults* for more faults.

Generic Device Driver Methods

MBIT provides the following generic device driver methods:

drvInstall() on page B-17

drvDeinstall() on page B-19

drvOpen() on page B-21

drvClose() on page B-22

drvRead() on page B-23

drvInstall()

Name

drvInstall()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvInstall(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device);
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

It is suggested that this method be invoked by the install test method for the specified subtest.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DRIVER_SEQUENCE_ERROR—driver methods invoked out of order
BIT_DEV_INSTALL_NOT_DEFINED—device installation is not configured

Refer to Chapter 5, *MBIT Faults* for more faults.

drvDeinstall()

Name

drvDeinstall()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvDeinstall(BIT_SUBTEST subtest,
                       BIT_LOGICAL_DEVICE device)
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

This method must be invoked by the de-install test method for the specified subtest.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DEV_INSTALL_NOT_DEFINED—device installation is not configured

Refer to Chapter 5, *MBIT Faults* for more faults.

drvOpen()

Name

drvOpen()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvOpen(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device)
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DRIVER_SEQUENCE_ERROR—driver methods invoked out of order
BIT_DEV_INSTALL_NOT_DEFINED—device installation is not configured

Refer to Chapter 5, *MBIT Faults* for more faults.

B

drvClose()

Name

drvClose()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvClose(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device)
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

This method must be called before the **drvDeinstall()** is called.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DEV_INSTALL_NOT_DEFINED—device installation is not configured

Refer to Chapter 5, *MBIT Faults* for more faults.

drvRead()

Name

drvRead()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvRead(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device,
                                unsigned int bufferSize,
                                char *buffer,
                                unsigned int *bytesRead)
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

bufferSize

is the size of *buffer* in bytes.

buffer

is the buffer to place data in.

bytesRead

is a pointer to the number of bytes read.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

This method must be called after the **drvOpen()** has been called.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DRIVER_SEQUENCE_ERROR—driver methods invoked out of order
BIT_INVALID_DEVICE_DESC—device descriptor has invalid field (configuration error)

Refer to Chapter 5, *MBIT Faults* for more faults.

drvWrite()

Name

drvWrite()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvWrite(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device,
                                unsigned int bufferSize,
                                char *buffer,
                                unsigned int *bytesWritten)
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

bufferSize

is the number of bytes from *buffer* to write.

buffer

is the buffer to write data from.

bytesWritten

is the pointer to the number of bytes written.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

This method must be called after **drvOpen()** has been called.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DRIVER_SEQUENCE_ERROR—driver methods invoked out of order
BIT_INVALID_DEVICE_DESC—device descriptor has invalid field (configuration error)

Refer to Chapter 5, *MBIT Faults* for more faults.

drvIoctl()

Name

drvIoctl()—finds the associated driver routine based on the subtest and logical device passed in

Synopsis

```
#include<api/bitGenericDriver.h>
BIT_FAULT drvIoctl(BIT_SUBTEST subtest,
                                BIT_LOGICAL_DEVICE device,
                                int function,
                                int argument)
```

Parameters

subtest

is the current subtest.

device

is the device to operate on.

function

is the driver-specific operation to perform on the device or driver.

argument

is a driver-specific argument for the *function*.

Description

This method finds the associated driver routine based on the subtest and logical device passed in. This method looks up the device descriptor based on the logical device passed in. The associated device driver routine is then called.

This routine must be called after the **drvOpen()** has been called.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_SUBTEST_NOT_SUPPORTED—selected subtest is not supported on this device
BIT_DEVICE_NOT_SUPPORTED—device is not supported
BIT_DRIVER_SEQUENCE_ERROR—driver methods invoked out of order
BIT_INVALID_DEVICE_DESC—device descriptor has invalid field (configuration error)

Refer to Chapter 5, *MBIT Faults* for more faults.

Device Driver Methods

MBIT provides the following device driver methods:

dev*XXX*Install()

Name

dev*XXX*Install()—allows MBIT to prepare the driver for subsequent access of the device

Synopsis

```
devXXXInstall(DEV_DESC *pDevDesc)
```

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform an install operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

Description

This method allows MBIT to prepare the driver for subsequent access of the device. The driver should save all of the necessary device registers for restoration when the test is complete. This method receives a pointer to the device descriptor of the device being tested. All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

This method is responsible for allocating required resources (that is, buffers, semaphores, etc.) and saving the state of the device. It is also responsible for installing any required interrupt service methods. This method may also disable the device driver supplied by the underlying OS if such a capability is supported.

The **dev*XXX*Install()** method receives a pointer to the device descriptor of the device being tested. All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

In addition to the register information, the structure should also identify the interrupt level and vector that the device uses. The driver should use these definitions when using interrupts. This ensures that the driver's interrupt processing is connected to the right device, assuming the device descriptor is built correctly.

Return Values

The return values for this method are determined by the developer.

dev*XXX*Deinstall()

Name

dev*XXX*Deinstall()—allows the user to terminate the use of the driver

Synopsis

```
devXXXDeinstall(DEV_DESC *pDevDesc)
```

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform a de-install operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

Description

This method allows the user to terminate the use of the driver. The driver de-installation method is responsible for resource reclamation, restoring the device state, and de-installing the interrupt service methods. The driver should restore all of the device registers that were saved when the driver was installed. This method receives a pointer to a device descriptor of the device being tested. All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

The **dev*XXX*Deinstall()** method receives a pointer to a device descriptor of the device being tested. All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

In addition to the register information, the structure should also identify the interrupt level and vector that the device uses. The driver should use these definitions when using interrupts. This ensures that the driver's interrupt processing affects the correct device.

Return Values

The return values for this method are determined by the developer.

dev*XXX*Open()

Name

dev*XXX*Open()—allows the user to prepare the device for testing

Synopsis

dev*XXX*open(DEV_DESC *_pDevDesc_)

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform an open operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

Description

This method allows the user to prepare the device for testing. It receives a pointer to the device descriptor of the device being tested. Also, driver variables may be initialized in preparation for subsequent driver use.

All of the necessary device registers are contained in this structure. The driver should use these register definitions to interface with the correct device.

Return Values

The return values for this method are determined by the developer.

B

dev*XXX*Close()

Name

dev*XXX*Close()—allows the user to close the device in preparation for terminating use of the device

Synopsis

```
devXXXClose(DEV_DESC *pDevDesc)
```

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform a close operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

Description

This method allows the user to close the device in preparation for terminating use of the device. It receives a pointer to the device descriptor of the device being tested.

Typically, a device driver does not need to do processing to close the driver in the MBIT environment. The important device control and resource allocation release should be performed in the **devXXXDeinstall()** driver method.

Return Values

The return values for this method are determined by the developer.

dev*XXX*Read()

Name

dev*XXX*Read()—allows the user to read information from the device

Synopsis

```
devXXXRead(DEV_DESC *pDevDesc,
                              UINT32 bufferSize,
                              INT8 *bufferAddr,
                              UINT32 *bytesRead)
```

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform a read operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

bufferSize

contains the number of bytes that the user expects to read from the device. If the device supports transfers wider than a byte, the driver should adjust the count appropriately.

bufferAddr

points to the first element of the data buffer that the driver stores the data that is read from the device. The caller must provide a data buffer sufficiently large enough to accept the requested number of bytes defined in the buffer size parameter.

bytesRead

points to the variable that the driver returns as the number of bytes read. In the event of an error, the byte count should reflect the actual byte count of the received data.

Description

This method allows the user to read information from the device. It receives a pointer to the device descriptor of the device being tested, the requested buffer size, a pointer to the buffer address, and a pointer to a variable that holds the number of bytes read. The developer should also update the variable pointed to by the *bytesRead* parameter before returning to the caller.

Return Values

The return values for this method are determined by the developer.

dev*XXX*Write()

Name

dev*XXX*Write()—allow the driver to write information to the device

Synopsis

```
devXXXWrite(DEV_DESC *pDevDesc,
                              UINT32 bufferSize,
                              INT8 *bufferAddr,
                              UINT32 *bytesWritten)
```

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform a write operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

bufferSize

contains the number of bytes that the user expects to write to the device. If the device supports transfers wider than a byte, the driver should adjust the count appropriately.

bufferAddr

points to the first element of the data buffer that the driver reads data that is to be written to the device. The caller should provide a data buffer sufficiently large enough to reflect the requested number of bytes defined in the *bufferSize* parameter.

bytesWritten

points to the variable that the driver returns as the number of bytes written. In the event of an error, the *bytesWritten* variable should reflect the actual byte count of the output data.

Description

This method allows the driver to write information to the device. It receives a pointer to the device descriptor of the device being tested, the requested buffer size, a pointer to the buffer address, and a pointer to a variable that holds the number of bytes written. The developer should also update the variable pointed to by the *bytesWritten* parameter before returning to the caller.

Return Values

The return values for this method are determined by the developer.

dev*XXX*Ioctl()

Name

dev*XXX*Ioctl()—allows the driver to perform special operations with the device

Synopsis

```
devXXXIoctl(DEV_DESC *pDevDesc,
                              INT32 function,
                              INT32 argument)
```

Parameters

pDevDesc

is a pointer to a structure that contains all of the registers that are needed to perform an ioctl operation on the device. Using the register definitions in the device descriptor guarantees that the correct device is being accessed.

function

contains the special operation that is to be performed. The actual function value and implementation is device specific.

argument

contains special information that is required by the method being performed. The actual value of the argument is method and device specific. It should be noted that the user is not limited to an integer value as an argument. The argument can be a pointer that is cast as an integer when the method is called. Being a pointer the user can pass a large amount of information to ioctl method.

Description

This method allows the driver to perform special operations with the device. It receives a pointer to the device descriptor of the device being tested, the requested function, and an argument.

Return Values

The return values for this method are determined by the developer.

Device Read and Write Utility Methods

The device read and write utility methods provided by MBIT are added to the device descriptor during device initialization. These are then used by **bitIn()/bitOut()** to access the device hardware addresses.

MBIT provides the following device read and write utility methods:

bitProbeIn8/16/32()

Name

bitProbeIn8/16/32()—reads 8/16/32-bit data from the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeIn8(ULONG addr, UINT32 *pdata);
STATUS bitProbeIn16(ULONG addr, UINT32 *pdata);
STATUS bitProbeIn32(ULONG addr, UINT32 *pdata);
```

Parameters

addr

is the address to read data from.

pdata

is the pointer to a 32-bit location to store data.

Description

The data read is written into a 8/16/32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()/bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitIn8()/bitIn16()/bitIn32()** should be used.

Return Values

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Chapter 5, *MBIT Faults* for more faults.

B

bitProbeOut8/16/32()

Name

bitProbeOu8/16/32()—writes 8/16/32-bit data to the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeOut8(ULONG addr, UINT8 *data);
STATUS bitProbeOut16(ULONG addr, UINT16 *data);
STATUS bitProbeOut32(ULONG addr, UINT32 *data);
```

Parameters

addr

is the address to write data to.

data

is the data to write out.

Description

This method writes 8/16/32-bit data to the designated address.

The data is written into an 8/16/32-bit memory location. The MBIT exception handler is enabled during the data write. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitOut8()/bitOut16()/bitOut32()** should be used.

Return Values

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitProbeInSwap16/32()

Name

bitProbeInSwap16/32()—reads and byte swaps 16/32-bit data from the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeInSwap16(ULONG addr, UINT32 *pdata);
STATUS bitProbeInSwap32(ULONG addr, UINT32 *pdata);
```

Parameters

addr

is the address to read data from.

pdata

is the pointer to a 32-bit location to store data.

Description

This method reads and byte swaps 16/32-bit data from the designated address. The data is read as little/big-endian and loaded as big/little-endian into a 32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()**/**bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitInSwap16()/bitInSwap32()** should be used.

Return Values

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Chapter 5, *MBIT Faults* for more faults.

B

bitProbeOutSwap16/32()

Name

bitProbeOutSwap16/32()—writes and byte swaps 16/32-bit data to the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitProbeOutSwap16(ULONG addr, UINT16 *data);
STATUS bitProbeOutSwap32(ULONG addr, UINT32 *data);
```

Parameters

addr

is the address to write data to.

data

is the data to write out.

Description

This method byte swaps and writes 16/32-bit data to the designated address. The big/little-endian data is written into a 16/32-bit data location as little/big-endian.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitOutSwap16()**/**bitOutSwap32()** should be used.

Return Values

Upon successful completion, this method returns **0**. If an exception occurs, **–1** is returned.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIn8/16/32()

Name

bitIn8/16/32()—reads 8/16/32-bit data from the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitIn8(ULONG addr, UINT32 *pdata);
STATUS bitIn16(ULONG addr, UINT32 *pdata);
STATUS bitIn32(ULONG addr, UINT32 *pdata);
```

Parameters

addr

is the address to read data from.

pdata

is the pointer to a 32-bit location to store data.

Description

This method reads 8/16/32-bit data from the designated address. The data read is written into a 32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()**/**bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is *not* enabled during the data read. If the MBIT exception handler is needed to "catch" exceptions caused by the access to the device, then **bitProbeIn8()**/**bitProbeIn16()**/**bitProbeIn32()** should be used.

Return Values

This method always returns **0**.

Refer to Chapter 5, *MBIT Faults* for more faults.

B

bitOut8/16/32()

Name

bitOut8/16/32()—writes 8/16/32-bit data to the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitOut8(ULONG addr, UINT8 *data);
STATUS bitOut16(ULONG addr, UINT16 *data);
STATUS bitOut32(ULONG addr, UINT32 *data);
```

Parameters

addr

is the address to write data to.

data

is the data to write out.

Description

This method writes 8/16/32-bit data to the designated address. The data is written into an 8/16/32-bit memory location.

The MBIT exception handler is *not* enabled during the data write. If the MBIT exception handler is needed to "catch" exceptions caused by the access to the device, then **bitProbeOut8()/bitProbeOut16()/bitProbeOut32()** should be used.

Return Values

This method always returns **0**.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInSwap16/32()

Name

bitInSwap16/32()—reads and byte swaps 16/32-bit data from the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitInSwap16(ULONG addr, UINT32 *pdata);
STATUS bitInSwap32(ULONG addr, UINT32 *pdata);
```

Parameters

addr

is the address to read data from.

pdata

is the pointer to a 32-bit location to store data.

Description

This method reads and byte swaps 16/32-bit data from the designated address. The data is read as little/big-endian and loaded as big/little-endian into a 32-bit memory location. This method requires a 32-bit memory location for storing the data because it is used by the device utility methods **bitIn()**/**bitOut()**. These methods store the location's contents in the 32-bit value field of the *ADDR_INFO* structure.

The MBIT exception handler is *not* enabled during the data read. If the MBIT exception handler is needed to "catch" exceptions caused by the access to the device, then **bitProbeInSwap16()**/**bitProbeInSwap32()** should be used.

Return Values

This method always returns **0**.

Refer to Chapter 5, *MBIT Faults* for more faults.

B

bitOutSwap16/32()

Name

bitOutSwap16/32()—writes and byte swaps 16/32-bit data to the designated address

Synopsis

```
#include <utilities/bitDeviceUtils.h>
STATUS bitOutSwap16(ULONG addr, UINT16 *data);
STATUS bitOutSwap32(ULONG addr, UINT32 *data);
```

Parameters

addr

is the address to write data to.

data

is the data to write out.

Description

This method byte swaps and writes 16/32-bit data to the designated address. The big/little-endian data is written into a 16/32-bit data location as little/big-endian.

The MBIT exception handler is enabled during the data read. If the MBIT exception handler is not necessary to "catch" exceptions caused by the access to the device, then **bitProbeOutSwap16()**/**bitProbeOutSwap32()** should be used.

Return Values

This method always returns **0**.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitPciWrite32()

Name

bitPciWrite32()—writes 32-bit data to PCI (I/O or memory) space in little-endian mode

Synopsis

```
#include <utilities/bitDeviceUtils.h>
void bitPciWrite32(ULONG addr, UINT32 data);
```

Parameters

addr

is the address to write data to.

data

is the data to write out.

Description

This method writes 32-bit data to PCI (I/O or memory) space in little-endian mode.

Return Values

This method has no return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

B

bitPciRead32()

Name

bitPciRead32()—reads 32-bit data to PCI (I/O or memory) space

Synopsis

```
#include <utilities/bitDeviceUtils.h>
void bitPciRead32(ULONG addr, UINT32 *pdata);
```

Parameters

addr

is the address to read data from.

pdata

is the pointer to a 32-bit location to store data.

Description

This method reads 32-bit data from PCI (I/O or memory) space.

Return Values

This method has no return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

Utility Methods' Reference Pages

C

This appendix provides detailed information about the utility methods mentioned in Chapter 4, *Utility Methods*.

Cache Utility Methods

MBIT provides the following cache utility methods:

C

bitDataCacheEnable()

Name

bitDataCacheEnable()—enables the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheEnable(void)
```

Parameters

No input parameters are required by this method.

Description

This method invalidates and then enables the L1 data cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, successful
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheDisable()

Name

bitDataCacheDisable()—disables the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheDisable(void)
```

Parameters

No input parameters are required by this method.

Description

This method flushes and then disables the L1 data cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_DATA_CACHE_NOT_ENABLED—data cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheIsEnabled()

Name

bitDataCacheIsEnabled()—gives the enabled state of the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BOOL bitDataCacheIsEnabled(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the boolean enable state of L1 data cache.

Return Values

TRUE—the L1 data cache is enabled
FALSE—the L1 data cache is *not* enabled

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheFlush()

Name

bitDataCacheFlush()—flushes the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheFlush(void)
```

Parameters

No input parameters are required by this method.

Description

This method flushes the entire L1 data cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_DATA_CACHE_NOT_ENABLED—data cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheFlushInvalidate()

Name

bitDataCacheFlushInvalidate()—flushes and invalidates the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheFlushInvalidate(void)
```

Parameters

No input parameters are required by this method.

Description

This method flushes and invalidates the entire L1 data cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_DATA_CACHE_NOT_ENABLED—data cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheInvalidate()

Name

bitDataCacheInvalidate()—invalidates the L1 data cache for a range of memory

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheInvalidate(void *address, UINT bytes)
```

Parameters

address

is the virtual address to begin invalidation.

bytes

is the number of bytes to invalidate.

Description

This method invalidates the L1 data cache for a range of memory.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheLock()

Name

bitDataCacheLock()—locks the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheLock(void)
```

Parameters

No input parameters are required by this method.

Description

This method locks the L1 data cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitDataCacheUnlock()

Name

bitDataCacheUnlock()—unlocks the L1 data cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitDataCacheUnlock(void)
```

Parameters

No input parameters are required by this method.

Description

This method unlocks the L1 data cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInstCacheEnable()

Name

bitInstCacheEnable()—invalidates and enables the L1 instruction cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheEnable(void)
```

Parameters

No input parameters are required by this method.

Description

This method invalidates and then enables the L1 instruction cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInstCacheDisable()

Name

bitInstCacheDisable()—disables the L1 instruction cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheDisable(void)
```

Parameters

No input parameters are required by this method.

Description

This method disables the L1 instruction cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_CACHE_NOT_ENABLED—cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInstCacheIsEnabled()

Name

bitInstCacheIsEnabled()—gives the enabled state of the L1 instruction cache

Synopsis

```
<utilities/bitCacheUtils.h>
BOOL bitInstCacheIsEnabled(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the boolean enable state of the L1 instruction cache.

Return Values

TRUE—the L1 instruction cache is enabled
FALSE—the L1 instruction cache is *not* enabled

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInstCacheLock()

Name

bitInstCacheLock()—locks the L1 instruction cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheLock(void)
```

Parameters

No input parameters are required by this method.

Description

This method locks the L1 instruction cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitInstCacheUnlock()

Name

bitInstCacheUnlock()—unlocks the L1 instruction cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitInstCacheUnlock(void)
```

Parameters

No input parameters are required by this method.

Description

This method unlocks the L1 instruction cache(s).

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheSizeGet()

Name

bitL2CacheSizeGet()—returns the size (in bytes) of the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
UINT bitL2CacheSizeGet(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the size (in bytes) of the L2 cache configured by the hardware (the L2 cache controller).

Return Values

numBytes—L2 cache size in bytes
0—size cannot be determined

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheEnable()

Name

bitL2CacheEnable()—enables the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheEnable(void)
```

Parameters

No input parameters are required by this method.

Description

This method enables the L2 cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheDisable()

Name

bitL2CacheDisable()—disables the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheDisable(void *pFlushBuffer, int buffSize)
```

Parameters

pFlushBuffer

is a pointer to a cacheable memory block twice the size of the L2 cache. If **NULL**, a local buffer is used and *buffSize* is ignored. The local buffer will not be guaranteed to be cacheable if BATs or page tables have been altered prior to calling.

buffSize

is the size of the flush buffer in bytes. It must be equal to twice the L2 cache size.

Description

This method flushes, invalidates and then disables the L2 cache. If a **NULL** buffer is provided, then a local buffer is allocated for use and freed before return. If the size of a given non-**NULL** buffer is not equal to twice the L2 cache size, then no action is taken by the method.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_CACHE_NOT_ENABLED—cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheOn()

Name

bitL2CacheOn()—enables the L2 cache without other actions

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheOn(void)
```

Parameters

No input parameters are required by this method.

Description

This method enables the L2 cache without any flushing or invalidation.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheOff()

Name

bitL2CacheOff()—disables the L2 cache without other actions

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheOff(void)
```

Parameters

No input parameters are required by this method.

Description

This method disables the L2 cache without any flushing or invalidation.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_CACHE_NOT_ENABLED—cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheIsEnabled()

Name

bitL2CacheIsEnabled()—gives the enabled state of the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BOOL bitL2CacheIsEnabled(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the boolean enable state of L2 cache.

Return Values

TRUE—the L2 cache is enabled
FALSE—the L2 cache is *not* enabled

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheFlush()

Name

bitL2CacheFlush()—flushes the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheFlush(void pFlushBuffer, int buffSize)
```

Parameters

pFlushBuffer

is a pointer to a cacheable memory block twice the size of the L2 cache. If **NULL**, a local buffer is used and *buffSize* is ignored. The local buffer will not be guaranteed to be cacheable if BATs or page tables have been altered prior to calling.

buffSize

is the size of the flush buffer in bytes. It must be equal to twice the L2 cache size.

Description

This method flushes the entire L2 cache. If a **NULL** buffer is provided, then a local buffer is allocated for use and freed before return. If the size of a given non-**NULL** buffer is not equal to twice the L2 cache size, then no action is taken by the routine.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_CACHE_NOT_ENABLED—cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheFlushInvalidate()

Name

bitL2CacheFlushInvalidate()—flushes and invalidates the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheFlushInvalidate(void)
```

Parameters

No input parameters are required by this method.

Description

This method flushes and invalidates the entire L2 cache.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheInvalidate()

Name

bitL2CacheInvalidate()—invalidates the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheInvalidate(void)
```

Parameters

No input parameters are required by this method.

Description

This method invalidates the entire L2 cache. Any modified data in the L2 cache is lost unless it is flushed first.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheLock()

Name

bitL2CacheLock()—locks the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheLock(void)
```

Parameters

No input parameters are required by this method.

Description

This method locks the L2 cache if the L2 cache supports it.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_CACHE_LOCK_NOT_SUPPORTED—cache locking is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheUnlock()

Name

bitL2CacheUnlock()—unlocks the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheUnlock(void)
```

Parameters

No input parameters are required by this method.

Description

This method unlocks the L2 cache if the L2 cache supports it.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_CACHE_ROUTINE_NOT_SUPPORTED—cache routine is not supported
BIT_CACHE_LOCK_NOT_SUPPORTED—cache locking is not supported

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheIsLockable()

Name

bitL2CacheIsLockable()—gives the lock capability of the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BOOL bitL2CacheIsLockable(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the boolean lock capability of the L2 cache.

Return Values

TRUE—the L2 cache is lockable
FALSE—the L2 cache is *not* lockable

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheFill()

Name

bitL2CacheFill()—fills the L2 cache with the specified pattern

Synopsis

```
<utilities/bitCacheUtils.h>
BIT_FAULT bitL2CacheFill(UINT *bufPtr, UINT *castOutBuf, UINT
wordCount, UINT pattern, int modifier)
```

Parameters

bufPtr

is a pointer to a cacheable buffer to fill.

castOutBuf

is a pointer to a cacheable buffer to fill that causes data to be cast-out from the L1 data cache. This may be necessary depending on the L1/L2 cache controller design (victim caches).

wordCount

is the number of 32-bit words to fill.

pattern

is the pattern to fill the buffer with.

modifier

is the value with which to modify the pattern after each write to the buffer.

Description

This method fills the specified number of 32-bit words in the L2 cache with the specified pattern. The pattern is incremented by the *modifier* after every write.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_PROCESSOR_NOT_SUPPORTED—unknown processor type
BIT_INVALID_TEST_PARAM—invalid test parameter was supplied
BIT_CACHE_NOT_ENABLED—cache must be enabled before calling cache function

Refer to Chapter 5, *MBIT Faults* for more faults.

bitL2CacheIsWritebackCapable()

Name

bitL2CacheIsWritebackCapable()—gives the write-back capability of the L2 cache

Synopsis

```
<utilities/bitCacheUtils.h>
BOOL bitL2CacheIsWritebackCapable(void)
```

Parameters

No input parameters are required by this method.

Description

This method returns the boolean write-back capability of the L2 cache.

Return Values

TRUE—the L2 cache supports write-back
FALSE—the L2 cache *does not* support write-back

Refer to Chapter 5, *MBIT Faults* for more faults.

Diagnostic Device Utility Methods

MBIT provides the following diagnostic device utility methods:

getDeviceDescriptor()

getDevTablePtr()

bitTrackChanges()

bitIn()

bitOut()

getDeviceDescriptor()

Name

getDeviceDescriptor()—returns a pointer to the device descriptor

Synopsis

```
<utilities/bitDeviceUtils.h>
DEV_DESC* getDeviceDescriptor(BIT_LOGICAL_DEVICE device)
```

Parameters

device

is the logical device to retrieve.

Description

This method takes a logical device number and returns a pointer to the device descriptor. The device descriptor contains all the information needed to interface with the device.

Return Values

DEV_DESC—a pointer to the device descriptor
NULL—the device descriptor is invalid

Refer to Chapter 5, *MBIT Faults* for more faults.

getDevTablePtr()

Name

getDevTablePtr()—returns a pointer to the device descriptor

Synopsis

```
<utilities/bitDeviceUtils.h>
DEV_DESC* getDevTablePtr(BIT_LOGICAL_DEVICE device)
```

Parameters

device

is the logical device to retrieve.

Description

This method takes a logical device number and returns a pointer to the device descriptor. The device descriptor contains all the information needed to interface with the device. This routine should only be used by routines initializing device descriptors.

Return Values

DEV_DESC—a pointer to the device descriptor
NULL—the device descriptor is invalid

Refer to Chapter 5, *MBIT Faults* for more faults.

bitTrackChanges()

Name

bitTrackChanges()—turns register bit change tracking on or off

Synopsis

```
<utilities/bitDeviceUtils.h>
BIT_FAULT bitTrackChanges(ADDR_INFO *reg, UINT on)
```

Parameters

reg

is the pointer to the location's *ADDR_INFO* structure.

on

is a boolean value to indicate starting or stopping changes tracking. **TRUE** starts tracking, **FALSE** stops tracking.

Description

This method starts or stops tracking register bits changes during hardware access.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_INVALID_TEST_PARAM—invalid test parameter was supplied

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIn()

Name

bitIn()—reads from the location described by the *ADDR_INFO* structure

Synopsis

```
<utilities/bitDeviceUtils.h>
BIT_FAULT bitIn(ADDR_INFO *reg)
```

Parameters

reg

is the pointer to the location's *ADDR_INFO* structure.

Description

This method reads from the location described by the *ADDR_INFO* structure passed to the method. If an exception is caused by the read, **BIT_BUS_ERROR** is returned to indicate an exception occurred. If a device is not enabled and enable/disable methods are defined, then the device is enabled, written to, and then disabled. The value read from the location is put into the *val* field of the *ADDR_INFO* structure passed to the method.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_BUS_ERROR—device did not respond to transfer
BIT_DEVICE_ENABLE_FAULT—failed to enable a disabled device
BIT_DEVICE_DISABLE_FAULT—failed to disable an enabled device
BIT_INVALID_TEST_PARAM—invalid test parameter was supplied
BIT_INVALID_DEVICE_DESC—device descriptor has invalid field (configuration error)

Refer to Chapter 5, *MBIT Faults* for more faults.

bitOut()

Name

bitOut()—writes to the location described by the *ADDR_INFO* structure

Synopsis

```
<utilities/bitDeviceUtils.h>
BIT_FAULT bitOut(ADDR_INFO *reg)
```

Parameters

reg

is the pointer to the location's *ADDR_INFO* structure.

Description

This method writes to the location described by the *ADDR_INFO* structure passed to the method. If an exception is caused by the write, **BIT_BUS_ERROR** is returned to indicate an exception occurred. If a device is not enabled and enable/disable methods are defined, then the device is enabled, written to, and then disabled. The value actually written to the location is the value in the *val* field of the *ADDR_INFO* structure passed to the method.

Return Values

BIT_NO_FAULT_DETECTED—no fault detected, success
BIT_BUS_ERROR—device did not respond to transfer
BIT_DEVICE_ENABLE_FAULT—failed to enable a disabled device
BIT_DEVICE_DISABLE_FAULT—failed to disable an enabled device
BIT_INVALID_TEST_PARAM—invalid test parameter was supplied
BIT_INVALID_DEVICE_DESC—device descriptor has invalid field (configuration error)

Refer to Chapter 5, *MBIT Faults* for more faults.

Interrupt Utility Methods

MBIT provides the following interrupt utility methods:

bitIntLock()

bitIntUnlock()

bitForceIntUnlock()

bitIntConnect()

isBitIntEnabled()

bitIntVectorSet()

bitIntEnable()

bitIntDisable()

bitIntLock()

Name

bitIntLock()—locks out all interrupts

Synopsis

```
<utilities/bitExceptionUtils.h>
void bitIntLock(void)
```

Parameters

No input parameters are required by this method.

Description

This methods increments an interrupts-locked reference count and if interrupts are not locked, it locks all interrupts.

Return Values

No return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIntUnlock()

Name

bitIntUnlock()—re-enables interrupts

Synopsis

```
<utilities/bitExceptionUtils.h>
void bitIntUnlock(void)
```

Parameters

No input parameters are required by this method.

Description

This method decrements the interrupts-locked reference count incremented by **bitIntLock()** and if the reference count is **0**, it unlocks all interrupts.

Return Values

No return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitForceIntUnlock()

Name

bitForceIntUnlock()—forces the re-enable of interrupts

Synopsis

```
<utilities/bitExceptionUtils.h>
void bitForceIntUnlock(void)
```

Parameters

No input parameters are required by this method.

Description

This method sets the interrupts-locked reference count to **0** and unlocks all the interrupts locked by **bitIntLock()**.

Return Values

No return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIntConnect()

Name

bitIntConnect()—connects an MBIT interrupt handler to the MBIT interrupt table

Synopsis

```
<kernel/kernelExceptionUtils.h>
STATUS bitIntConnect(VOIDFUNCPTR *vector,
                     VOIDFUNCPTR routine,
                     int param)
```

Parameters

vector

is the interrupt vector to connect.

routine

is the routine to connect to the specified vector.

param

is the parameter provided to the specified routine (when an interrupt occurs).

Description

This method connects an MBIT interrupt handler to the MBIT interrupt table. In software, there may be up to 256 interrupts connected, however, hardware may limit the actual number available. This method only connects one handler to any interrupt vector at any one time. All interrupt handlers not connected, and use this method, remain as they were installed by the operating system (chained handlers remain chained). To disconnect the handler from the vector, use **bitIntVectorSet()** with a **NULL** entry.

Return Values

OK—no fault detected, success
–1—not successful; vector < **0** or > 0xff

Refer to Chapter 5, *MBIT Faults* for more faults.

isBitIntEnabled()

Name

isBitIntEnabled()—checks if interrupts are enabled on a level

Synopsis

```
<kernel/kernelExceptionUtils.h>
STATUS isBitIntEnabled(int level)
```

Parameters

level

is the interrupt level to be tested.

Description

This method checks if an interrupt is enabled on a specified interrupt level.

Return Values

TRUE—enabled
FALSE—not enabled
–1—interrupt level could not be resolved;
level < **0**
level > **ERR_INTERRUPT_BASE**

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIntVectorSet()

Name

bitIntVectorSet()—saves a vector entry in the MBIT interrupt table

Synopsis

```
<kernel/kernelExceptionUtils.h>
STATUS bitIntVectorSet(VOIDFUNCPTR *vector, INT32 *entry)
```

Parameters

vector

is the interrupt vector to connect.

entry

is the method to connect to the specified vector.

Description

This method sets a vector entry in the MBIT interrupt table.

Return Values

No return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIntEnable()

Name

bitIntEnable()—enables an interrupt level

Synopsis

```
<kernel/kernelExceptionUtils.h>
INT32 bitIntEnable(INT32 level)
```

Parameters

level

is the interrupt level to enable.

Description

This function enables the interrupt level.

Return Values

OK—no fault detected, success
–1—interrupt level could not be resolved;
level < **0**
level > **ERR_INTERRUPT_BASE**

Refer to Chapter 5, *MBIT Faults* for more faults.

bitIntDisable()

Name

bitIntDisable()—disables an interrupt level

Synopsis

```
<kernel/kernelExceptionUtils.h>
INT32 bitIntDisable(INT32 level)
```

Parameters

level

is the interrupt level to disable.

Description

This function disables the interrupt level.

Return Values

OK—no fault detected, success
–1—interrupt level could not be resolved;
level < **0**
level > **ERR_INTERRUPT_BASE**

Refer to Chapter 5, *MBIT Faults* for more faults.

Time Utility Methods

This section contains the following time-related methods:

bitUsDelay()

bitMsDelay()

bitUsDelay()

Name

bitUsDelay()—delays for a requested number of microseconds

Synopsis

```
<utilities/bitTimeUtils.h>
void bitUsDelay(UINT32 micro)
```

Parameters

micro

is the number of microseconds to delay.

Description

This method delays *micro* number of microseconds. Note that the resolution may be more than a microsecond, so this call gives the smallest time-out possible in those cases.

Return Values

No return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

bitMsDelay()

Name

bitMsDelay()—delays for a requested number of milliseconds

Synopsis

```
<utilities/bitTimeUtils.h>
void bitMsDelay(UINT32 milli)
```

Parameters

milli

is the number of milliseconds to delay.

Description

This method delays for *milli* number of milliseconds.

Return Values

No return values.

Refer to Chapter 5, *MBIT Faults* for more faults.

Installing MBIT with Tornado 2.1 and VxWorks

D

This chapter provides instructions on how to install the board and system level versions of MBIT with the Tornado 2.1 development system. It also explains how to modify your MVME5100 board support package (BSP) to get the full use out of your new diagnostic software.

Installing MBIT from the CD-ROM

The MBIT CD-ROM contains four files: **README.txt**, **MBITLicense.txt**, **VDD.html**, and **objects.tar**.

1. Please read the license agreement in **MBITLicense.txt**. You must accept this agreement before installing MBIT.
2. Extract **objects.tar** in your Tornado 2.1 installation directory.

Installing MBIT on a Microsoft Windows Platform

If you are installing MBIT on a Microsoft Windows® platform for VxWorks development, and using the Winzip program to extract the **object.tar** file, change the Winzip configuration as follows:

1. Select **Options/Configuration**.
2. Select the **Miscellaneous** tab.
3. Under "Other," uncheck **TAR file smart CR/LF conversion**.
4. Select **OK**.

Creating a VxWorks Image with the MBIT API

To create a VxWorks image with MBIT included, and the necessary kernel extensions for MBIT, do the following:

Note These instructions assume **C:\Tornado** is the Tornado installation directory.

1. Create a directory for the project. The following instructions assume the directory created is **D:\BIT**.
2. Copy the **C:\Tornado\target\config\mv5100\configNet.h**, **C:\Tornado\target\src\MBIT\mv5100\50BIT-VME-slave.cdf**, **C:\Tornado\target\src\MBIT\mv5100\50MVME5100-512MB-Memory.cdf**, **C:\Tornado\target\src\MBIT\mv5100\60MVME5100.cdf**, **C:\Tornado\target\src\MBIT\mv5100\65MBIT.cdf**, **C:\Tornado\target\src\MBIT\mv5100\67BITBSPPARAMS.cdf**, and **C:\Tornado\target\src\MBIT\mv5100\70BIT.cdf** files to **D:\BIT**.
3. Edit **configNet.h** as described in *GD82559ER Ethernet Testing* on page D-9.

Building a VxWorks Image

To build a VxWorks image, complete the following steps:

1. Start Tornado.
2. Select **File/New Project...**.
3. Select **Create a bootable VxWorks image (custom configured)**. Choose **OK**.
4. Enter project name, description, and workspace as desired.
5. Enter project location as **D:\BIT** (the .cdf files must be in the same directory as the project file). Choose **Next**.
6. Select **A BSP**.

7. Select **mv5100** from the drop down list next to the "A BSP" radio button. Choose **Next** and then **Finish**.
8. Select the **VxWorks** tab in the workspace window.
9. Select the + next to the new project to display the project options.
10. Display "development tool components/WDB agent components/select WDB connection" using the + controls.
11. Right click on "WDB END driver connection" and select **Exclude WDB END driver connection**. Choose **OK**.
12. Display "development tool components/WDB agent components/select WDB mode" using the + controls.
13. Right click on "WDB system debugging" and select **Exclude WDB system debugging**. Choose **OK**.
14. Display "development tool components/WDB agent components/WDB agent services" using the + controls.
15. Right click on "WDB system agent hardware fpp support" and select **Exclude WDB system agent hardware fpp support**. Choose **OK**.
16. Display "development tool components/WDB agent components/select WDB connection" using the + controls.
17. Right click on "WDB network connection" and select **Include WDB network connection**. Choose **OK**.
18. Display "hardware/Diagnostics/BIT Diagnostics/BIT API Install Selection" using the + controls.
19. Right click on "BIT API Install" and select **Include 'BIT API Install...'**. Choose **OK**.
20. Display "hardware/Diagnostics/BIT Diagnostics/BIT API Selection" using the + controls.
21. Right click on "BIT API" and select **Include BIT API...**. Choose **OK**.

22. (Optional) For VME location monitor testing, setup a VME location monitor window as described in *VME Location Monitor Window Setup* on page D-12.
23. Select the **Builds** tab in the workspace window.
24. Click the + next to project name to display the "default" build.
25. Select the **default** build.
26. Right click in the workspace window and select **Properties** from the pop-up menu.
27. Select the **C/C++ compiler** tab and add **-DINCLUDE_I8250_SIO** and **-IC:/Tornado/target/h/MBIT** to the list of compiler options.
28. Click on **Apply**.
29. Select the **Assembler** tab and add **-IC:/Tornado/target/h/MBIT** to the list of compiler options.
30. Click on **Apply** and **OK**.
31. Select the **Files** tab in the workspace window.
32. Right click on the new project and select **Add Files...**.
33. Browse to **C:\Tornado\target\src\MBIT\mv5100** and select **kernelExceptionUtilsAsm.s** in the "Add Source File to..." dialog box. Choose **Add**.
34. Right click on the new project and select **Dependencies...**. Choose **OK**.
35. Wait for the dependency building to complete.
36. Right click on the new project and select **ReBuild All (VxWorks)**.
37. This produces "default\vxWorks" under the new project directory.

Building a VxWorks VME Slave Image

The following instructions assume Tornado 2.1 is installed on Windows NT in **C:\Tornado** and the project for the VME slave image is in **C:\Tornado\target\proj\vmeslave**.

Note: It is *not* necessary to complete step 2 of *Creating a VxWorks Image with the MBIT API* on page D-2 prior to building this slave kernel.

To build a VxWorks VME slave image, complete the following steps:

1. After extracting the **objects.tar** file **mkdir C:\Tornado\target\proj\vmeslave**, copy **C:\Tornado\target\src\MBIT\mv5100\50BIT-VME-slave.cdf**, **C:\Tornado\target\src\MBIT\mv5100\65MBIT.cdf**, and **C:\Tornado\target\src\MBIT\mv5100\67BITBSPPARAMS.cdf** to **C:\Tornado\target\proj\vmeslave**.
2. Start Tornado 2.1.
3. Select the **New** tab in the "Create Project in New/Existing Workspace" dialog box.
4. Select **Create a bootable VxWorks image (custom configured)** in the New tab area. Choose **OK**.
5. In the "Create a bootable VxWorks image (custom configured): step 1" dialog box, change the location to **C:\Tornado\target\config\proj\vmeslave**. Choose **Next**.
6. In the "Create a bootable VxWorks image (custom configured): step 2" dialog box, select the **A BSP** radio button.
7. Select **mv5100** from the drop-down list. Choose **Next**.
8. In the "Create a bootable VxWorks image (custom configured): step 2" dialog box, choose **Finish**.
9. In the "Workspace: ..." dialog box, select the **VxWorks** tab.
10. Using the **+** controls, navigate to and select **hardware/buses/Special BIT VME Slave Memory Configuration**. Right click on "Special BIT VME Slave Memory Configuration."

11. In the pop-up menu, select **Include 'Special BIT VME Slave Memory Configuration'**.
12. In the "Include Component(s)" dialog box, choose **OK**.
13. In the "Workspace: ..." dialog box, choose the **Builds** tab.
14. Using the + controls, navigate to and select the **default** build for the new project. Right click in the "Workspace: ..." window.
15. In the pop-up menu, choose **Dependencies...**.
16. In the "Dependencies" dialog box, choose **OK**.
17. Right click in the "Workspace: ..." window. In the pop-up menu, select **Rebuild all (vxWorks)**.
18. In the "Dependencies" dialog box, choose **OK**.

Configuring the Target

This information is in the VxWorks documentation and **target/config/mv5100/target.nr** from the MVME5100 BSP.

Booting the Target

This information is in the VxWorks documentation and **target/config/mv5100/target.nr** from the MVME5100 BSP.

Modifying the Image

The file **kernelExceptionUtils.c** in the kernel directory needs to be compiled into the VxWorks image. This is required because the VxWorks image files cannot be compiled with the **-mlongcall** option. In other words, the Tornado distribution CD-ROM contains only objects for the required files and a source is not available for the exception handling functions.

Modifying the MVME5100 BSP

After installing MBIT, some modifications to the MVME5100 BSP are required to run the complete set of MBIT subtests. Modifications must be made to *Flash Memory Testing* and *GD82559ER Ethernet Testing*.

Flash Memory Testing

Access to each block of Flash memory is software programmable by three software programmable control register bits: an overall enable, a write enable, and a reset vector enable (refer to the *MVME5100 Single Board Computer Programmer's Reference Guide*). At reset, the default access settings enable the first 1MB of Flash A to $FF000000-$FF100000 and Flash B to $FF400000-$FF500000. In addition, Flash B is enabled at $FFF00000-$FFFFFFFF. Since the visibility test does not modify the settings of these register bits, the MV5100 BSP needs modification to reflect the default settings for each block of Flash.

The following changes need to be made to **\\Tornado\target\config\mv5100\mv5100.h**

Note An exclamation point (!) indicates where changes need to be made.

Original:

```
  #define       HAWK_PHB_BASE_ADRS      0xfeff0000
  #define       HAWK_PHB_REG_SIZE       0x00010000
! #define       FLASH_BASE_ADRS         0xF4000000
! #define       FLASH_MEM_SIZE          0x01000000
/* MPIC configuration defines */
```

Modify and Add:

```
  #define       HAWK_PHB_BASE_ADRS      0xfeff0000
  #define       HAWK_PHB_REG_SIZE       0x00010000
! #define       FLASH_BASE_ADRS         0xFF000000
```

```
! #define       FLASH_MEM_SIZE          0x00100000

! #define       FLASHB_BASE_ADRS        0xFF400000

! #define       FLASHB_MEM_SIZE         0x00100000

! #define       FLASH_IO_BASE_ADRS      0xFFF00000

! #define       FLASH_IO_MEM_SIZE       0x00100000

/* MPIC configuration defines */

The following changes need to be made to

\\Tornado\target\config\mv5100\sysLib.c
```

Original:

```
   FLASH_MEM_SIZE,

   VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,

   VM_STATE_VALID | VM_STATE_WRITABLE |
VM_STATE_CACHEABLE_NOT

!  }

};

int sysPhysMemDescNumEnt = NELEMENTS (sysPhysMemDesc);
```

Modify and Add:

```
    FLASH_MEM_SIZE,

    VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,

    VM_STATE_VALID | VM_STATE_WRITABLE |
VM_STATE_CACHEABLE_NOT

! },

! {

!   (void *) FLASHB_BASE_ADRS,

!   (void *) FLASHB_BASE_ADRS,
```

```
!   FLASHB_MEM_SIZE,

!   VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,

!   VM_STATE_VALID | VM_STATE_WRITABLE  |
VM_STATE_CACHEABLE_NOT

! },

! {

!   (void *) FLASH_IO_BASE_ADRS,

!   (void *) FLASH_IO_BASE_ADRS,

!   FLASH_IO_MEM_SIZE,

!   VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,

!   VM_STATE_VALID | VM_STATE_WRITABLE |
VM_STATE_CACHEABLE_NOT

! }

};

int sysPhysMemDescNumEnt = NELEMENTS (sysPhysMemDesc);
```


GD82559ER Ethernet Testing

The GD82559ER Ethernet tests require they be run before the VxWorks Ethernet driver is started. One way to accomplish this is to prevent the drivers from starting during boot and then starting them after completing the Ethernet testing. **65MBIT.cdf** redefines the **INCLUDE_END** component to remove the **INIT_RTN** of **usrEndLibInit**. In addition, **endDevTbl** in **target/config/mv5100/configNet.h** must be modified to set the processed field of the entries for the GD82559ER instances to **TRUE** so they are skipped during boot. After booting and running the GD82559ER tests, the VxWorks driver may be started by setting the processed field to **FALSE** and executing **usrEndLibInit**, then attaching and configuring the interface.

To modify **endDevTbl**, copy **target/config/mv5100/configNet.h** to the project directory and edit it as follows:

Note An exclamation point (!) indicates where changes need to be made.

Original:

```
END_TBL_ENTRY endDevTbl [] =

{

!  { 0, END_LOAD_FUNC, END_LOAD_STRING, END_BUFF_LOAN, NULL,
FALSE},

#ifdef INCLUDE_SECONDARY_ENET

!  { 1, END_LOAD_FUNC, END_LOAD_STRING, END_BUFF_LOAN, NULL,
FALSE},

#endif /* INCLUDE_SECONDARY_ENET */

   { 0, END_TBL_END, NULL, 0, NULL, FALSE},

};
```

New:

```
END_TBL_ENTRY endDevTbl [] =

{

!  { 0, END_LOAD_FUNC, END_LOAD_STRING, END_BUFF_LOAN, NULL,
TRUE},

#ifdef INCLUDE_SECONDARY_ENET

!  { 1, END_LOAD_FUNC, END_LOAD_STRING, END_BUFF_LOAN, NULL,
TRUE},

#endif /* INCLUDE_SECONDARY_ENET */

   { 0, END_TBL_END, NULL, 0, NULL, FALSE},

};
```

To start the Ethernet interfaces after testing, code similar to the following (with appropriate changes to the network mask and IP address) may be used:

```
#include <vxWorks.h>
#include <end.h>
#include <config.h>

void startEthernet(void)

{

   int i;
   extern END_TBL_ENTRY endDevTbl[];

   for (i = 0; endDevTbl[i].endLoadFunc != END_TBL_END; i++)

   {

      endDevTbl[i].processed = FALSE;

   }

   usrEndLibInit();

   ipAttach(0,"er");
   ifMaskSet("er0", 0xffffff00);
   ifAddrSet("er0","192.168.0.3");

#ifdef INCLUDE_SECONDARY_ENET

   ipAttach(1,"er");

   ifMaskSet("er1", 0xffffff00);

   ifAddrSet("er1","192.168.1.3");

#endif

}
```

VME Location Monitor Window Setup

Complete the steps in this section to set up a VME location monitor window to allow testing of the VME location monitor. The optional steps below (that is, 5, 6,and 7) are only required if a VME location monitor window has not been previously configured for the user application. The optional steps require adding code to *sysPhysMemDesc[]* in **sysLib.c** to configure the location monitor window.

The VME location monitor window is described by the following parameters:

BIT_VME_LM_SLV_SIZE—VME location monitor slave size (default = 0x00001000)

BIT_VME_LM_MSTR_SIZE—VME location monitor master size (default = 0x00001000)

BIT_VME_LM_MSTR_LOCAL—VME location monitor master local address (default = (**VME_RAI_MSTR_LOCAL+VME_RAI_MSTR_SIZE**)

BIT_VME_LM_MSTR_BUS—VME location monitor master bus address (default = (**VME_RAI_MSTR_BUS+VME_RAI_MSTR_SIZE**)

To configure the location monitor window complete the following steps:

1. Display **hardware/Diagnostics** using the + controls.
2. Right click on "BIT VME Location Monitor Window Description Parameters" and select **Properties...**.
3. Select the **Params** tab in the "Properties..." window.
4. Modify the parameters described above to configure the location monitor window. Choose **OK**.

   Note If a VME location monitor has not been configured, the defaults may be used. Otherwise, modify parameters to match existing VME location monitor window.

5. Edit **Tornado\target\config\mv5100\sysLib.c** to add the following to *sysPhysMemDesc[]*:

```
{
(void *) BIT_VME_LM_MSTR_LOCAL,
(void *) BIT_VME_LM_MSTR_LOCAL,
BIT_VME_LM_MSTR_SIZE,
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID | VM_STATE_WRITABLE |
VM_STATE_CACHEABLE_NOT
}
```

6. Display **hardware/Diagnostics** using the + controls.
7. Right click on "BIT VME Location Monitor Window Setup" and select **Include 'BIT VME Location Monitor Window Setup'**. Choose **OK**.

Known Issues

E

This release of the MBIT diagnostic software (1.01) has the following known issues:

Installation

Please refer to Appendix D, *Installing MBIT with Tornado 2.1 and VxWorks*, for the most up to date installation instructions. Your CD-ROM may not have the latest installation updates.

Subtest Results

1. For the MVME5110 models only:

   The interrupt controller test list reports "MPIC interrupt controller marginal." for test #1. The test plan indicates that the expected result should be
   "Operation succeeded – Test successful.".

2. For the MVME5106 models only:

   The Ethernet test list for Ethernet device 2 reports "Data miscompare on write and read sequence." for test #6. The test plan indicates that the expected result should be
   "Operation succeeded – Test successful.".

   The parameter verification for the Ethernet External Loopback subtest on Ethernet device 2 reports "Data miscompare on write and read sequence." for tests #2 and #6. The test plan indicates the expected result should be
   "Operation succeeded – Test successful.".

3. For the MVME5101/5107/5110 models:

   For the SCSI subtests with fault level 3, all tests report the correct expected results, but additional VxWorks output indicates "interrupt: MPIC Spurious Interrupt!". Only MBIT test results are expected; any additional VxWorks output indicates an error condition.

   For the L2 Cache Lock subtest with fault level 3, the test reports "Operation succeeded – Test successful.". The test plan indicates that the expected result should be "Data miscompare on write and read sequence.".

For the MVME5100/5101/5106/5107/5110 models:

For the L2 Cache Invalidate test with fault level 3, test reports "Operation succeeded – Test successful.". The test plan indicates that the expected result should be "Data miscompare on write and read sequence.".

For the Serial Port 3 tests, the monitored output from the device was incomplete.

Related Documentation

F

Motorola Computer Group Documents

The Motorola publications listed below are referenced in this manual. You can obtain paper or electronic copies of Motorola Computer Group publications by:

- ❏ Contacting your local Motorola sales office
- ❏ Visiting Motorola Computer Group's World Wide Web literature site, http://www.motorola.com/computer/literature

Table F-1. Motorola Computer Group Documents

Document Title	Motorola Publication Number
Motorola Built-In Test (MBIT) Diagnostic Software Test Reference Guide	MBITA/RM
MVME5100 Single Board Computer Installation and Use	V5100A/IH
MVME5100 Single Board Computer Programmer's Reference Guide	V5100A/PG
IPMC712/761 Module Installation and Use	VIPMCA/IH
MVME712M Transition Module Installation and Use	MVE712MA/IH
MVME761 Transition Module Installation and Use	VME761A/IH

To obtain the most up-to-date product information in PDF or HTML format, visit http://www.motorola.com/computer/literature.

Manufacturers' Documents

For additional information, refer to the following table for manufacturers' data sheets or user's manuals. As an additional help, a source for the listed document is provided. Please note that, while these sources have been verified, the information is subject to change without notice.

Table F-2. Manufacturers' Documents

Document Title and Source	Publication Number
PowerPC™ Microprocessor Family: The Programming Environment for 32-Bit Microprocessors Literature Distribution Center for Motorola Telephone: 1-800- 441-2447 FAX: (602) 994-6430 or (303) 675-2150 Web Site: http://e-www.motorola.com/webapp/DesignCenter/ E-mail: ldcformotorola@hibbertco.com OR IBM Microelectronics Programming Environment Manual Web Site: http://www.chips.ibm.com/techlib/products/powerpc/manuals	MPCFPE/AD G522-0290-01
LSI53C895A PCI to Ultra2 SCSI Controller Technical Manual LSI Logic Corporation http://www.lsilogic.com/techlib/techdocs/storage_stand_prod/PCISCSICont/Chips/895a.pdf	v2.1
DS1621 Thermometer Data Sheet Dallas Semiconductor http://www.dalsemi.com	DS1621

URLs

The following URLs (uniform resource locators) may provide helpful sources of additional information about this product, related services, and development tools. Please note that, while these URLs have been verified, they are subject to change without notice.

- Motorola Computer Group, http://www.motorola.com/computer
- Motorola Computer Group OEM Services, http://www.motorola.com/computer/support
- Wind River Systems, Inc., http://www.windriver.com

Index

C

D

INDEX

E

F

INDEX

G

H

I

K

L

M

INDEX

N

O

P

R

S

INDEX

T

U

V