Self-Certification Runtime Test Utility Users Guide

August 2007

Revision 1.0

Contents

1 Introduction 1
1.1 Overview 1
1.2 Related Information 1
1.3 Terms 2
1.4 Conventions Used in this document 5
1.4.1 Data Structure Descriptions 5
1.4.2 Pseudo-code Conventions 6
1.4.3 Typographic Conventions 6

2 How to Build SCRT 9
2.1 Overview 9
2.2 Building SCRT 9
2.2.1 Source Code Overview 9
2.2.2 Build Source Code 10

3 How to Use SCRT 15
3.1 Overview 15
3.2 System Configuration 15
3.3 Run SCRT Utility 15
3.4 Analyze SCRT Test Result 18
3.4.1 COM1/COM2 output 18
3.4.2 Port 80 Display 21
3.4.3 Each Assertion Information 22
3.4.4 System Hang 25

4 How to Add SCRT Test Cases 27
4.1 Overview 27
4.2 Add Test Cases 27
4.3 Example: Adding a Test Case 28

Figures

Figure 1 Select Boot Manager 16
Figure 2 Select Internal EFI Shell 16
Figure 3 Switch to the device 17
Figure 4 Run SCRT Utility 18
Figure 5 Terminal Software Configuration 19
Figure 6 Capture log files 20

Tables

Table 1 Build Tips in UEFI SCT Source Tree 10
Table 2 Port 80 display and Log file Relationship for Each assertion 22

Revision History

Revision Number	Description	Revision Date
1.0	Initial release.	August 2007

§

1 Introduction

1.1 Overview

The document is intended for Self-Certification Runtime Test (SCRT) utility users. SCRT is a toolset for platform firmware developers to validate UEFI Runtime Services implementations on IA-32, x64, and Itanium Architecture-based platforms for compliance to the UEFI 2.0 Specification. UEFI 2.0 Runtime services need to convert their related pointers when exiting boot service environment. Through this, these services are accessible in Runtime environment, typically virtual addressing mode.

Because the UEFI Self-Certification Test (SCT) Utility provides overall validation for the UEFI implementation, including runtime services in the boot time phase, SCRT acts as a supplement to SCT, and is intended for validating runtime services in a runtime environment. SCT covers more function testing, but SCRT focuses on simulating a runtime environment with virtual addressing mode and checking runtime services pointers convert issues.

Therefore, the document has three main objectives:

- Describe how to obtain the resources to build SCRT utility
- Describe how to use the SCRT utility and how to analyze the test results
- Describe how to customize the Runtime Test Cases.

Following are the chapter classifications:

- Chapter 1: Introduction
- Chapter 2: How to build SCRT
- Chapter 3: How to use SCRT
- Chapter 4: How to add SCRT Test Cases

1.2 Related Information

The following publications and sources of information may be useful or are referred to by this specification:

- *Extensible Firmware Interface Specification*, Version 1.10, Intel, 2001, http://developer.intel.com/technology/efi.
- *Unified Extensible Firmware Interface Specification*, Version 2.0, Unified EFI, Inc, 2006, http://www.uefi.org.
- *UEFI 2.0 SCT Document, Test Source, and Binary Code,* http:// www.uefi.org/specs/download/

- *Unified Extensible Firmware Interface Specification*, Version 2.1, Unified EFI, Inc, 2007, http://www.uefi.org.
- *Platform Initialization Specification,* Version 1.0, Unified EFI, Inc, 2006, http://www.uefi.org.
- *Intel® Platform Innovation Framework for EFI Specifications*, Intel, 2006, http://www.intel.com/technology/framework/.

1.3 Terms

The following terms are used throughout this document to describe varying aspects of input localization:

BDS

Framework Boot Device Selection phase.

BNF

BNF is an acronym for "Backus Naur Form." John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language.

Component

An executable image. Components defined in this specification support one of the defined module types.

DXE

Framework Driver Execution Environment phase.

DXE SAL

A special class of DXE module that produces SAL Runtime Services. DXE SAL modules differ from DXE Runtime modules in that the DXE Runtime modules support Virtual mode OS calls at OS runtime and DXE SAL modules support intermixing Virtual or Physical mode OS calls.

DXE SMM

A special class of DXE module that is loaded into the System Management Mode memory.

DXE Runtime

Special class of DXE module that provides Runtime Services

EFI

Generic term that refers to one of the versions of the EFI specification: EFI 1.02, EFI 1.10, or UEFI 2.0.

EFI 1.10 Specification

Intel Corporation published the Extensible Firmware Interface Specification. Intel donated the EFI specification to the Unified EFI Forum, and the UEFI now owns future updates of the EFI specification. See UEFI Specifications.

Foundation

The set of code and interfaces that glue implementations of EFI together.

Framework

Intel® Platform Innovation Framework for EFI consists of the Foundation, plus other modular components that characterize the portability surface for modular components designed to work on any implementation of the Tiano architecture.

GUID

Globally Unique Identifier. A 128-bit value used to name entities uniquely. An individual without the help of a centralized authority can generate a unique GUID. This allows the generation of names that will never conflict, even among multiple, unrelated parties.

HII

Human Interface Infrastructure. This generally refers to the database that contains string, font, and IFR information along with other pieces that use one of the database components.

IFR

Internal Forms Representation. This is the binary encoding that is used for the representation of user interface pages.

Library Class

A library class defines the API or interface set for a library. The consumer of the library is coded to the library class definition. Library classes are defined via a library class .h file that is published by a package. See the *EDK 2.0 Module Development Environment Library Specification* for a list of libraries defined in this package.

Library Instance

A an implementation of one or more library classes. See the *EDK 2.0 Module Development Environment Library Specification* for a list of library defined in this package.

Module

A module is either an executable image or a library instance. For a list of module types supported by this package, see module type.

Module Type

All libraries and components belong to one of the following module types: BASE, SEC, PEI_CORE, PEIM, DXE_CORE, DXE_DRIVER, DXE_RUNTIME_DRIVER, DXE_SMM_DRIVER, DXE_SAL_DRIVER, UEFI_DRIVER, or UEFI_APPLICATION. These definitions provide a framework that is consistent with a similar set of requirements. A module that is of module type BASE, depends only on headers and libraries provided in the MDE, while a module that is of module type DXE_DRIVER depends on common DXE components. For a definition of the various module types, see module type.

Module Surface Area (MSA)

The MSA is an XML description of how the module is coded. The MSA contains information about the different construction options for the module. After the module is constructed the MSA can describe the interoperability requirements of a module.

Package

A package is a container. It can hold a collection of files for any given set of modules. Packages may be described as one of the following types of modules:

- source modules, containing all source files and descriptions of a module
- binary modules, containing EFI Sections or a Framework File System and a description file specific to linking and binary editing of features and attributes specified in a Platform Configuration Database (PCD,)
- mixed modules, with both binary and source modules

Multiple modules can be combined into a package, and multiple packages can be combined into a single package.

Protocol

An API named by a GUID as defined by the EFI specification.

PCD

Platform Configuration Database.

PEI

Pre-EFI Initialization Phase.

PPI

A PEIM-to-PEIM Interface that is named by a GUID as defined by the PEI CIS.

SAL

System Abstraction Layer. A firmware interface specification used on Intel® Itanium® Processor based systems.

Runtime Services

Interfaces that provide access to underlying platform-specific hardware that might be useful during OS runtime, such as time and date services. These services become active during the boot process but also persist after the OS loader terminates boot services.

SEC

Security Phase is the code in the Framework that contains the processor reset vector and launches PEI. This phase is separate from PEI because some security schemes require ownership of the reset vector.

UEFI Application

An application that follows the UEFI specification. The only difference between a UEFI application and a UEFI driver is that an application is unloaded from memory when it exits regardless of return status, while a driver that returns a successful return status is not unloaded when its entry point exits.

UEFI Driver

A driver that follows the UEFI specification.

UEFI Specification Version 2.0

First version of the EFI specification released by the Unified EFI Forum. This specification builds on the EFI 1.10 specification and transfers ownership of the EFI specification from Intel to a non-profit, industry trade organization.

Unified EFI Forum

A non-profit collaborative trade organization formed to promote and manage the UEFI standard. For more information, see www.uefi.org.

1.4 Conventions Used in this document

This document uses the typographic and illustrative conventions described below.

1.4.1 Data Structure Descriptions

Intel® processors based on 32 bit Intel® architecture (IA 32) are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Intel® Itanium® processor family may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked reserved. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

1.4.2 Pseudo-code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a list is an unordered collection of homogeneous objects. A queue is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the Extensible Firmware Interface Specification.

1.4.3 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text (Body)	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text [blue] (Cross-Reference)	Any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. USE ONLY IF YOU MAKE AN ACTUAL CROSS-REFERENCE LINK.
Bold **(Bold or GlossTerm)**	In text, a Bold typeface identifies a processor register name. In other instances, a Bold typeface can be used as a running head within a paragraph. or as a definition heading (GlossTerm)
Italic	In text, an Italic typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace` **(CodeCharacter and CodePargraph)**	Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`Bold Monospace`	Words in a **`Bold Monospace`** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. USE ONLY IF YOU MAKE AN ACTUAL HYPERLINK.
`Italic Monospace` *`(ArgCharacter and ArgParagraph)`*	In code or in text, words in Italic Monospace indicate placeholder names for variable information that must be supplied (i.e., arguments).

Plain Monospace (CodeCharacter + Not Bold)	In code, words in a **Plain Monospace** typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

See the glossary sections in the UEFI 2.0 Specification for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the references sections in the UEFI 2.0 Specification for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

2
How to Build SCRT

2.1 Overview

This chapter introduces the Self-Certification Runtime Test (SCRT) Utility and focuses on how to build the SCRT source code. Since SCRT is a supplement to SCT, it could be built based on the SCT build infrastructure.

UEFI SCT is open source, located at http://www.uefi.org/specs/download/. Users may download the UEFI SCT release package from the website.

To set up theSCT build environment, refer to the instructions in the documents *UEFI SCT Getting Started* and *UEFI SCT User Guide*, which are included in the SCT release package.

2.2 Building SCRT

2.2.1 Source Code Overview

The SCRT source code package contains two modules: **SCRTApp** and **SCRTDriver**. **SCRTApp** is an application and **SCRTDriver** is a runtime driver.

These two modules both support IA32, x64 (EM64T) and IPF (Itanium) tip. The following are the source tree structures of **SCRTApp** and **SCRTDriver.**

```
SCRTApp\
    |----SCRTApp.c
    |----SCRApp.h
    |----SCRApp.inf
    |----ia32
       |---- GoVirtual.asm
       |---- VirtualMemory.c
    |----x64
       |---- GoVirtual.asm
       |---- VirtualMemory.c
    |----ipf
       |---- GoVirtual.S
       |---- VirtualMemory.c

SCRTDriver\
    |----Guid.h
    |----Guid.c
    |----Print.c
    |----TestCase.c
    |----SCRTDriver.c
    |----SCRTDriver.h
    |----SCRTDriver.inf
    |----ia32
       |---- Port80.asm
    |----x64
       |---- Port80.asm
    |----ipf
       |---- Port80.c
```

2.2.2 Build Source Code

The SCRT utility needs to build based on the SCT build infrastructure.

Firstly, extract the SCT source code package to the location **C:\Test\UefiSct.**

The UEFI SCT supports three build tips: IA32, x64 (EM64T) and IPF (Itanium). Each is given a hosting directory within the build tree. Table 1 lists these build tips.

Table 1 Build Tips in UEFI SCT Source Tree

Build Tip	Description
C:\Test\UefiSct\Platform\IntelTest\UEFI\IA32	The UEFI build environment for IA32 Architecture-based platforms.
C:\Test\UefiSct\Platform\IntelTest\UEFI\x64	The UEFI build environment for EM64T-based platforms.
C:\Test\UefiSct\Platform\ IntelTest\UEFI\IPF	The UEFI build environment for Itanium Architecture-based platforms.

Secondly, extract the SCRT source code package to the SCT source code tree **C:\Test\UefiSct\Platform\IntelTest\SCRT.**

This way, the SCRT utility in the SCT source code tree structure appears as follows:

```
C:\Test\UefiSct\Platform\IntelTest\SCRT\
                                        |-----SCRTApp
                                        |   |--ia32
                                        |   |--x64
                                        |   |--ipf
                                        |   |---…..
                                        |-----SCRTDriver
                                        |   |--ia32
                                        |   |--x64
                                        |   |--ipf
                                        |   |---…..
```

2.2.2.1 IA32 Build Tip

Follow the steps below to build the SCT IA32 tip to include the SCRT utility:

1. Add SCRT Utility **SCRTApp.inf** and **SCRTDriver.inf** files into **C:\Test\UefiSct\Platform\IntelTest\UEFI\IA32\Build\UEFI_SCT_IA32.dsc** file **[Components]** field.

```
[Components]
……..
#
# Components
#


#
# SCRT Utility
#
Platform\IntelTest\SCRT\SCRTApp\SCRTApp.inf
Platform\IntelTest\SCRT\SCRTDriver\SCRTDriver.inf
```

2. Run Visual Studio .NET 2003 Command Prompt to go to the command line environment. Then run the commands as given below:

   a. **cd C:\Test\UefiSct\Platform\IntelTest\UEFI\IA32**

   b. **set efi_source = C:\Test\UefiSct**

   c. **nmake uefi**

If the build is successful, the image files *.efi will be created in the directory **C:\Test\UefiSct\Platform\IntelTest\UEFI\IA32\uefi\IA32**.

If successful, **SCRTApp.efi** and **SCRTDriver.efi** will be located there also.

2.2.2.2 x64 Build Tip

Follow the steps below to build SCT x64 tip including SCRT utility:

1. Add SCRT Utility **SCRTApp.inf** and **SCRTDriver.inf** files into **C:\Test\UefiSct\Platform\IntelTest\UEFI\X64\Build\UEFI_SCT_X64.dsc** file **[Components]** field.

```
[Components]
……..
#
# Components
#

#
# SCRT Utility
#
Platform\IntelTest\SCRT\SCRTApp\SCRTApp.inf
Platform\IntelTest\SCRT\SCRTDriver\SCRTDriver.inf
```

2. Run Visual Studio .NET 2003 Command Prompt to go to the command line environment. Then run the commands as shown below:

   a. **cd C:\Test\UefiSct\Platform\IntelTest\UEFI\X64**

   b. **set efi_source = C:\Test\UefiSct**

   c. **nmake uefi**

If the build is successful, the image files *.efi are created in the directory **C:\Test\UefiSct\Platform\IntelTest\UEFI\X64\uefi\X64**.

If successful, **SCRTApp.efi** and **SCRTDriver.efi** are located there also.

2.2.2.3 IPF Build Tip

Follow the steps below to build the IPF tip to include the SCRT utility:

1. Add SCRT Utility **SCRTApp.inf** and **SCRTDriver.inf** files into **C:\Test\UefiSct\Platform\IntelTest\UEFI\IPF\Build\UEFI_SCT_IPF.dsc** file **[Components]** field.

```
[Components]
……..
#
# Components
#

#
# SCRT Utility
#
Platform\IntelTest\SCRT\SCRTApp\SCRTApp.inf
Platform\IntelTest\SCRT\SCRTDriver\SCRTDriver.inf
```

2. Run Visual Studio .NET 2003 Command Prompt to go to the command line environment. Then run the commands as shown below:

   a. **cd C:\Test\UefiSct\Platform\IntelTest\UEFI\IPF**

   b. **set efi_source = C:\Test\UefiSct**

   c. **nmake uefi**

If the build is successful, the image files *.efi are created in **C:\Test\UefiSct\Platform\IntelTest\UEFI\IPF\uefi\IPF** directory.

If successful, **`SCRTApp.efi`** and **`SCRTDriver.efi`** are located there also.

3 How to Use SCRT

3.1 Overview

As a supplement to SCT, SCRT is used to validate UEFI Runtime Services implementations for compliance to the *UEFI 2.0 Specification*. SCRT is invoked under the EFI shell environment. This chapter describes how to use SCRT utility in the EFI shell environment and how to analyze the test results.

3.2 System Configuration

To ensure SCRT runs in the runtime environment without unexpected behavior, for targeted platforms the physical memory on the target machine is limited to the following rules:

- IA32 architecture-based platform: Physical memory <= 4G.
  - The physical memory plugged on board is recommend that less or equal than 4 G Bytes.
- x64 architecture-based platform: Physical memory <= 32G.
  - The physical memory plugged on board is recommend that less or equal than 32 G Bytes.
- Itanium architecture-based platform: Physical memory <= 1024G.
  - The physical memory plugged on board is recommend that less or equal than 1024 G Bytes.

3.3 Run SCRT Utility

Before using SCRT utility, perform the following:

First, burn the UEFI implementation BIOS image into the machine targeted for test.

Then, obtain the corresponding version, IA32, X64, IPF, of **`SCRTApp.efi`** and **`SCRTDriver.efi`** files from the SCT build tip mentioned in Section 2.2.2

Finally, copy SCRT utility, **`SCRTApp.efi`** and **`SCRTDriver.efi`**, to the device used to perform SCRT on the target machine.

The following screenshots show the steps to run the SCRT utility in a shell environment:

1. When the target machine starts, select the menu Boot Manager as shown in Figure 1.

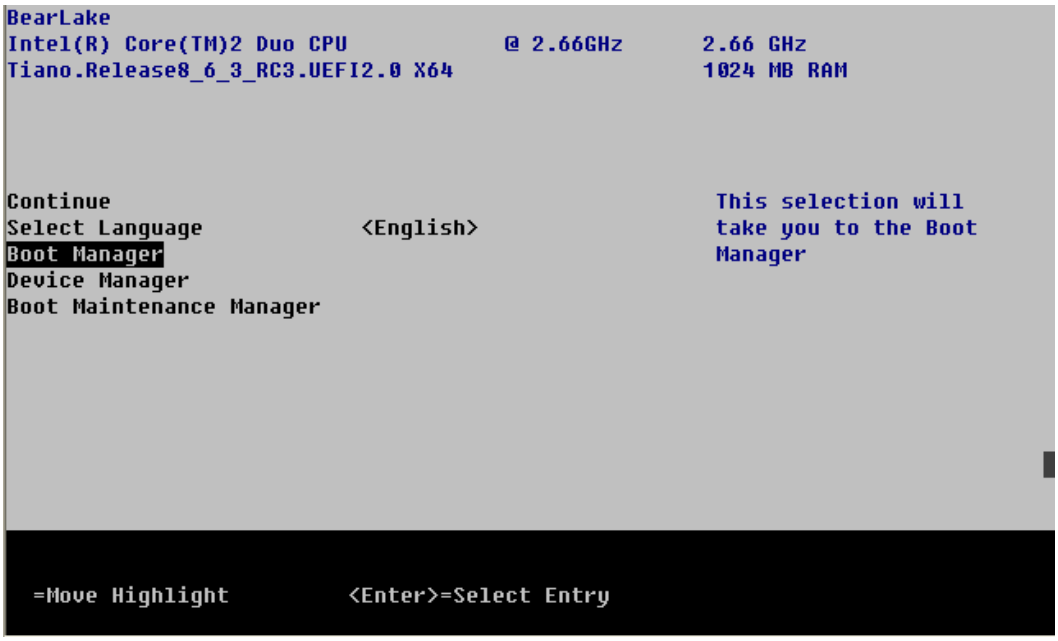

Figure 1 Select Boot Manager

2. Select the menu “Internal EFI Shell” in the Boot Manager as shown in Figure 2.

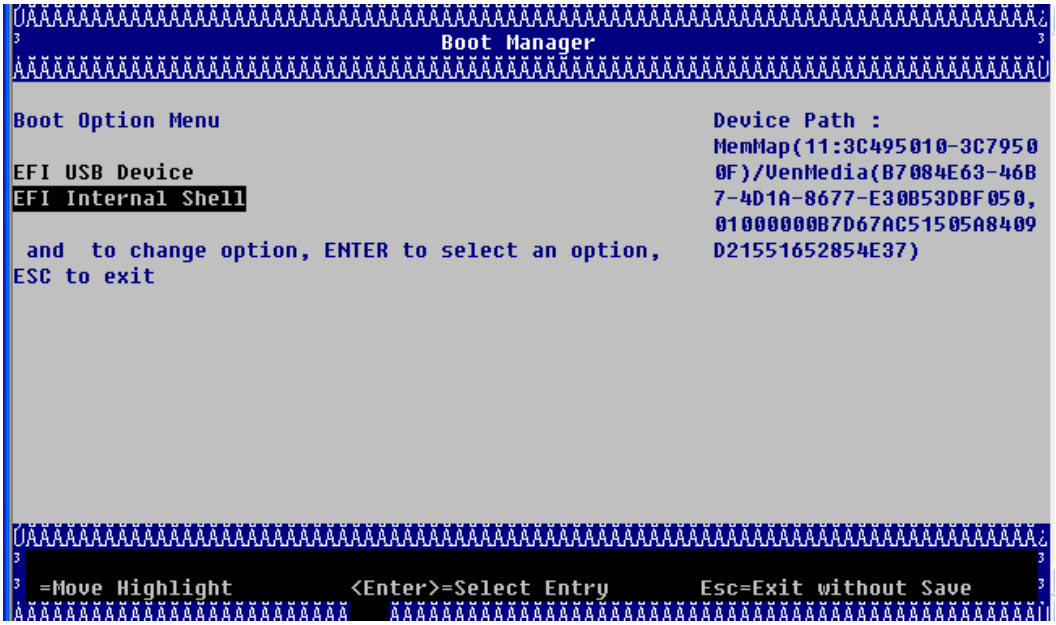

Figure 2 Select Internal EFI Shell

3. At the shell command line enter the command '**map -r**' to list your device name. Then switch to the device where SCRT is located, as shown in Figure 3.

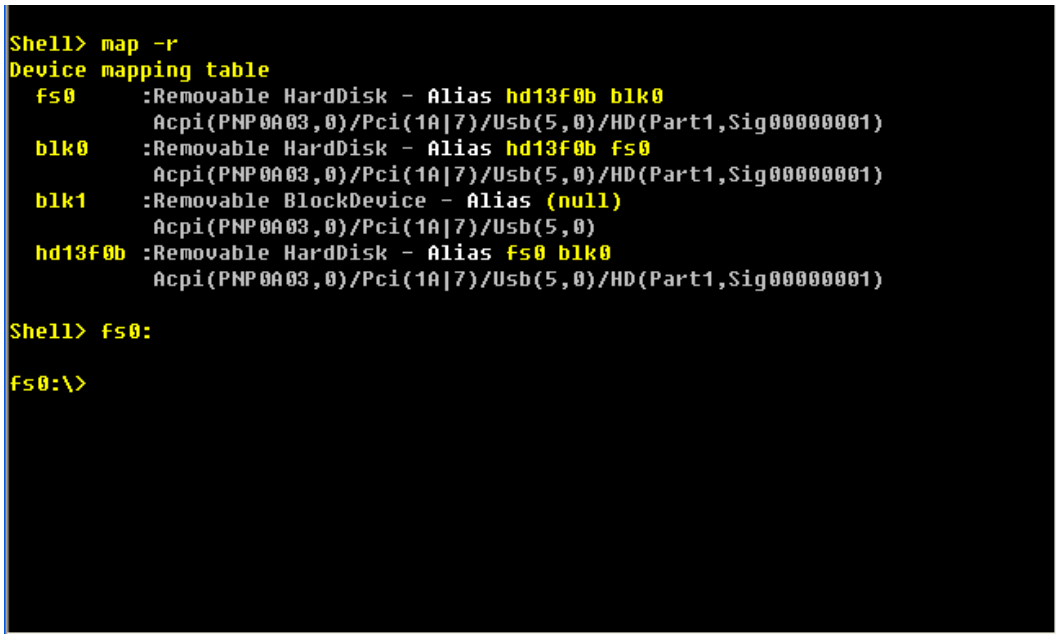

Figure 3 Switch to the device

4. Run the SCRT utility. First run the command **Load SCRTDriver.efi**, then run the command **SCRTApp.efi** as shown in Figure 4. To this point SCRT has begun to test UEFI 2.0 Runtime Services in the runtime environment.

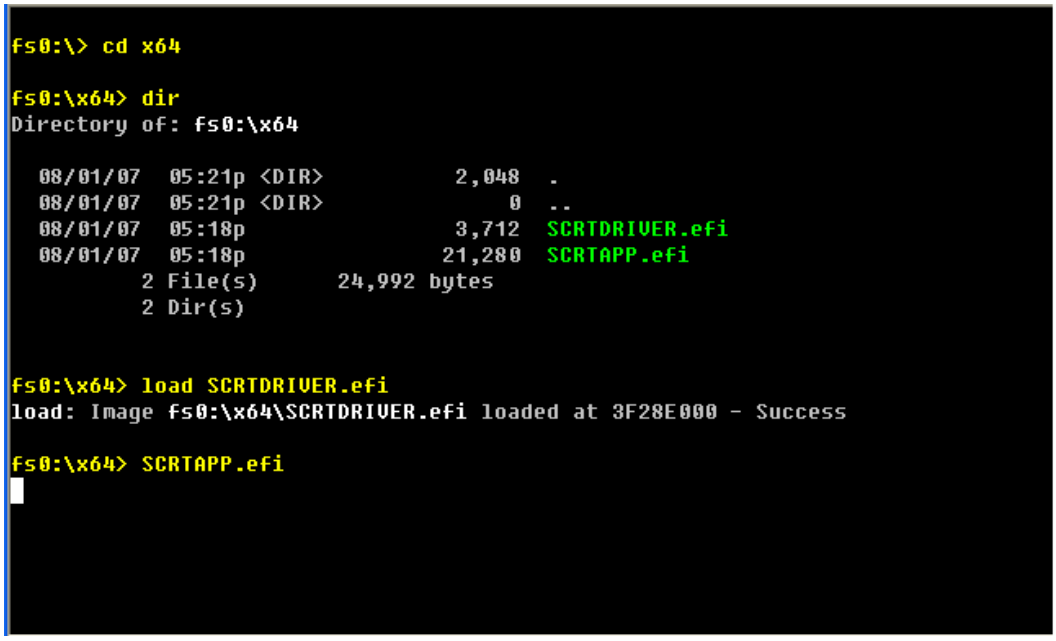

Figure 4 Run SCRT Utility

3.4 Analyze SCRT Test Result

Unlike SCT, SCRT cannot create a test log file in a runtime environment because it lacks some boot services. But SCRT can print out a similar format test log to COM1 and COM2. And at the same time, SCRT can send debug messages to Port 80. Using these messages the user can easily pinpoint test error locations.

3.4.1 COM1/COM2 output

If the target machine tested has either COM1 or COM2 on board, connect them to the host machine with the proper cable. With the help of terminal software such as **HyperTerminal**, take over COM1/COM2 output.

3.4.1.1 Configure Terminal Software

Figure 5 shows how to configure the terminal software. This example assumes use of COM1. After connecting, the terminal displays the COM1 output correctly.

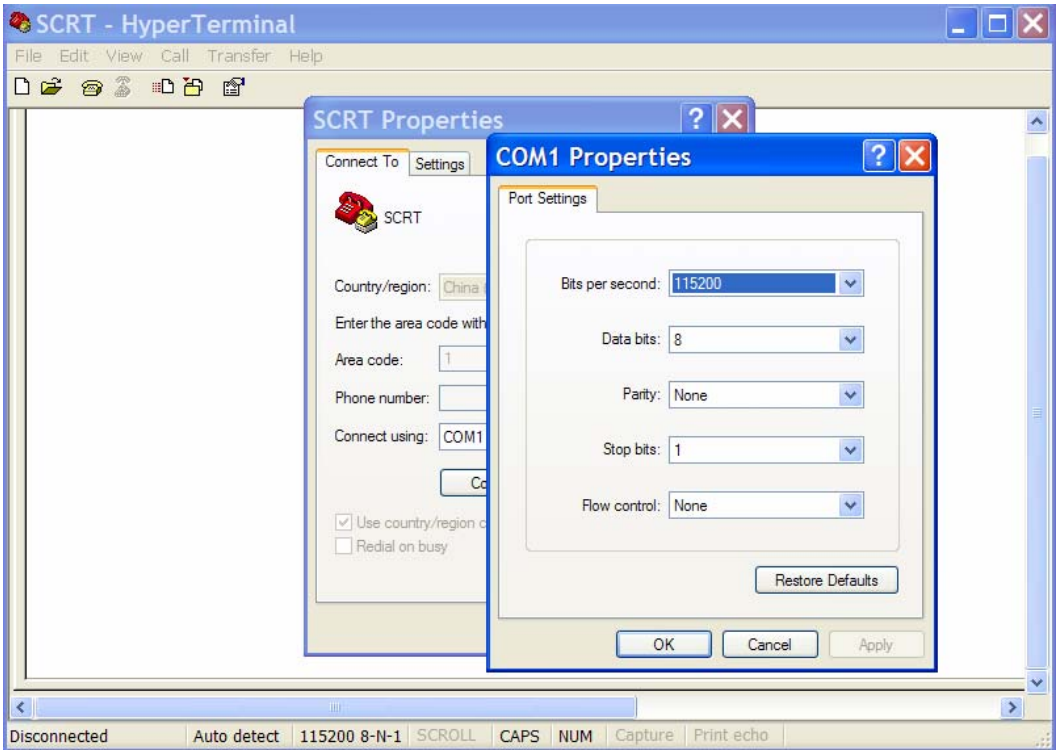

Figure 5 Terminal Software Configuration

3.4.1.2 Capture Log Files

Terminal software can capture SCRT test log files as well. Figure 6 shows how to capture log files.

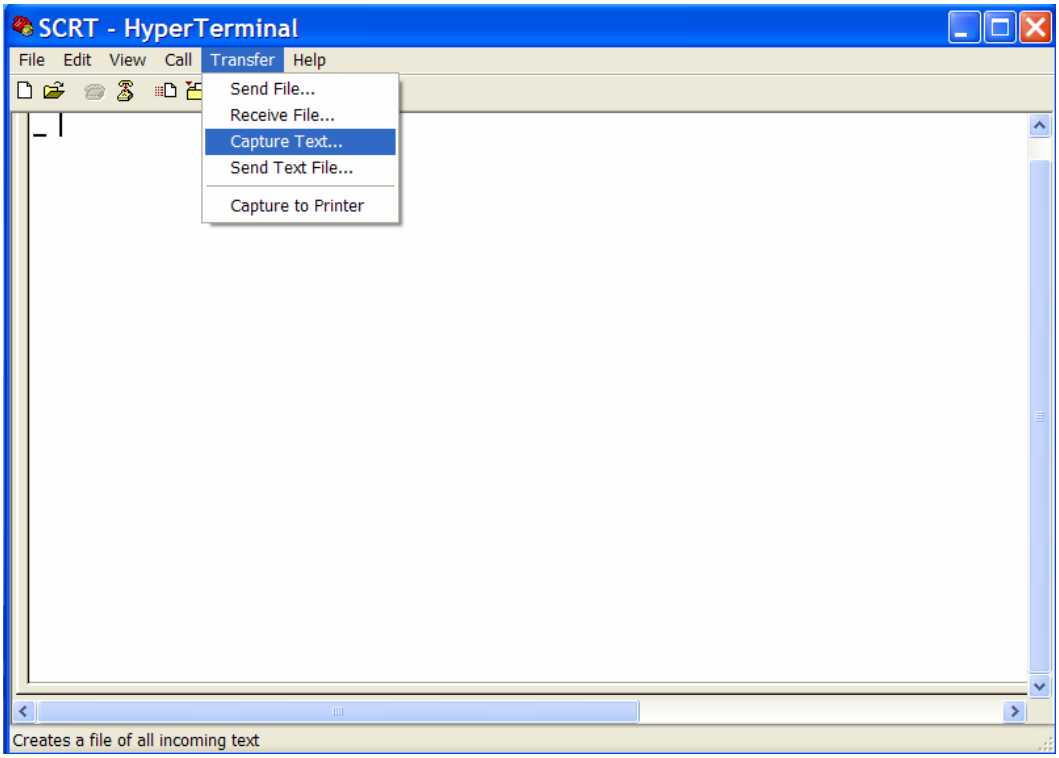

Figure 6 Capture log files

3.4.1.3 Log File Overview

The SCRT log format is similar to SCT log files. SCRT log file is divided into several groups:

```
Variable Services Test
Time Services Test
Capsule Service Test
Misc Services Test
Reset Services Test
```

Each group has the identical log format, and it is similar to a SCT log. The following is an example of the captured log file of **`Variable Services Test`**.

```
==================Variable Services Test Start==================

RT.SetVariable - Set a test variable named UEFIRuntimeVariable, should be
EFI_SUCCESS - PASS
BFF7E548-F13A-497C-8E21-AEC237A6CCE3
P:\RC3\TIANO\Platform\SCRTDriver\TestCase.c:73:Status - Success, Expected
- Success

RT.GetVariable - Get the test variable named UEFIRuntimeVariable, should
be EFI_SUCCESS - PASS
F556B5AD-AACE-4BF0-B724-E129EE00EA37
P:\RC3\TIANO\Platform\SCRTDriver\TestCase.c:94:Status - Success, Expected
- Success

RT.GetNextVariableName - The test variable named UEFIRuntimeVariableTest
should be found - PASS
BAC20972-9662-4F24-8AAC-664142B56DDE
P:\RC3\TIANO\Platform\SCRTDriver\TestCase.c:145

RT.QueryVariableInfo - Query Variable Information of the platform should
be EFI_SUCCESS - PASS
8BCDA7A3-2848-413D-BF05-07E1098D42D2
P:\RC3\TIANO\Platform\SCRTDriver\TestCase.c:167:Status - Success,
Expected - Success

==================Variable Services Test End====================
```

Similar to the SCT log file, each checkpoint is identified by a GUID. From this GUID, users can easily locate the corresponding checkpoint. The description for the checkpoint is printed out, as well as the general **PASS** or **FAILURE** test status. An example follows:

```
RT.SetVariable - Set a test variable named UEFIRuntimeVariable, should be
EFI_SUCCESS - PASS
BFF7E548-F13A-497C-8E21-AEC237A6CCE3
```

Additionally, more details about checkpoints are listed, including file name, line number, the returned status and the expected status.

```
P:\RC3\TIANO\Platform\SCRTDriver\TestCase.c:73:Status - Success, Expected
- Success
```

With this format log file, the user can easily find which checkpoint fails and locate it to the source code.

3.4.2 Port 80 Display

If the target machine under test has neither COM1 nor COM2, use Port 80 to trace the test case work flow. For every checkpoint, Port 80 will display a unique hex number. The number from **00, 01 … 0A, 0B …** to **11, 12** is already defined in current SCRT test cases.

3.4.3 Each Assertion Information

For each assertion, Port 80 displays a unique hex number, while also printing a GUID and description to COM1/COM2. The relationship is shown in Table 2.

Table 2 shows the detailed information for each assertion in the UEFI SCRT tests. It can be used by UEFI SCRT users as a case assertion reference.

Table 2 Port 80 display and Log file Relationship for Each assertion

Port 80 Display	GUID	Assertion	Test Description
01	0xbff7e548, 0xf13a, 0x497c, 0x8e, 0x21, 0xae, 0xc2, 0x37, 0xa6, 0xcc, 0xe3	**RT.SetVariable** - Set a test variable named *UEFIRuntimeVariable*, should be **EFI_SUCCESS**	1. Call **RT.SetVariable** with the special name and Guid. And the variable is set with 8 Bytes data size. The return status should be **EFI_SUCCESS**.
02	0xf556b5ad, 0xaace, 0x4bf0, 0xb7, 0x24, 0xe1, 0x29, 0xee, 0x0, 0xea, 0x37	**RT.GetVariable** - Get the test variable named *UEFIRuntimeVariable*, should be **EFI_SUCCESS**	Call **RT.GetVariable** to get the test variable just set. The return status should be **EFI_SUCCESS**
03	0xd66e4a7f, 0x6d54, 0x4cc0, 0xb9, 0x3b, 0xf6, 0x2f, 0x48, 0x57, 0xa6, 0xff	**RT.GetVariable** - The test variable named *UEFIRuntimeVariable* should have 8 Bytes data size.	The test variable get should have 8 Bytes data size.
04	0xaa5c5763, 0x36cd, 0x4f00, 0x84, 0x36, 0xf4, 0xa9, 0xd5, 0xaf, 0x12, 0xfb	**RT.GetNextVariableName** - Get the next variable name should be **EFI_SUCCESS**	Loop to call **RT.GetNextVariableName** to get the next variable. The return status should be **EFI_SUCCESS**.

Port 80 Display	GUID	Assertion	Test Description
05	0xbac20972, 0x9662, 0x4f24, 0x8a, 0xac, 0x66, 0x41, 0x42, 0xb5, 0x6d, 0xde	**RT.GetNextVariableName** - The test variable named *UEFIRuntimeVariableTest* should be found	Find the test variable in the loop call **RT.GetNextVariableName**.
06	0x8bcda7a3, 0x2848, 0x413d, 0xbf, 0x5, 0x7, 0xe1, 0x9, 0x8d, 0x42, 0xd2	**RT.QueryVariableInfo** - Query Variable Information of the platform should be **EFI_SUCCESS.**	Call **RT.QueryVariableInfo** to query variable information. The return status should be **EFI_SUCCESS**.
07	0x67b4e72a, 0xc792, 0x4f74, 0x92, 0x1d, 0xea, 0xb3, 0x66, 0x4f, 0x95, 0x3b	**RT.GetTime** - Get the current time and date information should be **EFI_SUCCESS**	Call **RT.GetTime** with **NULL** capabilities. The return status should be **EFI_SUCCESS**.
08	0xdbb5195f, 0x3584, 0x427d, 0xa1, 0x68, 0x3f, 0x5e, 0x1d, 0x24, 0x3b, 0xb9	**RT.SetTime** - Change *Time.Year* to 2060 to set the time should be **EFI_SUCCESS**	Change the current data to year=2060. And set time as it. The return status should be **EFI_SUCCESS**.
09	0x8e75d9a9, 0x3c14, 0x4095, 0xbe, 0x76, 0xad, 0xcf, 0x55, 0xab, 0x8e, 0x6c	**RT.GetTime** - Get the current time to check whether the modification happens, should be **EFI_SUCCESS**	Call **RT.GetTime** to get the current time. The return status should be **EFI_SUCCESS**.
0A	0xe8cd357a, 0xd254, 0x4f7b, 0x92, 0xc3, 0x23, 0xfd, 0x4d, 0xd6, 0xc0, 0xa3	**RT.GetTime** - The current time should be changed to *Time.Year* = 2060	The get time should be Year = 2060.

Port 80 Display	GUID	Assertion	Test Description
0B	0x6417f479, 0xa174, 0x4614, 0x80, 0xcd, 0xe6, 0x96, 0x85, 0x8c, 0xd9, 0xfa	**RT.GetTime** - Get the current time and date information to modify, should be **EFI_SUCCESS**	Call **RT.GetTime** again to modify the time to set wakeup time. The return status should be **EFI_SUCCESS**.
0C	0xd6a3c41a, 0xe6cf, 0x42fc, 0xa0, 0x39, 0x68, 0xf8, 0x39, 0xbb, 0xbf, 0xe3	**RT.SetWakeupTime** - Set wakeup time in 1 hour later from now on, should be **EFI_SUCCESS**	Call **RT.SetWakeupTime** to set wake up time, the time is 1 hour later from now on. The return status should be **EFI_SUCCESS**.
0D	0xd6b952a9, 0x3d54, 0x4277, 0xbf, 0x60, 0xab, 0xfb, 0x3, 0x71, 0x5, 0xd5	**RT.GetWakeupTime** - Get the current wakeup alarm clock setting information, should be **EFI_SUCCESS.**	Call **RT.GetWakeupTime** to get the current wake up time. The return status should be **EFI_SUCCESS**.
0E	0x3f65c680, 0xae51, 0x4830, 0xb3, 0xd1, 0xd7, 0xc9, 0x2a, 0xcd, 0x14, 0x8a	**RT.QueryCapsuleCapabilities** - Query the capsule capabilities the platform supports, should be **EFI_SUCCESS.**	Call **RT.QueryCapsuleCapabilities** to query the capsule capabilities. The return status should be **EFI_SUCCESS**.
0F	0x4611524b, 0xbfd2, 0x42d4, 0x85, 0xa8, 0x9b, 0xf, 0xd1, 0xc6, 0x27, 0xd3	**RT.GetNextHighMonotonicCount** - First get next high monotonic counter, should be **EFI_SUCCESS.**	Call **RT.GetNextHighMonotonicCount** to get next high monotonic counter for the first time. The return status should be **EFI_SUCCESS**.
10	0x5c2cbd54, 0x1388, 0x4e87, 0xab, 0x11, 0x2c, 0x12, 0x3d, 0x24, 0x5, 0xbd	**RT.GetNextHighMonotonicCount** - Second get next high monotonic counter, should be **EFI_SUCCESS**	Call **RT.GetNextHighMonotonicCount** again to get the counter. The return status should be **EFI_SUCCESS**.

Port 80 Display	GUID	Assertion	Test Description
11	0x9e39a3e3, 0xcbb6, 0x4fcc, 0xb2, 0x21, 0x73, 0x24, 0x79, 0xf1, 0x21, 0x77	**RT.GetNextHighMonotonicCount** - Second get counter should increase 1 compared with First get counter	The second call get the bigger counter: The 2nd counter = the 1st counter + 1.
12	0xda790c1e, 0xdcbf, 0x4c0e, 0xaf, 0xf7, 0x46, 0x3a, 0xc4, 0x47, 0xb0, 0x6e	**RT.ResetSystem** - Machine should shut down! We should never come here	Call **RT.ResetSystem** to shut down. Should never come here.

3.4.4 System Hang

SCRT validates the Runtime Services implementation in the runtime environment. If some pointers are not converted, the system hangs. If the system hangs at any checkpoint, the SCRT records the last step information in the test log file and displays it in Port 80. With the relationship shown in Table 2, the user could find which checkpoint hangs easily.

4
How to Add SCRT Test Cases

4.1 Overview

SCRT is used to validate Runtime Services in al runtime environment. The functionalities of these runtime services are already covered during boot time, and users can refer to the test case in the UEFI SCT. Therefore, the SCRT focuses more on validating whether the runtime service pointer is correctly converted when switch to runtime environment.

When more detailed test cases for runtime services are needed, users could develop the required test case, and readily add it to the SCRT infrastructure.

This chapter describes how to add test cases when it is necessary to customize SCRT test cases.

4.2 Add Test Cases

SCRTDriver in the SCRT utility is responsible for performing the test cases. In **SCRTDriver** module, GUID definition for the checkpoints is declared in **Guid.h** and **Guid.c**, and test cases are located in **TestCase.c**.

To extend the test coverage, the user can add the test cases in **TestCase.c** and add the new GUID definitions **in Guid.h/Guid.c**.

```
SCRTDriver\
    |----Guid.h[1]
    |----Guid.c[1]
    |----TestCase.c[2]
    |----Print.c
    |----SCRTDriver.c
    |----SCRTDriver.h
    |----SCRTDriver.inf
    |----ia32
       |---- Port80.asm
    |----x64
       |---- Port80.asm
    |----ipf
       |---- Port80.c
```

Note: ***Guid.h/Guid.c*** *declares GUID definition.*

Note: ***TestCase.c*** *consists of the test cases.*

In **TestCase.c**, we allow for adding more checkpoints. For each new checkpoint, the user needs create a new GUID for it and declare it in **Guid.h/Guid.c**.

4.3 Example: Adding a Test Case

Because the call Runtime Service **UpdateCapsule** behaves differently for different platforms, for example, a system reset, this checkpoint is not included in **TestCase.c** as a common test case. Users can add a case in **TestCase.c** to verify the service like below.

Here is a sample code to add the checkpoint in **EfiCapsuleTestVirtual(), TestCase.c:**

```
Status = VRT->UpdateCapsule (
                xxxxx,
                xxxxx,
                xxxxx
                 );

Port80(xxx);

RecordAssertion (
    Status,
    gSCRTAssertionGuidxxx,
    "RT. UpdateCapsule - should be EFI_SUCCESS",
    "%a:%d:Status - %r, Expected - %r",
    __FILE__,
    __LINE__,
    Status,
    EFI_SUCCESS
    );
```

Meanwhile, define **gSCRTAssertionGuidxxx** in **Guide.h** and **Guide.c** as shown below:

In **Guide.c**:

```
EFI_GUID gSCRTAssertionGuidxxx = EFI_TEST_SCRT_ASSERTION_xxx_GUID;
```

In **Guide.h**:

```
#define EFI_TEST_SCRT_ASSERTION_xxx_GUID \
{ xxxxxxxx, xxxx, xxxx, { xx, xx, xx, xx, xx, xx, xx, xx } }

extern EFI_GUID gSCRTAssertionGuidxxx;
```

Filename: SCRT User Guide_Myron_Aug21.doc
Directory: M:\UEFI\WorkingFolder\UTWG
Template: C:\Documents and Settings\tding1\Application Data\Microsoft\Templates\MasterEFI14.dot
Title:
Subject:
Author: mporter
Keywords:
Comments:
Creation Date: 8/21/2007 9:52:00 AM
Change Number: 9
Last Saved On: 8/21/2007 10:12:00 AM
Last Saved By: mporter
Total Editing Time: 13 Minutes
Last Printed On: 8/21/2007 10:12:00 AM
As of Last Complete Printing
Number of Pages: 32
Number of Words: 5,554 (approx.)
Number of Characters: 31,439 (approx.)