EFI Developer's Guide

Draft for Review

Version 1.10

January 5, 2004

Revision History

Revision	Revision History	Date
0.1	Initial review draft	5/3/99
0.2	Updated for the 0.91 Reference Implementation	7/14/99
0.9	Updated for the 0.91.009 Reference Implementation	11/18/99
1.10	Updated for the 1.10.14.62 release of the EFI Application Toolkit.	1/5/04

Contents

Tables

1 Introduction

1.1 Overview

The *Extensible Firmware Interface Specification* (hereafter referred to as the *"EFI Specification"* or *"EFI 1.10 Specification"*) describes a set of application programming interfaces (APIs) and data structures that are exported by a system's firmware. Software that takes advantage of these APIs and data structures may take one of many forms, including any of the following:

- EFI device drivers
- EFI Shell
- EFI system utilities
- EFI system diagnostics
- Operating system (OS) loaders

In addition, the *EFI Specification* describes a set of runtime services that are available to an OS while the OS has full control of the system.

The EFI Sample Implementation is a reference implementation of the *EFI Specification* and includes an EFI Shell. An EFI Shell is a special type of EFI application that allows other EFI applications to be launched. The combination of the EFI firmware and the EFI Shell provide an environment that can be modified to easily adapt to many different hardware configurations. The EFI Shell is a simple, interactive environment that allows EFI device drivers to be loaded, EFI applications to be launched, and operating systems to be booted. In addition, the Shell also provides a set of basic commands that are used to manage files and the system environment variables.

An EFI application has access to the APIs and data structures that are exported by the EFI firmware. An EFI application may be any number of things, including a system utility, a system diagnostic, or an OS loader. Many of the EFI Shell commands are implemented as EFI applications. An OS loader is a special type of EFI application because it typically will load a number of other files related to the OS into memory, and then transfer control an OS kernel. During this process, the OS loader may collect the system's configuration information that is maintained by the EFI firmware and put it into a form that can be used by the OS kernel.

An EFI device driver provides a set of interfaces that allow communication with a hardware device. The types of devices that are typically available in the EFI environment include keyboard, display, hard drive, network, CD-ROM, floppy, and others. The *EFI Specificatio*n provides mechanisms to add new types of device drivers to the base firmware so new types of devices can be accessed from the EFI environment and presented to the OS loader during the boot process. These device drivers can be loaded from the system's nonvolatile storage, a PCI option ROM, or any system device that is natively supported by the EFI environment. Device drivers are very similar to EFI applications. They have the same access to all the APIs and data structures. The main difference is that they typically add a set of interfaces that provide access to a hardware device. When the device driver exits, the EFI device driver stays resident in memory, so those interfaces may be used by other types of EFI applications to access the hardware device.

The EFI Sample Implementation of the EFI firmware and the EFI Shell includes a set of library functions and macros. These functions and macros represent one possible minimal implementation of the EFI environment. Together, the EFI APIs and EFI Library functions provide the functionality that is required for basic console I/O, basic disk I/O, memory management, and string manipulation. The EFI Library also provides functions for 64-bit math operations, Cyclic Redundancy Checks (CRCs), and spin locks, as well as helper functions to manage device handles, device protocols, and device paths. The appendices in this document contain a complete list of the APIs, library functions, and library macros that are available to an EFI developer. See the appropriate specification for complete details on all the APIs and macros.

1.2 Organization of this Document

This specification is organized as listed in Table 1-1.

Table 1-1. Organization and Contents of This Document

Chapter	Description
Chapter 1: Introduction	Introduction to EFI development.
Chapter 2 : EFI Applications	Describes how to write, compile, and run EFI applications.
Chapter 3: EFI OS Loaders	Describes how to write an EFI OS loader.
Appendix A: EFI Boot Services	Lists and describes at a high level the EFI Boot Services.
Appendix B: EFI Runtime Services	Lists and describes at a high level the EFI Runtime Services.
Appendix C: EFI Protocol Interfaces	Lists and describes at a high level the EFI protocol interfaces.
Appendix D: EFI Library Overview	Lists and describes at a high level the EFI Library functions.
Appendix E: Sample OS Loader Listing	C source code to OS loader sample.
Appendix F: Sample OS Loader Output	Output from OS loader sample.
Apendix G: Glossary	Glossary of acronyms that are used in this document.

1.3 Goals

The primary goal of this document is to provide an overview of the EFI environment and its capabilities. This overview includes writing EFI applications and writing EFI device drivers.

1.4 Target Audience

This document is intended for the following readers:

- Developers who will be using and writing applications, including tests and diagnostics, OS loaders, and device drivers for the EFI environment.

1.5 Prerequisite Specifications

This document assumes you are familiar with the following prerequisite specifications:

- *Extensible Firmware Interface Specification*, version 1.10, Intel Corporation, 2002.
- *EFI 1.10 Specification Update*, version -001, Intel Corporation, 2003.
- *Extensible Firmware Interface Library Specification* (hereafter referred to as the *EFI Library Specification*), version 1.10.1, Intel Corporation, 2004.

1.6 Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

1.6.1 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*.

1.6.2 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Bold	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.
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`	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.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	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 in the *EFI 1.10 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 section in the *EFI 1.10 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
EFI Applications

This section describes how to write, compile, and run applications in the sample implementation of the EFI environment. Building applications in the sample implementation is described for illustration purposes.

2.1 Writing EFI Applications

EFI applications can be added to the EFI source tree. It is recommended that all standalone EFI applications be placed below the **`\efi\apps`** directory. This recommendation is not strictly required. However, it does provide a convenient build environment. To add a new application to the build environment, a new subdirectory needs to be created, and a makefile (**`hello.mak`**) along with the source code needs to be placed in that subdirectory. For this example, the files for a simple test application are placed in the directory **`hello`**, and their contents are listed below. The makefile **`hello.mak`** specifies the following:

- A list of source files
- The path to the include directories
- The path the library directories
- The entry point for the application
- The name of the executable EFI application image

The contents of **`hello.mak`** are listed below.

```
!include $(SDK_INSTALL_DIR)\build\$(SDK_BUILD_ENV)\sdk.env

BASE_NAME          = hello
IMAGE_ENTRY_POINT = InitializeHelloApplication

# Globals needed by master.mak

TARGET_APP = $(BASE_NAME)
SOURCE_DIR = $(SDK_INSTALL_DIR)\apps\$(BASE_NAME)
BUILD_DIR  = $(SDK_BUILD_DIR)\apps\$(BASE_NAME)

# Include paths

!include $(SDK_INSTALL_DIR)\include\$(EFI_INC_DIR)\makefile.hdr
INC = -I $(SDK_INSTALL_DIR)\include\$(EFI_INC_DIR) \
      -I $(SDK_INSTALL_DIR)\include\$(EFI_INC_DIR)\$(PROCESSOR) $(INC)

all : dirs $(LIBS) $(OBJECTS)

# Program object files

OBJECTS = $(OBJECTS) $(BUILD_DIR)\$(BASE_NAME).obj

# Source file dependencies
```

```
$(BUILD_DIR)\$(BASE_NAME).obj : $(*B).c $(INC_DEPS)

# Handoff to master.mak

!include $(SDK_INSTALL_DIR)\build\master.mak
```

The test application listed below is the simplest possible application that can be written. It does not depend upon any EFI Library functions, so the EFI Library is not linked into the executable that is generated. This test application uses the *`SystemTable`* that is passed into the entry point to get access to the EFI console devices. The console output device is used to display a message using the **`OutputString()`** function of the **`SIMPLE_TEXT_OUTPUT_INTERFACE`** protocol, and the application waits for a keystroke from the user on the console input device using the **`WaitForEvent()`** service with the *`WaitForKey`* event in the **`SIMPLE_INPUT_INTERFACE`** protocol. Once a key is pressed, the application exits.

```
/*++
Copyright (c) 1998  Intel Corporation
Module Name:
    hello.c
Abstract:
Author:
Revision History
--*/

#include "efi.h"

EFI_STATUS
InitializeHelloApplication (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
    )
{
    UINTN Index;

    //
    // Send a message to the ConsoleOut device.
    //

    SystemTable->ConOut->OutputString(SystemTable->ConOut,
                                      L"Hello application started\n\r");

    //
    // Wait for the user to press a key.
    //

    SystemTable->ConOut->OutputString(SystemTable->ConOut,
                                      L"\n\r\n\r\n\rHit any key to exit\n\r");

    SystemTable->BootServices->WaitForEvent (1,
                                             &(SystemTable->ConIn->WaitForKey),
                                             &Index);

    SystemTable->ConOut->OutputString(SystemTable->ConOut,L"\n\r\n\r");
    //
    // Exit the application.
    //

    return EFI_SUCCESS;
}
```

2.2 Compiling EFI Applications

EFI applications are PE/COFF images with a modified header signature. The modified header helps distinguish EFI images from PE/COFF images. See the *EFI Specification* for details on EFI images.

Before the EFI image type is integrated into the standard commercial tool chain, a utility for converting PE/COFF images to 64-bit EFI images is available. This utility is provided in the current releases of the EFI Sample Implementation.

Before a new EFI application can be compiled, the makefile for each build tip needs to be modified. These files include the following:

```
\efi\build\ia32-emb\makefile
\efi\build\sal64\makefile
\efi\build\bios32\makefile
```

In each of these files, there is a section labeled “Everything to build.” Add a line to the end of this section for each new EFI application. The section of a makefile that supports two test applications is listed below. Once these changes have been made, the new application will be compiled when **`nmake`** is run from a build tip.

```
#
# Everything to build
#

makemaker:
    $(BUILD_TOOL)\genmake
    $(MAKE) -f output\lib\makefile                             all
    $(MAKE) -f output\corefw\drivers\dskio\makefile            all
    $(MAKE) -f output\corefw\drivers\pblkio\makefile           all
    $(MAKE) -f output\corefw\drivers\ramdisk\makefile          all
    $(MAKE) -f output\corefw\drivers\fat\makefile              all
    $(MAKE) -f output\corefw\fw\efi\makefile                   all
    $(MAKE) -f output\corefw\fw\platform\nt\makefile           all
    $(MAKE) -f output\corefw\fw\platform\nt\ntemulc\makefile   all
    $(MAKE) -f output\corefw\fw\platform\drivers\defio\makefile all
    $(MAKE) -f output\corefw\fw\platform\BootMgr\makefile      all
    $(MAKE) -f output\corefw\fw\platform\Unicode\makefile      all
    $(MAKE) -f output\shell\makefile                           all
    $(MAKE) -f output\apps\hello\makefile                      all
    $(MAKE) -f output\apps\hellolib\makefile                   all
```

2.3 Writing EFI Applications with EFI Library Support

If an EFI application wants to use the EFI Library functions, then the file **efilib.h** needs to be included and a call to **InitializeLib()** needs to be added. The following is a listing of the EFI application **hellolib.c**. This application makes use of the EFI Library to print text to the console output device. It also uses the global variable *ST* instead of *SystemTable* to make the standard EFI API calls.

```
/*++
Copyright (c) 1999  Intel Corporation
Module Name: hellolib.c
Abstract: This is an example EFI program
Author:
Revision History: 1.0
--*/

#include "efi.h"
#include "efilib.h"

EFI_STATUS
InitializeHelloLibApplication (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
    )
{
    //
    // Initialize the Library. Set BS, RT, & ST globals
    //  BS = Boot Services
    //  RT = Runtime Services
    //  ST = System Table
    //

    InitializeLib (ImageHandle, SystemTable);

    //
    // Print a message to the console device using a library function.
    //

    Print(L"HelloLib application started\n");

    //
    // Wait for a key to be pressed on the console device.
    //

    Print(L"\n\n\nHit any key to exit this image\n");
    WaitForSingleEvent (ST->ConIn->WaitForKey, 0);

    //
    // Print a message to the console device using a protocol
    // interface.
    //

    ST->ConOut->OutputString (ST->ConOut, L"\n\n");
    //
    // Return control to the Shell.
    //

    return EFI_SUCCESS;
}
```

2.4 Running EFI Applications

An EFI application may be executed by typing the program's name at the EFI Shell command line. The following examples show how to run the test applications described in the sections above from the EFI Shell. Both applications wait for the user to press a key before they will return to the EFI Shell prompt. It is assumed that **`hello.efi`** and **`hellolib.efi`** are in the search path of the EFI Shell environment.

Examples

```
Shell> hello

Hello application started



Hit any key to exit this image


Shell> hellolib

HelloLib application started



Hit any key to exit this image


Shell>
```

3
EFI OS Loaders

This section discusses the special considerations that are required when writing an OS loader. An OS loader is a special type of EFI application. It is responsible for transitioning a system from a firmware environment into an OS environment. To accomplish this task, several important steps must be taken, as follows:

1. The OS loader must determine where it was loaded from. This determination will allow an OS loader to retrieve additional files from the same location.
2. The OS loader must determine where in the system the OS exists. Typically, the OS will reside on a partition of a hard drive. However, the partition where the OS exists may not use a file system that is recognized by the EFI environment. In this case, the OS loader can only access the partition as a block device using only block I/O operations. The OS loader will then be required to implement or load the file system driver to access files on the OS partition.
3. The OS loader must build a memory map of the physical memory resources so that the OS kernel can know what memory to manage. Some of the physical memory in the system must remain untouched by the OS kernel, so the OS loader must use the EFI APIs to retrieve the system’s current memory map.
4. An OS has the option of storing boot paths and boot options in nonvolatile storage in the form of environment variables. The OS loader may need to use some of the environment variables that are stored in nonvolatile storage. In addition, the OS loader may be required to pass some of the environment variables to the OS kernel.
5. The next step is to call **`ExitBootServices()`**. This call can be done from either the OS loader or from the OS kernel. Special care must be taken to guarantee that the most current memory map has been retrieved prior to making this call. Once **`ExitBootServices()`** had been called, no more EFI Boot Services calls can be made. At some point, either just prior to calling **`ExitBootServices()`** or just after, the OS loader will transfer control to the OS kernel.
6. Finally, after **`ExitBootServices()`** has been called, the EFI Boot Services calls are no longer available. This lack of availability means that once an OS kernel has taken control of the system, the OS kernel may only call EFI Runtime Services.

A complete listing of a sample application for an OS loader can be found in Appendix E, and the output from the OS loader sample can be found in Appendix F. The code fragments in the following sections do not perform any error checking. The code listing in Appendix E has similar code sections, but all the return status codes are checked. Also, the OS loader sample application makes use of several EFI Library functions to simplify the implementation.

The output shown in Appendix F shows the OS loader sample going through the steps listed above. It starts by printing out the device path and the file path of the OS loader itself. It also shows where in memory the OS loader resides and how many bytes it is using. Next, it loads the file **`OSKERNEL.BIN`** into memory. The file **`OSKERNEL.BIN`** was retrieved from the same directory as the image of the OS loader sample.

The next section of the output shows the first block of several block devices. The first one is the first block of the floppy drive with a FAT12 file system. The second one is the Master Boot Record (MBR) from the hard drive. The third one is the first block of a large FAT32 partition on the same hard drive, and the fourth one is the first block of a smaller FAT16 partition on the same hard drive.

The next section shows the pointers to all the system configuration tables, the system's current memory map, and a list of all the system's environment variables. The very last step shown is the OS loader calling **`ExitBootServices()`**.

3.1 Device Path and Image Information of the OS Loader

The following code fragment shows the steps that are required to get the device path and file path to the OS loader itself. The first call to **`HandleProtocol()`** gets the **`LOADED_IMAGE_PROTOCOL`** interface from the *`ImageHandle`* that was passed into the OS loader application. The second call to **`HandleProtocol()`** gets the **`DEVICE_PATH_PROTOCOL`** interface to the device handle of the OS loader image. With these two calls, the device path of the OS loader image is known, and the file path along with other image information is known about the OS loader.

```
BS->HandleProtocol(ImageHandle, &LoadedImageProtocol, LoadedImage);

BS->HandleProtocol(LoadedImage->DeviceHandle, &DevicePathProtocol, &DevicePath);

Print (L"Image device : %s\n", DevicePathToStr (DevicePath));
Print (L"Image file   : %s\n", DevicePathToStr (LoadedImage->FilePath));
Print (L"Image Base   : %X\n", LoadedImage->ImageBase);
Print (L"Image Size   : %X\n", LoadedImage->ImageSize);
```

3.2 Accessing Files in the Device Path of the OS Loader

Section 3.1 shows how to retrieve the device path and the image path of the OS loader image. The following code fragment shows how to use this information to open another file called **`OSKERNEL.BIN`** that resides in the same directory as the OS loader itself. The first step is to use **`HandleProtocol()`** to get the **`FILE_SYSTEM_PROTOCOL`** interface to the device handle retrieved in the previous section. Then, the disk volume can be opened so file access calls can be made. The end result is that the variable *`CurDir`* is a file handle to the same partition in which the OS loader resides.

```
BS->HandleProtocol(LoadedImage->DeviceHandle, &FileSystemProtocol, &Vol);
Vol->OpenVolume (Vol, &RootFs);
CurDir = RootFs;
```

The next step is to build a file path to **`OSKERNEL.BIN`** that exists in the same directory as the OS loader image. Once the path is built, the file handle *`CurDir`* can be used to call **`Open()`**, **`Close()`**, **`Read()`**, and **`Write()`** on the **`OSKERNEL.BIN`** file. The following code fragment builds a file path, opens the file, reads it into an allocated buffer, and closes the file.

```
StrCpy(FileName,DevicePathToStr(LoadedImage->FilePath));
for(i=StrLen(FileName);i>=0 && FileName[i]!='\\';i--);
FileName[i] = 0;
StrCat(FileName,L"\\OSKERNEL.BIN");
CurDir->Open (CurDir, &FileHandle, FileName, EFI_FILE_MODE_READ, 0);
Size = 0x00100000;
BS->AllocatePool(EfiLoaderData, Size, &OsKernelBuffer);
FileHandle->Read(FileHandle, &Size, OsKernelBuffer);
FileHandle->Close(FileHandle);
```

3.3 Finding the OS Partition

The EFI sample environment materializes a **`BLOCK_IO_PROTOCOL`** instance for every partition that is found in a system. An OS loader can search for OS partitions by looking at all the **`BLOCK_IO`** devices. The following code fragment uses **`LibLocateHandle()`** to get a list of **`BLOCK_IO`** device handles. These handles are then used to retrieve the first block from each one of these **`BLOCK_IO`** devices. The **`HandleProtocol()`** API is used to get the **`DEVICE_PATH_PROTOCOL`** and **`BLOCK_IO_PROTOCOL`** instances for each of the **`BLOCK_IO`** devices. The variable *`BlkIo`* is a handle to the **`BLOCK_IO`** device using the **`BLOCK_IO_PROTOCOL`** interface. At this point, a **`ReaddBlocks()`** call can be used to read the first block of a device. The sample OS loader just dumps the contents of the block to the display. A real OS loader would have to test each block read to see if it is a recognized partition. If a recognized partition is found, then the OS loader can implement a simple file system driver using the EFI API **`ReadBlocks()`** to load additional data from that partition.

```
NoHandles = 0;
HandleBuffer = NULL;
LibLocateHandle(ByProtocol, &BlockIoProtocol, NULL, &NoHandles, &HandleBuffer);
for(i=0;i<NoHandles;i++) {
    BS->HandleProtocol (HandleBuffer[i], &DevicePathProtocol, &DevicePath);
    BS->HandleProtocol (HandleBuffer[i], &BlockIoProtocol, &BlkIo);
    Block = AllocatePool (BlkIo->BlockSize);
    MediaId = BlkIo->MediaId;
    BlkIo->ReadBlocks(BlkIo, MediaId, (EFI_LBA)0, BlkIo->BlockSize, Block);
    Print(L"\nBlock #0 of device %s\n",DevicePathToStr(DevicePath));
    DumpHex(0,0,BlkIo->BlockSize,Block);
}
```

3.4 Getting the Current System Configuration

All the system configuration is available through the *`SystemTable`* data structure that is passed into the OS loader. Five tables are available, and their structure and contents are described in the appropriate specifications.

```
LibGetSystemConfigurationTable(&AcpiTableGuid,&AcpiTable);
LibGetSystemConfigurationTable(&SMBIOSTableGuid,&SMBIOSTable);

LibGetSystemConfigurationTable(&SalSystemTableGuid,&SalSystemTable);
LibGetSystemConfigurationTable(&MpsTableGuid,&MpsTable);

Print(L"  ACPI Table is at address           : %X\n",AcpiTable);
Print(L"  SMBIOS Table is at address         : %X\n",SMBIOSTable);
Print(L"  Sal System Table is at address     : %X\n",SalSystemTable);
Print(L"  MPS Table is at address            : %X\n",MpsTable);
```

3.5 Getting the Current Memory Map

There is one EFI Library function that can retrieve the memory map maintained by the EFI environment. The following code fragment shows the use of this Library function, and it displays the contents of the memory map. An OS loader must pay special attention to the *`MapKey`* parameter. Every time that the EFI environment modifies the memory map that it maintains, the *`MapKey`* is incremented. An OS loader needs to pass the current memory map to the OS kernel. Depending on what functions the OS loader calls between the time the memory map is retrieved and the time that **`ExitBootServices()`** is called, the memory map may be modified. In general, the OS loader should retrieve the memory map just before calling **`ExitBootServices()`**. If **`ExitBootServices()`** fails because the *`MapKey`* does not match, then the OS loader must get a new copy of the memory map and try again.

```
MemoryMap = LibMemoryMap(&NoEntries,&MapKey,&DescriptorSize,&DescriptorVersion);

Print(L"Memory Descriptor List:\n\n");
Print(L"  Type        Start Address     End Address       Attributes      \n");
Print(L"  ==========  ================  ================  ================\n");
MemoryMapEntry = MemoryMap;
for(i=0;i<NoEntries;i++) {
    Print(L"  %s  %lX  %lX  %lX\n",
          OsLoaderMemoryTypeDesc[MemoryMapEntry->Type],
          MemoryMapEntry->PhysicalStart,
          MemoryMapEntry->PhysicalStart +
              LShiftU64(MemoryMapEntry->NumberOfPages,PAGE_SHIFT)-1,
          MemoryMapEntry->Attribute);
    MemoryMapEntry = NextMemoryDescriptor(MemoryMapEntry, DescriptorSize);
}
```

3.6 Getting Environment Variables

The following code fragment shows how to extract all the environment variables maintained by the EFI environment. It uses the **GetNextVariableName()** API to walk the entire list.

```
VariableName[0] = 0x0000;
VendorGuid = NullGuid;
Print(L"GUID                                Variable Name        Value\n");
Print(L"=================================== ==================== ========\n");
do {
  VariableNameSize = 256;
  Status = RT-GetNextVariableName(&VariableNameSize,VariableName,&VendorGuid);
  if (Status == EFI_SUCCESS) {
    VariableValue = LibGetVariable(VariableName,&VendorGuid);
    Print(L"%.-35g %.-20s %X\n",&VendorGuid,VariableName,VariableValue);
  }
} while (Status == EFI_SUCCESS);
```

3.7 Transitioning to an OS Kernel

A single call to **ExitBootServices()** terminates all the EFI Boot Services that the EFI environment provides. From that point on, only the EFI Runtime Services may be used. Once this call is made, the OS loader needs to prepare for the transition to the OS kernel. It is assumed that the OS kernel has full control of the system and that only a few firmware functions are required by the OS kernel. These functions are the EFI Runtime Services. The OS loader must pass the *SystemTable* to the OS kernel, so the OS kernel can make the Runtime Services calls. The exact mechanism that is used to transition from the OS loader to the OS kernel is implementation dependent. It is important to note that the OS loader could transition to the OS kernel prior to calling **ExitBootServices()**. In this case, the OS kernel is responsible for calling **ExitBootServices()** before taking full control of the system.

Appendix A
EFI Boot Services

This section provides a high-level description of the EFI Boot Services. These services are APIs that are exported by the EFI environment, and they are available only at boot time before an OS takes control of the system. Once a call to **`ExitBootServices()`** is made, the EFI Boot Services APIs may no longer be used. The EFI Boot Services are organized into the following groups, which are described in the following subsections:

- Event Services
- Memory Allocation Services
- Protocol Handler Services
- Image Services
- Miscellaneous Services

A.1 Event Services

Table A-1 lists the functions that are used to manage events.

Table A-1. Event Services

Name	**Description**
CreateEvent	Creates a general-purpose event structure.
CloseEvent	Closes and frees an event structure.
SignalEvent	Signals an event.
WaitForEvent	Waits for an event to be signaled.
CheckEvent	Checks whether an event is in the signaled state.
SetTimer	Sets the type of timer and the trigger time for a timer event.
RaiseTPL	Raises the task priority level and returns its previous level.
RestoreTPL	Restores and lowers the task priority level to its previous level.

A.2 Memory Allocation Services

Table A-2 lists the functions that are used to allocate and free memory.

Table A-2. Memory Allocation Services

Name	**Description**
AllocatePages	Allocates memory pages of a particular type from the system.
FreePages	Frees memory pages.
GetMemoryMap	Returns the current boot services memory map and memory map key.
AllocatePool	Allocates pool of a particular type.
FreePool	Frees allocated pool and returns it to the system.

A.3 Protocol Handler Services

Table A-3 lists the functions that are used to manage protocol handles.

Table A-3. Protocol Handler Services

Name	Description
InstallProtocolInterface	Adds a protocol handler onto an existing or new device handle.
UninstallProtocolInterface	Removes a protocol handler from a device handle.
ReinstallProtocolInterface	Replaces a protocol interface.
HandleProtocol	Queries the list of protocol handlers on a device handle for the requested protocol interface.
PCHandleProtocol	Queries the list of protocol handlers on a device handle for the requested protocol interface. This function was included in the 0.99 version of the *EFI Specification* and was removed from the *EFI 1.10 Specification;* this function is required, however, for compatibility with EFI 1.02. To be used by pseudo code (P-code) EFI drivers.
RegisterProtocolNotify	Registers an event that is to be signaled whenever an interface for a particular protocol interface is installed.
LocateHandle	Locates the handle(s) that support the specified protocol.
LocateDevicePath	Locates the closest handle that supports the specified protocol on the specified device path.
OpenProtocol	Queries a handle to determine if it supports a specified protocol. If the protocol is supported by the handle, it opens the protocol on behalf of the calling agent.
CloseProtocol	Closes a protocol on a handle that was opened using **`OpenProtocol()`**.
OpenProtocolInformation	Retrieves the list of agents that currently have a protocol interface opened.
ConnectController	Connects one or more drivers to a controller.
DisconnectController	Disconnects one or more drivers from a controller.
ProtocolsPerHandle	Retrieves the list of protocol interface GUIDs that are installed on a handle in a buffer allocated from pool. The return buffer is automatically allocated.
LocateHandleBuffer	Returns an array of handles that support the requested protocol in a buffer allocated from pool. The return buffer is automatically allocated.
LocateProtocol	Returns the first protocol instance that matches the given protocol.
InstallMultipleProtocolInterfaces	Installs one or more protocol interfaces onto a handle in the boot services environment.
UninstallMultipleProtocolInterfaces	Uninstalls one or more protocol interfaces from a handle in the boot services environment.

A.4 Image Services

Table A-4 lists the functions that are used to load, execute, and exit EFI images.

Table A-4. Image Services

Name	Description
LoadImage	Loads an EFI image into memory.
StartImage	Transfers control to a loaded image's entry point.
UnloadImage	Unloads an image loaded with **`LoadImage()`**.
EFI_IMAGE_ENTRY_POINT	The declaration of an EFI image entry point. This can be the entry point to an EFI application, an EFI boot services driver, or an EFI runtime driver.
Exit	Terminates the currently loaded EFI image and returns control to boot services.
ExitBootServices	Terminates all boot services.

A.5 Miscellaneous Services

Table A-5 lists the miscellaneous EFI Boot Services functions. These functions are boot services that do not fall under the other boot services categories but that are required to complete the definition of the EFI environment.

Table A-5. Miscellaneous Services

Name	Description
SetWatchdogTimer	Resets and sets the system's watchdog timer.
Stall	Stalls the processor for a specified number of microseconds.
CopyMem	Copies the contents of one buffer to another buffer
SetMem	Fills a buffer with a specified value.
GetNextMonotonicCount	Returns a monotonically increasing count for the platform.
InstallConfigurationTable	Adds, updates, or removes a configuration table entry from the EFI System Table.
CalculateCrc32	Computes and returns a 32-bit CRC for a data buffer.

Appendix B
EFI Runtime Services

This section provides a high-level description of the EFI Runtime Services. These services are APIs that are exported by the EFI environment, and they are always available. The EFI Runtime Services are organized into the following groups, which are described in the following subsections:

- Variable Services
- Time Services
- Virtual Memory Services
- Miscellaneous Services

B.1 Variable Services

The Variable Services are used to maintain environment variables in nonvolatile storage. Table B-1 lists the functions that are available for retrieving, creating, modifying, and deleting environment variables in nonvolatile storage.

Table B-1. Variable Services

Name	Description
GetVariable	Returns the value of the specified variable.
GetNextVariableName	Enumerates the current variables names.
SetVariable	Sets the value of the specified variable.

B.2 Time Services

Table B-2 lists the functions that are available to access the system's Real Time Clock (RTC) and timer hardware.

Table B-2. Time Services

Name	Description
GetTime	Returns the current time and date information and the time-keeping capabilities of the hardware platform.
SetTime	Sets the current time and date.
GetWakeupTime	Returns the current wakeup alarm clock setting.
SetWakeupTime	Sets the current wakeup alarm clock setting.

B.3 Virtual Memory Services

Table B-3 lists the functions that are used transition the firmware from flat physical mode to virtual runtime mode.

Table B-3. Virtual Memory Services

Name	Description
SetVirtualAddressMap	Used by an OS loader to convert from the runtime addressing mode of EFI firmware from physical addressing to virtual addressing.
ConvertPointer	Used by EFI components to determine the new virtual address that is to be used on subsequent memory accesses.

B.4 Miscellaneous Services

Table B-4 lists the miscellaneous runtime services.

Table B-4. Miscellaneous Services

Name	Description
ResetSystem	Resets all processors and devices and reboots the system.
GetNextHighMonotonicCount	Returns the next high 32 bits of the platform's monotonic counter.

Appendix C
EFI Protocol Interfaces

This section provides a high-level description of the EFI protocol interfaces. These services are APIs that are exported by the EFI environment through GUID-based protocol handles. The standard set of protocol interfaces are listed here. Additional protocol interfaces may be added to a system by an EFI device driver. The EFI protocol interfaces are organized into the following groups, which are described in the following subsections:

- EFI Loaded Image Protocol
- Device Path Protocol
- EFI Driver Binding Protocol
- EFI Platform Driver Override Protocol
- EFI Bus Specific Driver Override Protocol
- EFI Driver Configuration Protocol
- EFI Driver Diagnostics Protocol
- EFI Component Name Protocol
- Simple Input Protocol
- Simple Text Output Protocol
- UGA Draw Protocol
- UGA I/O Protocol
- Simple Pointer Protocol
- Serial I/O Protocol
- Load File Protocol
- File System Protocol
- **EFI_FILE** Protocol
- Disk I/O Protocol
- Block I/O Protocol
- Unicode Collation Protocol
- PCI Root Bridge I/O Protocol
- EFI PCI I/O Protocol
- SCSI Pass Thru Protocol
- USB Host Controller Protocol
- EFI USB I/O Protocol
- Simple Network Protocol
- Network Interface Identifier Protocol
- PXE Base Code Protocol
- PXE Base Code Callback Protocol
- Boot Integrity Services Protocol

- EFI Debug Support Protocol
- EFI Debugport Protocol
- Decompress Protocol
- Device I/O Protocol
- EBC Interpreter Protocol

C.1 EFI Loaded Image Protocol

Table C-1 lists the functions that are used by an EFI image that has been loaded into memory.

Table C-1. EFI Loaded Image Protocol Interfaces

Name	Description
Unload	Unloads an image from memory.

C.2 Device Path Protocol

This protocol does not contain any interfaces. It simply returns a pointer to the device path for the protocol instance.

C.3 EFI Driver Binding Protocol

Table C-2 lists the functions that are used to manage a controller.

Table C-2. EFI Driver Binding Protocol Interfaces

Name	Description
Supported	Tests if the driver supports a given controller.
Start	Starts a controller using the driver.
Stop	Stops a controller using the driver.

C.4 EFI Platform Driver Override Protocol

Table C-3 lists the functions that are used to match one or more drivers to a controller.

Table C-3. EFI Platform Driver Override Protocol Interfaces

Name	Description
GetDriver	Retrieves the image handle of the platform override driver for a controller in the system.
GetDriverPath	Retrieves the device path of the platform override driver for a controller in the system.
DriverLoaded	Associates a device path to an image handle

C.5 EFI Bus Specific Driver Override Protocol

Table C-4 lists the function that are used to override the default driver selection algorithm.

Table C-4. EFI Bus Specific Driver Override Protocol Interfaces

Name	Description
GetDriver	Uses a bus-specific algorithm to retrieve a driver image handle for a controller.

C.6 EFI Driver Configuration Protocol

Table C-5 lists the functions that are used to set configuration options for a controller.

Table C-5. EFI Driver Configuration Protocol Interfaces

Name	Description
SetOptions	Allows the user to set driver-specific configuration options for a controller.
OptionsValid	Tests to see if a controller’s current configuration options are valid.
ForceDefaults	Forces a driver to set the default configuration options for a controller.

C.7 EFI Driver Diagnostics Protocol

Table C-6 lists the function that are used to perform diagnostics on a controller.

Table C-6. EFI Driver Diagnostics Protocol Interfaces

Name	Description
RunDiagnostics	Runs diagnostics on a controller.

C.8 EFI Component Name Protocol

Table C-7 lists the functions that are used to retrieve user-readable names of EFI drivers and controllers.

Table C-7. EFI Component Name Protocol Interfaces

Name	Description
GetDriverName	Retrieves a Unicode string that is the user-readable name of the driver.
GetControllerName	Retrieves a user-readable Unicode name of a controller managed by the driver.

C.9 Simple Input Protocol

Table C-8 lists the functions that are used to access a simple input hardware device.

Table C-8. Simple Input Protocol Interfaces

Name	Description
Reset	Resets a simple input device.
ReadKeyStroke	Reads a keystroke from a simple input device.

C.10 Simple Text Output Protocol

Table C-9 lists the functions that are used to access the standard output device, the standard error device, or any other text-based output device.

Table C-9. Simple Text Output Protocol Interfaces

Name	Description
Reset	Resets the `ConsoleOut` device.
OutputString	Displays the Unicode string on the device at the current cursor location.
TestString	Tests to see if the `ConsoleOut` device supports this Unicode string.
QueryMode	Queries information concerning the output device's supported text mode.
SetMode	Sets the current mode of the output device.
SetAttribute	Sets the foreground and background color of the text that is output.
ClearScreen	Clears the screen with the currently set background color.
SetCursorPosition	Sets the current cursor position.
EnableCursor	Turns on and off the visibility of the cursor.

C.11 UGA Draw Protocol

Table C-10 lists the functions that are used to access a graphics controller.

Table C-10.UGA Draw Protocol Interfaces

Name	Description
GetMode	Returns information about the geometry and configuration of the graphics controller's current frame buffer configuration.
SetMode	Sets the graphics device into a given mode and clears the frame buffer to black.
Blt	Draws on the video device's frame buffer.

C.12 UGA I/O Protocol

Table C-11 lists the functions that are used to send I/O requests to the graphics device.

Table C-11.UGA I/O Protocol Interfaces

Name	Description
CreateDevice	Creates a `UGA_DEVICE` object.
DeleteDevice	Deletes the `UGA_DEVICE` that was returned from `CreateDevice()`.
DispatchService	Dispatches I/O requests to the display device and its associate child devices.

C.13 Simple Pointer Protocol

Table C-12 lists the functions that are used to access a pointer device.

Table C-12.Simple Pointer Protocol Interfaces

Name	Description
Reset	Resets the pointer device.
GetState	Retrieves the current state of the pointer device.

C.14 Serial I/O Protocol

Table C-13 lists the functions that are used to access a serial I/O device.

Table C-13.Serial I/O Protocol Interfaces

Name	Description
Reset	Resets the hardware device.
SetAttributes	Sets communication parameters for a serial device. These include the baud rate, receive FIFO depth, transmit/receive timeout, parity, data bits, and stop bit attributes.
SetControl	Sets the control bits on a serial device. These include Request to Send and Data Terminal Ready.
GetControl	Reads the status of the control bits on a serial device. These include Clear to Send, Data Set Ready, Ring Indicator, and Carrier Detect.
Write	Sends a buffer of characters to a serial device.
Read	Receives a buffer of characters from a serial device.

C.15 Load File Protocol

Table C-14 lists the functions that are used to load drivers from sources other than a file system.

Table C-14.Load File Protocol Interfaces

Name	Description
LoadFile	Causes the driver to load the requested file.

C.16 File System Protocol

Table C-15 lists the functions that are used to open a partition for file access.

Table C-15.File System Protocol Interfaces

Name	Description
OpenVolume	Opens the volume for file I/O access.

C.17 EFI_FILE Protocol

Table C-16 lists the functions that are used to maintain files in a partition containing a supported file system.

Table C-16.EFI_FILE Protocol Interfaces

Name	Description
Open	Opens or creates a new file.
Close	Closes the current file handle.
Delete	Deletes a file.
Read	Reads bytes from the file.
Write	Writes bytes to the file.
GetPosition	Returns the current file position.
SetPosition	Sets the current file position.
GetInfo	Gets the requested file or volume information.
SetInfo	Sets the requested file information.
Flush	Flushes all modified data that is associated with the file to the device.

C.18 Disk I/O Protocol

Table C-17 lists the functions that are used to access a block-based I/O device as a byte-stream device.

Table C-17.Disk I/O Protocol Interfaces

Name	Description
ReadDisk	Reads data from the disk.
WriteDisk	Writes data to the disk.

C.19 Block I/O Protocol

Table C-18 lists the functions that are used to access a block-based I/O device.

Table C-18.Block I/O Protocol Interfaces

Name	Description
Reset	Resets the block device hardware.
ReadBlocks	Reads the requested number of blocks from the device.
WriteBlocks	Writes the requested number of blocks to the device.
FlushBlocks	Flushes any cached blocks.

C.20 Unicode Collation Protocol

Table C-19 lists the functions that are used to perform case-insensitive Unicode string comparisons.

Table C-19.Unicode Collation Protocol Interfaces

Name	Description
StriColl	Performs a case-insensitive comparison between two Unicode strings.
MetaiMatch	Performs a case-insensitive comparison between a Unicode pattern string and a Unicode string. The pattern string can use the '?' wildcard for match any character, and the '*' wildcard for match any substring.
StrLwr	Converts characters in a Unicode string to uppercase characters.
StrUpr	Converts characters in a Unicode string to lowercase characters.
FatToStr	Converts a FAT 8.3 file name to a Unicode string
StrToFat	Converts a Null-terminated Unicode string to legal characters in a FAT file name using an original equipment manufacturer (OEM) character set.

C.21 PCI Root Bridge I/O Protocol

Table C-20 lists the functions that are used to access PCI controllers behind a PCI root bridge controller.

Table C-20.PCI Root Bridge I/O Protocol Interfaces

Name	Description
PollMem	Polls an address in memory-mapped I/O space until an exit condition is met or a timeout occurs.
PollIo	Polls an address in I/O space until an exit condition is met or a timeout occurs.
Mem	Allows reads from and writes to memory-mapped I/O space.
Io	Allows reads from and writes to I/O space.
Pci	Allows reads from and writes to PCI configuration space.
CopyMem	Allows one region of PCI root bridge memory space to be copied to another region of PCI root bridge memory space.
Map	Provides the PCI controller–specific addresses that are needed to access system memory for Direct Memory Access (DMA).
Unmap	Releases any resources that were allocated by **`Map()`**.
AllocateBuffer	Allocates pages that are suitable for a common buffer mapping.
FreeBuffer	Free pages that were allocated with **`AllocateBuffer()`**.
Flush	Flushes all PCI posted write transactions to system memory.
GetAttributes	Gets the attributes that a PCI root bridge supports setting with **`SetAttributes()`** and the attributes that a PCI root bridge is currently using.
SetAttributes	Sets attributes for a resource range on a PCI root bridge.
Configuration	Gets the current resource settings for this PCI root bridge.

C.22 EFI PCI I/O Protocol

Table C-21 lists the functions that are used to access memory and I/O on a PCI controller.

Table C-21.EFI PCI I/O Protocol Interfaces

Name	Description
PollMem	Polls an address in PCI memory space until an exit condition is met or a timeout occurs.
PollIo	Polls an address in PCI I/O space until an exit condition is met or a timeout occurs.
Mem	Allows Base Address Register (BAR) relative reads from and writes to PCI memory space.
Io	Allows BAR relative reads from and writes to PCI I/O space.
Pci	Allows PCI controller relative reads from and writes to PCI configuration space.
CopyMem	Allows one region of PCI memory space to be copied to another region of PCI memory space.
Map	Provides the PCI controller–specific address that are needed to access system memory for DMA.
Unmap	Releases any resources that were allocated by **`Map()`**.
AllocateBuffer	Allocates pages that are suitable for a common buffer mapping.
FreeBuffer	Frees pages that were allocated with **`AllocateBuffer()`**.
Flush	Flushes all PCI posted write transactions to system memory.
GetLocation	Retrieves this PCI controller’s current PCI bus number, device number, and function number.
Attributes	Performs an operation on the attributes that this PCI controller supports.
GetBarAttributes	Gets the attributes that this PCI controller supports
SetBarAttributes	Sets the attributes for a range of a BAR on a PCI controller.

C.23 SCSI Pass Thru Protocol

Table C-22 lists the functions that are used to access SCSI devices.

Table C-22.SCSI Pass Thru Protocol Interfaces

Name	Description
PassThru	Sends a SCSI Request Packet to a SCSI device that is connected to the SCSI channel.
GetNextDevice	Used to retrieve the list of legal Target IDs and Logical Unit Numbers (LUNs) for the SCSI devices on a SCSI channel.
BuildDevicePath	Used to allocate and build a device path node for a SCSI device on a SCSI channel.
GetTargetLun	Used to translate a device path node to a Target ID and LUN.
ResetChannel	Resets the SCSI channel.
ResetTarget	Resets a SCSI device that is connected to the SCSI channel.

C.24 USB Host Controller Protocol

Table C-23 lists the functions that are used to access a USB host controller.

Table C-23.USB Host Controller Protocol Interfaces

Name	Description
Reset	Provides a software reset for the USB host controller.
GetState	Retrieves the current state of the USB host controller.
SetState	Sets the USB host controller to a specific state.
ControlTransfer	Submits a control transfer to a target USB device.
BulkTransfer	Submits a bulk transfer to a bulk endpoint of a USB device.
AsyncInterruptTransfer	Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device
SyncInterruptTransfer	Submits a synchronous interrupt transfer to an interrupt endpoint of a USB device.
IsochronousTransfer	Submits an isochronous transfer to an isochronous endpoint of a USB device.
AsyncIsochronousTransfer	Submits a nonblocking USB isochronous transfer.
GetRootHubPortNumber	Retrieves the number of root hub ports that are produced by the USB host controller.
GetRootHubPortStatus	Retrieves the status of the specified root hub port.
SetRootHubPortFeature	Sets the feature for the specified root hub port.
ClearRootHubPortFeature	Clears the feature for the specified root hub port.

C.25 EFI USB I/O Protocol

Table C-24 lists the functions that are used to access the USB device.

Table C-24.EFI USB I/O Protocol Interfaces

Name	Description
UsbControlTransfer	Accesses the USB device through the USB Control Transfer Pipe.
UsbBulkTransfer	Accesses the USB device through the USB Bulk Transfer Pipe.
UsbAsyncInterruptTransfer	Submits a nonblocking USB interrupt transfer.
UsbSyncInterruptTransfer	Accesses the USB device through the USB Synchronous Interrupt Transfer Pipe.
UsbIsochronousTransfer	Accesses the USB device through USB Isochronous Transfer Pipe.
UsbAsyncIsochronousTransfer	Submits a nonblocking USB isochronous transfer.
UsbGetDeviceDescriptor	Retrieves the device descriptor of a USB device.
UsbGetConfigDescriptor	Retrieves the activated configuration descriptor of a USB device.
UsbGetInterfaceDescriptor	Retrieves the interface descriptor of a USB controller.
UsbGetEndpointDescriptor	Retrieves the endpoint descriptor of a USB controller.
UsbGetStringDescriptor	Retrieves the string descriptor inside a USB device.
UsbGetSupportedLanguages	Retrieves the array of languages that the USB device supports.
UsbPortReset	Resets and reconfigures the USB controller.

C.26 Simple Network Protocol

Table C-25 lists the functions that are used to provide a packet-level interface to a network adapter.

Table C-25.Simple Network Protocol Interfaces

Name	Description
Start	Changes the state of a network interface from “stopped” to “started.”
Stop	Changes the state of a network interface from “started” to “stopped.”
Initialize	Resets a network adapter and allocates the transmit and receive buffers that are required by the network interface; optionally, also requests allocation of additional transmit and receive buffers.
Reset	Resets a network adapter and reinitializes it with the parameters that were provided in the previous call to `Initialize()`.
Shutdown	Resets a network adapter and leaves it in a state that is safe for another driver to initialize.
ReceiveFilters	Manages the multicast receive filters of a network interface.
StationAddress	Modifies or resets the current station address, if supported.
Statistics	Resets or collects the statistics on a network interface.
MCastIPtoMAC	Converts a multicast IP address to a multicast hardware Media Access Controller (MAC) address.

continued

Table C-25.Simple Network Protocol Interfaces (continued)

Name	Description
NvData	Performs read and write operations on the nonvolatile RAM (NVRAM) device that is attached to a network interface.
GetStatus	Reads the current interrupt status and recycled transmit buffer status from a network interface.
Transmit	Places a packet in the transmit queue of a network interface.
Receive	Receives a packet from a network interface.

C.27 Network Interface Identifier Protocol

This protocol is used to describe details about the software layer that is used to produce the Simple Network Protocol. This protocol is optional and does not contain any interfaces.

C.28 PXE Base Code Protocol

Table C-26 lists the functions that are used to access Preboot Execution Environment (PXE)–compatible devices for network access and network booting.

Table C-26.PXE Base Code Protocol Interfaces

Name	Description
Start	Enables the use of the PXE Base Code Protocol functions.
Stop	Disables the use of the PXE Base Code Protocol functions.
Dhcp	Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R. (solicit / advertise / request / reply) sequence.
Discover	Attempts to complete the PXE Boot Server and/or boot image discovery sequence.
Mtftp	Used to perform TFTP and MTFTP services.
UdpWrite	Writes a User Datagram Protocol (UDP) packet to the network interface.
UdpRead	Reads a UDP packet from the network interface.
SetIpFilter	Updates the IP receive filters of a network device and enables software filtering.
Arp	Uses the Address Resolution Protocol (ARP) to resolve a MAC address.
SetParameters	Updates the parameters that affect the operation of the PXE Base Code Protocol.
SetStationIp	Updates the station IP address and/or subnet mask values of a network device.
SetPackets	Updates the contents of the cached DHCP and Discover packets.

C.29 PXE Base Code Callback Protocol

Table C-27 lists the functions that are used when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet.

Table C-27.PXE Base Code Callback Protocol Interfaces

Name	Description
Callback	Callback function that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet.

C.30 Boot Integrity Services Protocol

Table C-28 lists the functions that are used to check a digital signature of a data block.

Table C-28.Boot Integrity Services Protocol Interfaces

Name	Description
Initialize	Initializes an application instance of the **EFI_BIS** Protocol.
Shutdown	Ends the lifetime of an application instance of the **EFI_BIS** Protocol.
Free	Frees memory structures that were allocated and returned by other functions in the **EFI_BIS** Protocol.
GetBootObjectAuthorization Certificate	Retrieves the current digital certificate (if any) used by the **EFI_BIS** Protocol.
GetBootObjectAuthorization CheckFlag	Retrieves the current setting of the authorization check flag that indicates whether or not authorization checks are required for boot objects..
GetBootObjectAuthorization UpdateToken	Retrieves an uninterpreted token whose value gets included and signed in a subsequent request to alter the configuration parameters, to protect against attempts to “replay” such a request.
GetSignatureInfo	Retrieves information about the digital signature
UpdateBootObject Authorization	Requests that the configuration parameters be altered by installing or removing an authorization certificate or changing the setting of the check flag.
VerifyBootObject	Verifies a boot object according to the supplied digital signature and the current authorization certificate and check flag setting.
VerifyObjectWithCredential	Verifies a data object according to a supplied digital signature and a supplied digital certificate.

C.31 EFI Debug Support Protocol

Table C-29 lists the functions that are used to support a debug agent.

Table C-29.EFI Debug Support Protocol Interfaces

Name	Description
GetMaximumProcessorIndex	Returns the maximum processor index value that may be used with `RegisterPeriodicCallback()` and `RegisterExceptionCallback()`.
RegisterPeriodicCallback	Registers a callback function that will be invoked periodically and asynchronously to the execution of EFI.
RegisterExceptionCallback	Registers a callback function that will be called each time the specified processor exception occurs.
InvalidateInstructionCache	Invalidates the instruction cache of the processor.

C.32 EFI Debugport Protocol

Table C-30 lists the functions that are used by the debug agent to communicate with the remote debug host.

Table C-30.EFI Debugport Protocol Interfaces

Name	Description
Reset	Resets the debugport hardware.
Write	Sends a buffer of characters to the debugport device.
Read	Receives a buffer of characters from the debugport device.
Poll	Determines if there is any data available to be read from the debugport device.

C.33 Decompress Protocol

Table C-31 lists the functions that are used to provide a decompression service.

Table C-31.Decompress Protocol Interfaces

Name	Description
GetInfo	Retrieves the size of the uncompressed destination buffer and the size of the scratch buffer required to perform the decompression.
Decompress	Decompresses a compressed source buffer into an uncompressed destination buffer.

C.34 Device I/O Protocol

Table C-32 lists the functions that are used by device drivers to access hardware devices in a bus-specific manner.

Table C-32.Device I/O Protocol Interfaces

Name	Description
Mem	Reads and writes to memory-mapped I/O space on a bus.
Io	Reads and writes to I/O ports on a bus.
Pci	Reads and writes to PCI configuration space.
Map	Provides the device-specific addresses that are needed to access host memory for DMA.
PciDevicePath	Provides an EFI device path for a PCI device with the given PCI configuration space address.
Unmap	Releases any resources that were allocated with `Map()`.
AllocateBuffer	Allocates pages that are suitable for a common buffer mapping.
Flush	Flushes any posted write data to the device.
FreeBuffer	Frees pages that were allocated with `AllocateBuffer()`.

C.35 EBC Interpreter Protocol

Table C-33 lists the functions that are used to allow execution of EFI Byte Code (EBC) images.

Table C-33.EBC Interpreter Protocol Interfaces

Name	Description
CreateThunk	Creates a thunk for an EBC image entry point or protocol service and returns a pointer to the thunk.
UnloadImage	Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution.
RegisterICacheFlush	Registers a callback function that the EBC interpreter can call to flush the processor instruction cache after creating thunks.
GetVersion	Called to get the version of the associated EBC interpreter.

Appendix D
EFI Library Functions and Macros

This section provides a high-level description of the EFI Library functions and macros. These library functions are complementary to the APIs described in the *EFI Specification.* EFI Library functions and macros are organized into the following groups, which are described in the following subsections. See the *EFI Library Specification* for definitions of the functions and macros that are included in this section.

- Initialization Functions
- Linked List Support Macros
- String Functions
- Memory Support Functions
- Text I/O Functions
- Math Functions
- Spin Lock Functions
- Handle and Protocol Functions
- File I/O Support Functions
- Device Path Support Functions
- Miscellaneous Functions

D.1 Initialization Functions

The initialization functions in the EFI Library are used to initialize the execution environment so that other EFI Library function may be used. Table D-1 lists the initialization support.

Table D-1. Initialization Functions

Name	Description
InitializeLib	Initializes the EFI Library.
InitializeUnicodeSupport	Initializes the use of language-dependant Unicode library functions.

D.2 Linked List Support Macros

The EFI Library supplies a set of macros that allow doubly linked lists to be created and maintained. Table D-2 contains the list of macros.

Table D-2. Linked List Support Macros

Name	Description
InitializeListHead	Initializes the head node of a doubly linked list.
IsListEmpty	Determines if a doubly linked list is empty.
RemoveEntryList	Removes a node from a doubly linked list.
InsertTailList	Adds a node to the end of a doubly linked list.
InsertHeadList	Adds a node to the beginning of a doubly linked list.

D.3 String Functions

The string functions in the EFI Library perform operations on Unicode and ASCII strings. Table D-3 contains the list of string support functions.

Table D-3. String Functions

Name	Description
StrCmp	Compares two Unicode strings.
StrnCmp	Compares a portion of two Unicode strings.
StriCmp	Performs a case insensitive comparison of two Unicode strings.
StrCpy	Copies one Unicode string to another Unicode string.
StrCat	Concatenates two Unicode strings.
StrLen	Determines the length of a Unicode string.
StrSize	Determines the size of a Unicode string in bytes.
StrDuplicate	Creates a duplicate of a Unicode string.
StrLwr	Converts characters in a Unicode string to lowercase characters.
StrUpr	Converts characters in a Unicode string to uppercase characters.
strlena	Determines the length of an ASCII string.
strcmpa	Compares two ASCII strings.
strncmpa	Compares a portion of two ASCII strings.
xtoi	Converts a hexadecimal-formatted Unicode string to an integer.
Atoi	Converts a decimal-formatted Unicode string to an integer.

continued

Table D-3. String Functions (continued)

Name	**Description**
MetaMatch	Checks to see if a Unicode string matches a given pattern.
MetaiMatch	Performs a case-insensitive comparison of a Unicode pattern string and a Unicode string.
ValueToString	Converts an integer to a decimal-formatted Unicode string.
ValueToHex	Converts an integer to a hexadecimal-formatted Unicode string.
TimeToString	Converts a data structure containing the time and date into a Unicode string.
GuidToString	Converts a 128-bit GUID into a Unicode string.
StatusToString	Converts an **`EFI_STATUS`** value into a Unicode string.
DevicePathToStr	Converts a device path data structure into a Unicode string.

D.4 Memory Support Functions

The EFI Library provides a set of functions that operating on buffers in memory. Buffers can either be allocated on the stack, as global variables, or from the memory pool Table D-4 contains the list of memory support functions.

Table D-4. Memory Support Functions

Name	**Description**
ZeroMem	Fills a buffer with zeros.
SetMem	Fills a buffer with a value.
CopyMem	Copies the contents of one buffer to another buffer.
CompareMem	Compares the contents of two buffers.
AllocatePool	Allocates a buffer from pool.
AllocateZeroPool	Allocates a buffer from pool and fills it with zeros.
ReallocatePool	Adjusts the size of a previously allocated buffer.
FreePool	Frees a previously allocated buffer.
GrowBuffer	Allocates a new buffer or grows the size of a previously allocated buffer.
LibMemoryMap	Retrieves the system’s current memory map.

D.5 Text I/O Functions

The text I/O functions in the EFI Library provide a simple means to get input and output from a console device. Table D-5 lists the text I/O functions.

Table D-5. Text I/O Functions

Name	Description
Input	Inputs a Unicode string at the current cursor location using the console-in and console-out device.
Iinput	Inputs a Unicode string at the current cursor location using the specified input and output devices.
Output	Sends a Unicode string to the console-out device at the current cursor location.
Print	Sends a formatted Unicode string to the console-out device at the current cursor location.
PrintAt	Sends a formatted Unicode string to the specified location on the console-out device.
Iprint	Sends a formatted Unicode string to the specified output device.
IprintAt	Sends a formatted Unicode string to the specified location of the specified console device.
Aprint	Sends a formatted Unicode string to the console-out device using an ASCII format string.
Sprint	Sends a formatted Unicode string to the specified buffer.
PoolPrint	Sends a formatted Unicode string to a buffer allocated from pool.
CatPrint	Concatenates a formatted Unicode string to a string allocated from pool.
DumpHex	Prints the contents of a buffer in hexadecimal format.
LibIsValidTextGraphics	Determines if a graphic is a supported Unicode box drawing character.
IsValidAscii	Determines if a character is legal ASCII element.
IsValidEfiCntlChar	Determines if a character is an EFI control character.

D.6 Math Functions

The EFI Library provides a few math functions to operate on 64-bit operands. These operations include shift operations, multiplication, and division. Table D-6 lists the set of 64-bit math functions.

Table D-6. Math Functions

Name	Description
LshiftU64	Shifts a 64-bit integer left between 0 and 63 bits.
RshiftU64	Shifts a 64-bit integer right between 0 and 63 bits.
MultiU64x32	Multiplies a 64-bit unsigned integer by a 32-bit unsigned integer and generates a 64-bit unsigned result.
DivU64x32	Divides a 64-bit unsigned integer by a 32-bit unsigned integer and generates a 64-bit unsigned result with an optional 32-bit unsigned remainder.

D.7 Spin Lock Functions

Spin locks are used to protect data structures that may be updated by more than one processor at a time or a single processor that may update the same data structure while running a several different priority levels. Table D-7 lists the support functions for creating and maintaining spin locks.

Table D-7. Spin Lock Functions

Name	Description
InitializeLock	Initializes a spin lock.
AcquireLock	Acquires a spin lock.
ReleaseLock	Releases a spin lock.

D.8 Handle and Protocol Support Functions

The EFI Library contains a set of functions that help drivers maintain the protocol interfaces in the EFI Boot Services environment. Table D-8 lists the set of helper functions.

Table D-8. Handle and Protocol Support Functions

Name	Description
LibLocateHandle	Finds all device handles that match the specified search criteria.
LibLocateHandleByDiskSignature	Finds all device handles that support the Block I/O Protocol and have a disk with a matching disk signature.
LibLocateProtocol	Finds the first protocol instance that matches a given protocol.
LibInstallProtocolInterfaces	Installs one or more protocol interfaces into the EFI Boot Services environment.
LibUninstallProtocolInterfaces	Removes one or more protocol interfaces from the EFI Boot Services environment.
LibReinstallProtocolInterfaces	Reinstalls one or more protocol interfaces into the EFI Boot Services environment.
LibScanHandleDatabase	Scans the handle database and collects EFI Driver Model–related information.
LibGetManagingDriverBindingHandles	Retrieves the list of Driver Binding handles that are managing a controller.
LibGetParentControllerHandles	Retrieves the list of controllers that are parents of another controller.
LibGetChildControllerHandles	Retrieves the list of controllers that are children of another controller.
LibGetManagedControllerHandles	Retrieves the list of controllers that a driver is managing.
LibGetManagedChildControllerHandles	Retrieves the list of child controllers that a driver has produced.

D.9 File I/O Support Functions

Table D-9 lists the set of file I/O support functions.

Table D-9. File I/O Support Functions

Name	Description
LibOpenRoot	Opens and returns a file handle to a root directory of a volume.
LibFileInfo	Retrieves the file information on an open file handle.
LibFileSystemInfo	Retrieves the file system information on an open file handle.
LibFileSystemVolumeLabelInfo	Retrieves the file system information on an open file handle.
ValidMBR	Determines if a hard drive’s MBR is valid.
OpenSimpleReadFile	Opens a file from several possible sources and returns a file handle.
ReadSimpleReadFile	Reads from a file that was opened with **`OpenSimpleReadFile()`**.
CloseSimpleReadFile	Closes a file that was opened with **`OpenSimpleReadFile()`**.

D.10 Device Path Support Functions

Table D-10 lists the support functions for creating and maintaining device path data structures.

Table D-10.Device Path Support Functions

Name	Description
DevicePathFromHandle	Retrieves the device path from a specified handle.
DevicePathInstance	Retrieves the next device path instance from a device path.
DevicePathInstanceCount	Returns the number of device path instances in a device path.
AppendDevicePath	Appends a device path to all the instances of another device path.
AppendDevicePathNode	Appends a device path node to all the instances of a device path.
AppendDevicePathInstance	Appends a device path instance to a device path.
AllocateAndAppendDevicePathNode	Appends a file path to a device path.
FileDevicePath	Returns the size of a device path in bytes.
DevicePathSize	Creates a new copy of a device path.
DuplicateDevicePath	Retrieves a protocol interface for a device.
LibDevicePathToInterface	Naturally aligns all the nodes in a device path.
UnpackDevicePath	Reports membership of a single-instance device path in a possible multiple-instance device path.
LibMatchDevicePaths	Creates a second corresponding instance of a given device path.
LibDuplicateDevicePathInstance	Creates a second corresponding instance of a given device path.

D.11 PCI Functions and Macros

Table D-11 lists some helper functions that are related to PCI devices and a set of functions and macros that are used to access PCI I/O and PCI configuration space.

Table D-11.PCI Functions and Macros

Name	Description
PCIFindDeviceClass	Finds a PCI device that matches the PCI base class and subclass.
PCIFindDevice	Finds a PCI device that matches the PCI device ID and vendor ID.
InitializeGlobalIoDevice	Retrieves the Device I/O Protocol instance for a given device.
ReadPort	Reads an I/O port.
WritePort	Writes to an I/O port.
ReadPciConfig	Reads an I/O port.
WritePciConfig	Writes to an I/O port.
Inp	Reads an 8-bit value from an I/O port.
Outp	Writes an 8-bit value to an I/O port.
Inpw	Reads a 16-bit value from an I/O port.
Outpw	Writes a 16-bit value to an I/O port.
Inpd	Reads a 32-bit value from an I/O port.
Outpw	Writes a 32-bit value to an I/O port.
readpci8	Reads an 8-bit value from PCI configuration space.
writepci8	Writes an 8-bit value to PCI configuration space.
readpci16	Reads a 16-bit value from PCI configuration space.
writepci16	Writes a 16-bit value to PCI configuration space.
readpci32	Reads a 32-bit value from PCI configuration space.
writepci32	Writes a 32-bit value to PCI configuration space.

D.12 Miscellaneous Functions and Macros

Table D-12 lists some miscellaneous helper functions that are included in the EFI Library.

Table D-12.Miscellaneous Functions and Macros

Name	Description
LibGetVariable	Retrieves an environment variable's value.
LibGetVariableAndSIze	Retrieves an environment variable's value and its size in bytes.
LibDeleteVariable	Removes a given variable from the variable store
CompareGuid	Compares two 128-bit GUIDs.
CR	Returns a pointer to an element's containing record.
DecimaltoBCD	Converts a decimal value to a Binary Coded Decimal (BCD) value.
BCDtoDecimal	Converts a BCD value to a decimal value.
LibCreateProtocolNotifyEvent	Creates a notification event that fires every time a protocol instance is created.
WaitForSingleEvent	Waits for an event to fire or a timeout to expire.
WaitForEventWithTimeout	Waits for either a **`SIMPLE_INPUT`** event or a timeout to occur.
LibEnableVirtualMappings	Converts internal library pointers to virtual runtime pointers.
ConvertList	Converts pointers in a linked list to virtual runtime pointers.
LibGetSystemConfigurationTable	Retrieves a system configuration table from the EFI System Table.

Appendix E
Sample OS Loader Listing

```
/*++

Copyright (c) 1998  Intel Corporation
Module Name:
    osloader.c
Abstract:
Author:
Revision History

--*/

#include "efi.h"
#include "efilib.h"

static CHAR16  *OsLoaderMemoryTypeDesc[EfiMaxMemoryType]  = {
            L"reserved  ",
            L"LoaderCode",
            L"LoaderData",
            L"BS_code   ",
            L"BS_data   ",
            L"RT_code   ",
            L"RT_data   ",
            L"available ",
            L"Unusable  ",
            L"ACPI_recl ",
            L"ACPI_NVS  ",
            L"MemMapIO  ",
            L"MemPortIO ",
            L"PAL_code  "
    };

EFI_STATUS
InitializeOSLoader (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
    )
{
    EFI_STATUS              Status;
    UINT16                  InputString[20];

    EFI_TIME                Time;

    CHAR16                  *DevicePathAsString;

    EFI_LOADED_IMAGE        *LoadedImage;
    EFI_DEVICE_PATH         *DevicePath;

    EFI_FILE_IO_INTERFACE   *Vol;
    EFI_FILE_HANDLE         RootFs;
    EFI_FILE_HANDLE         CurDir;
    EFI_FILE_HANDLE         FileHandle;
    CHAR16                  FileName[100];
    UINTN                   i;
    UINTN                   Size;
    VOID                    *OsKernelBuffer;

    UINTN                   NoHandles;
    EFI_HANDLE              *HandleBuffer;

    EFI_BLOCK_IO            *BlkIo;
    EFI_BLOCK_IO_MEDIA      *Media;
    UINT8                   *Block;
    UINT32                  MediaId;

    VOID                    *AcpiTable              = NULL;
    VOID                    *SMBIOSTable            = NULL;
    VOID                    *SalSystemTable         = NULL;
    VOID                    *MpsTable               = NULL;

    EFI_MEMORY_DESCRIPTOR   *MemoryMap;
    EFI_MEMORY_DESCRIPTOR   *MemoryMapEntry;
    UINTN                   NoEntries;
    UINTN                   MapKey;
    UINTN                   DescriptorSize;
    UINT32                  DescriptorVersion;

    UINTN                   VariableNameSize;
    CHAR16                  VariableName[256];
    EFI_GUID                VendorGuid;
    UINT8                   *VariableValue;

    //
    // Initialize the Library. Set BS, RT, & ST globals
    //  BS = Boot Services RT = Runtime Services
    //  ST = System Table
    //

    InitializeLib (ImageHandle, SystemTable);

    //
    // Print a message to the console output device.
    //

    Print(L"OS Loader application started\n");

    //
    // Print Date and Time
    //

    Status = RT->GetTime(&Time,NULL);

    if (!EFI_ERROR(Status)) {
        Print(L"Date : %02d/%02d/%04d  Time : %02d:%02d:%02d\n",
              Time.Month,Time.Day,Time.Year,Time.Hour,Time.Minute,Time.Second);
    }

    //
    // Get the device handle and file path to the EFI OS Loader itself.
    //

    Status = BS->HandleProtocol (ImageHandle,
                                 &LoadedImageProtocol,
                                 &LoadedImage
                                 );

    if (EFI_ERROR(Status)) {
        Print(L"Can not retrieve a LoadedImageProtocol handle for ImageHandle\n");
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    Status = BS->HandleProtocol (LoadedImage->DeviceHandle,
                                 &DevicePathProtocol,
                                 &DevicePath
                                 );

    if (EFI_ERROR(Status) || DevicePath==NULL) {
        Print(L"Cannot find a DevicePath handle for LoadedImage->DeviceHandle\n");
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    DevicePathAsString = DevicePathToStr(DevicePath);
    if (DevicePathAsString != NULL) {
        Print (L"Image device : %s\n", DevicePathAsString);
        FreePool(DevicePathAsString);
    }

    DevicePathAsString = DevicePathToStr(LoadedImage->FilePath);
    if (DevicePathAsString != NULL) {
        Print (L"Image file   : %s\n", DevicePathToStr (LoadedImage->FilePath));
        FreePool(DevicePathAsString);
    }

    Print (L"Image Base   : %X\n", LoadedImage->ImageBase);
    Print (L"Image Size   : %X\n", LoadedImage->ImageSize);

    //
    // Open the volume for the device where the EFI OS Loader was loaded from.
    //

    Status = BS->HandleProtocol (LoadedImage->DeviceHandle,
                                 &FileSystemProtocol,
                                 &Vol
                                 );

    if (EFI_ERROR(Status)) {
        Print(L"Cannot get a FileSystem handle for LoadedImage->DeviceHandle\n");
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    Status = Vol->OpenVolume (Vol, &RootFs);

    if (EFI_ERROR(Status)) {
        Print(L"Cannot open the volume for the file system\n");
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    CurDir = RootFs;

    //
    // Open the file OSKERNEL.BIN in the same path as the EFI OS Loader.
    //
```

```
DevicePathAsString = DevicePathToStr(LoadedImage->FilePath);
if (DevicePathAsString!=NULL) {
    StrCpy(FileName,DevicePathAsString);
    FreePool(DevicePathAsString);
}
for(i=StrLen(FileName);i>0 && FileName[i]!='\\';i--);
FileName[i] = 0;
StrCat(FileName,L"\\OSKERNEL.BIN");

Status = CurDir->Open (CurDir,
                       &FileHandle,
                       FileName,
                       EFI_FILE_MODE_READ,
                       0
                       );

if (EFI_ERROR(Status)) {
    Print(L"Can not open the file %s\n",FileName);
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
}

Print(L"Opened %s\n",FileName);

//
// Allocate a 1 MB buffer for OSKERNEL.BIN
//

Size = 0x00100000;
BS->AllocatePool(EfiLoaderData,
                 Size,
                 &OsKernelBuffer
                 );

if (OsKernelBuffer == NULL) {
    Print(L"Can not allocate a buffer for the file %s\n",FileName);
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
}

//
// Load OSKERNEL.BIN into the allocated memory.
//

Status = FileHandle->Read(FileHandle,
                          &Size,
                          OsKernelBuffer
                          );

if (EFI_ERROR(Status)) {
    Print(L"Can not read the file %s\n",FileName);
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
}

Print(L"%X bytes of OSKERNEL.BIN read into memory at %X\n",Size,OsKernelBuffer);

//
// Close OSKERNEL.BIN
//

Status = FileHandle->Close(FileHandle);
if (EFI_ERROR(Status)) {
    Print(L"Can not close the file %s\n",FileName);
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
}

//
// Free the resources allocated from pool.
//

FreePool(OsKernelBuffer);

Print(L"\nPress [ENTER] to continue...\n");
Input(NULL,InputString,20);

//
// Get a list of all the BLOCK_IO devices
//

NoHandles = 0;
HandleBuffer = NULL;
Status = LibLocateHandle(ByProtocol,
                         &BlockIoProtocol,
                         NULL,
                         &NoHandles,
                         &HandleBuffer
                         );
if (EFI_ERROR(Status)) {
    Print(L"Can not get the array of BLOCK_IO handles\n");
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
}

//
// Read the first block from each BLOCK_IO device and display it
//

for(i=0;i<NoHandles;i++) {

    //
    // Get the DEVICE_PATH Protocol Interface to the device
    //

    Status = BS->HandleProtocol (HandleBuffer[i],
                                 &DevicePathProtocol,
                                 &DevicePath
                                 );

    if (EFI_ERROR(Status) || DevicePath==NULL) {
        Print(L"Can not get a DevicePath handle for HandleBuffer[%d]\n",i);
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    //
    // Get the BLOCK_IO Protocol Interface to the device
    //

    Status = BS->HandleProtocol (HandleBuffer[i],
                                 &BlockIoProtocol,
                                 &BlkIo
                                 );

    if (EFI_ERROR(Status) || BlkIo==NULL) {
        Print(L"Handle does not support the BLOCK_IO protocol\n");
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    //
    // Allocate a buffer for the first block on the device.
    //

    Media = BlkIo->Media;

    Block = AllocatePool (Media->BlockSize);
    if (Block == NULL) {
        Print(L"Can not allocate buffer for a block\n");
        BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
    }

    //
    // Read block #0 from the device.
    //

    MediaId = Media->MediaId;

    Status = BlkIo->ReadBlocks(BlkIo,
                               MediaId,
                               (EFI_LBA)0,
                               Media->BlockSize,
                               Block);

    //
    // Display block#0 from the device.
    //

    DevicePathAsString = DevicePathToStr(DevicePath);
    if (DevicePathAsString != NULL) {
        Print(L"\nBlock #0 of device %s\n",DevicePathAsString);
        FreePool(DevicePathAsString);
    }
    DumpHex(0,0,Media->BlockSize,Block);

    //
    // Free the resources allocated from pool.
    //

    FreePool(Block);

    Print(L"\nPress [ENTER] to continue...\n");
    Input(NULL,InputString,20);
}

//
// Free the resources allocated from pool.
//

FreePool(HandleBuffer);

//
// Get System Configuration
//

Print(L"System Configuration Tables:\n\n");
```

```
Status = LibGetSystemConfigurationTable(&AcpiTableGuid,&AcpiTable);
if (!EFI_ERROR(Status)) {
    Print(L"  ACPI Table is at address            : %X\n",AcpiTable);
}
Status = LibGetSystemConfigurationTable(&SMBIOSTableGuid,&SMBIOSTable);
if (!EFI_ERROR(Status)) {
    Print(L"  SMBIOS Table is at address          : %X\n",SMBIOSTable);
}
Status = LibGetSystemConfigurationTable(&SalSystemTableGuid,&SalSystemTable);
if (!EFI_ERROR(Status)) {
    Print(L"  Sal System Table is at address      : %X\n",SalSystemTable);
}
Status = LibGetSystemConfigurationTable(&MpsTableGuid,&MpsTable);
if (!EFI_ERROR(Status)) {
    Print(L"  MPS Table is at address             : %X\n",MpsTable);
}

Print(L"\nPress [ENTER] to continue...\n");
Input(NULL,InputString,20);

//
// Display the current Memory Map
//

MemoryMap = LibMemoryMap(&NoEntries,&MapKey,&DescriptorSize,&DescriptorVersion);
if (MemoryMap == NULL) {
    Print(L"Cannot retrieve the current memory map\n");
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
}

Print(L"Memory Descriptor List:\n\n");
Print(L"  Type        Start Address     End Address       Attributes      \n");
Print(L"  ==========  ================  ================  ================\n");
MemoryMapEntry = MemoryMap;
for(i=0;i<NoEntries;i++) {
    Print(L"  %s  %lX  %lX  %lX\n",
          OsLoaderMemoryTypeDesc[MemoryMapEntry->Type],
          MemoryMapEntry->PhysicalStart,
          MemoryMapEntry->PhysicalStart +
              LShiftU64(MemoryMapEntry->NumberOfPages,PAGE_SHIFT)-1,
          MemoryMapEntry->Attribute);
    MemoryMapEntry = NextMemoryDescriptor(MemoryMapEntry, DescriptorSize);
}

//
// Free the resources allocated from pool.
//

FreePool(MemoryMap);

Print(L"\nPress [ENTER] to continue...\n");
Input(NULL,InputString,20);

//
// Display all the Environment Variables
//

Print(L"Environment Variable List:\n\n");
```

```
    VariableName[0] = 0x0000;
    VendorGuid = NullGuid;
    Print(L"GUID                                Variable Name        Value\n");
    Print(L"=================================== ==================== ========\n");
    do {
        VariableNameSize = 256;
        Status = RT->GetNextVariableName(&VariableNameSize,
                                         VariableName,
                                         &VendorGuid
                                         );

        if (Status == EFI_SUCCESS) {
            VariableValue = LibGetVariable(VariableName,&VendorGuid);
            if (VariableValue != NULL) {
                Print(L"%.-35g %.-20s %X\n",&VendorGuid,VariableName,VariableValue);
                FreePool(VariableValue);
            }
        }
    } while (Status == EFI_SUCCESS);

    Print(L"\nPress [ENTER] to continue...\n");
    Input(NULL,InputString,20);

    //
    // Get the most current memory map.
    //

    MemoryMap = LibMemoryMap(&NoEntries,&MapKey,&DescriptorSize,&DescriptorVersion);

    //
    // Transition from Boot Services to Runtime Services.
    //

    Print(L"Call ExitBootServices()\n");
//    BS->ExitBootServices(ImageHandle,MapKey);

    Print(L"\nPress [ENTER] to continue...\n");
    Input(NULL,InputString,20);

    return EFI_SUCCESS;
}
```

Appendix F
Sample OS Loader Output

```
OS Loader application started
Date : 07/14/1999  Time : 14:15:45
Image device : Acpi(PNP0A03,0)/Pci(0|0)/Acpi(PNP0604,0)
Image file   : \osloader.efi
Image Base   : 07E68000
Image Size   : 00008FFF
Opened \OSKERNEL.BIN
00009000 bytes of OSKERNEL.BIN read into memory at 07D67008

Press [ENTER] to continue...

Block #0 of device
00000000: EB 3C 90 49 4E 54 45 4C-20 20 20 00 02 01 01 00  *.<.INTEL   .....*
00000010: 02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00  *...@............*
00000020: 00 00 00 00 00 01 29 20-20 20 20 45 46 49 20 46  *......)    EFI F*
00000030: 4C 4F 50 50 59 20 46 41-54 31 32 20 20 20 8C C8  *LOPPY FAT12   ..*
00000040: 8E D0 05 00 10 8E D8 BC-00 7C 8B EC B4 08 B2 00  *................*
00000050: CD 13 33 C0 8A C6 24 3F-FE C0 50 8A C1 24 3F 50  *..3...$?..P..$?P*
00000060: 81 BE FE 01 55 AA 0F 85-6F 01 80 BE 00 00 EB 0F  *....U...o.......*
00000070: 85 66 01 81 BE 0B 00 00-02 0F 85 5C 01 80 BE 0D  *.f.........\....*
00000080: 00 00 0F 84 53 01 83 BE-11 00 00 0F 84 4A 01 8B  *....S........J..*
00000090: 86 0E 00 3B 86 13 00 0F-8F 3E 01 33 C9 8D 36 03  *...;.....>.3..6.*
000000A0: 00 B1 08 E8 25 01 8D 36-2B 00 B1 0B E8 1C 01 81  *....%..6+.......*
000000B0: BE 36 00 46 41 0F 85 20-01 81 BE 38 00 54 31 0F  *.6.FA.. ...8.T1.*
000000C0: 85 16 01 B8 20 20 39 86-3B 00 0F 85 0B 01 39 86  *....  9.;.....9.*
000000D0: 3C 00 0F 85 03 01 80 BE-3A 00 32 74 0C 80 BE 3A  *<.......:.2t...:*
000000E0: 00 36 0F 84 F3 00 E9 F0-00 8B 8E 11 00 C1 E1 05  *.6..............*
000000F0: 8B D9 81 E3 FF 01 0F 85-DF 00 8B D9 C1 EB 09 BF  *................*
00000100: 00 00 8A 86 10 00 32 E4-F7 A6 16 00 03 86 0E 00  *......2.........*
00000110: 1E 07 E8 6D 00 03 C3 89-46 00 81 3D 45 46 75 27  *...m....F..=EFu'*
00000120: 81 7D 02 49 4C 75 20 81-7D 04 44 52 75 19 B8 20  *...ILu ...DRu.. *
00000130: 20 39 45 06 75 11 39 45-08 75 0C 39 45 09 75 07  * 9E.u.9E.u.9E.u.*
00000140: 8A 45 0B 24 58 74 0B 83-C7 20 83 E9 20 75 CB E9  *.E.$Xt... .. u..*
00000150: 87 00 8B 4D 1A 8C C8 05-00 20 8E C0 33 FF 8B C1  *...M..... ..3...*
00000160: 83 E8 02 32 FF 8A 9E 0D-00 F7 E3 03 46 00 32 FF  *...2........F.2.*
00000170: 8A 9E 0D 00 06 E8 0A 00-58 89 86 80 01 EA 00 00  *........X.......*
00000180: 00 20 60 8B F0 8B CB BD-FC 7B 8B C6 33 D2 F7 76  *. `.........3..v*
00000190: 00 42 8B 5E 00 2B DA 43-3B CB 7F 02 8B D9 51 8A  *.B.^.+.C;.....Q.*
000001A0: CA 33 D2 F7 76 02 53 8A-F2 B2 00 8A E8 8A C3 B4  *.3..v.S.........*
000001B0: 02 8B DF CD 13 5B 59 03-F3 2B CB 8C C0 C1 E3 05  *.....[Y..+......*
000001C0: 03 C3 8E C0 83 F9 00 75-BE 61 C3 80 3A 7F 7F 09  *.......u.a..:...*
000001D0: 80 3A 20 7C 04 46 E2 F3-C3 CC EB FD 00 00 00 00  *.: ..F..........*
000001E0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
000001F0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA  *..............U.*

Press [ENTER] to continue...

Block #0 of device
00000000: 33 C0 8E D0 BC 00 7C FB-50 07 50 1F FC BE 1B 7C  *3.......P.P.....*
00000010: BF 1B 06 50 57 B9 E5 01-F3 A4 CB BE BE 07 B1 04  *...PW...........*
00000020: 38 2C 7C 09 75 15 83 C6-10 E2 F5 CD 18 8B 14 8B  *8,..u...........*
00000030: EE 83 C6 10 49 74 16 38-2C 74 F6 BE 10 07 4E AC  *....It.8,t....N.*
00000040: 3C 00 74 FA BB 07 00 B4-0E CD 10 EB F2 89 46 25  *<.t...........F%*
```

```
00000050: 96 8A 46 04 B4 06 3C 0E-74 11 B4 0B 3C 0C 74 05  *..F...<.t...<.t.*
00000060: 3A C4 75 2B 40 C6 46 25-06 75 24 BB AA 55 50 B4  *:.u+@.F%.u$..UP.*
00000070: 41 CD 13 58 72 16 81 FB-55 AA 75 10 F6 C1 01 74  *A..Xr...U.u....t*
00000080: 0B 8A E0 88 56 24 C7 06-A1 06 EB 1E 88 66 04 BF  *....V$.......f..*
00000090: 0A 00 B8 01 02 8B DC 33-C9 83 FF 05 7F 03 8B 4E  *.......3.......N*
000000A0: 25 03 4E 02 CD 13 72 29-BE 75 07 81 3E FE 7D 55  *%.N...r).u..>..U*
000000B0: AA 74 5A 83 EF 05 7F DA-85 F6 75 83 BE 3F 07 EB  *.tZ.......u..?..*
000000C0: 8A 98 91 52 99 03 46 08-13 56 0A E8 12 00 5A EB  *...R..F..V....Z.*
000000D0: D5 4F 74 E4 33 C0 CD 13-EB B8 00 00 80 01 44 16  *.Ot.3.........D.*
000000E0: 56 33 F6 56 56 52 50 06-53 51 BE 10 00 56 8B F4  *V3.VVRP.SQ...V..*
000000F0: 50 52 B8 00 42 8A 56 24-CD 13 5A 58 8D 64 10 72  *PR..B.V$..ZX.d.r*
00000100: 0A 40 75 01 42 80 C7 02-E2 F7 F8 5E C3 EB 74 49  *.@u.B......^..tI*
00000110: 6E 76 61 6C 69 64 20 70-61 72 74 69 74 69 6F 6E  *nvalid partition*
00000120: 20 74 61 62 6C 65 2E 20-53 65 74 75 70 20 63 61  * table. Setup ca*
00000130: 6E 6E 6F 74 20 63 6F 6E-74 69 6E 75 65 2E 00 45  *nnot continue..E*
00000140: 72 72 6F 72 20 6C 6F 61-64 69 6E 67 20 6F 70 65  *rror loading ope*
00000150: 72 61 74 69 6E 67 20 73-79 73 74 65 6D 2E 20 53  *rating system. S*
00000160: 65 74 75 70 20 63 61 6E-6E 6F 74 20 63 6F 6E 74  *etup cannot cont*
00000170: 69 6E 75 65 2E 00 00 00-00 00 00 00 00 00 00 00  *inue............*
00000180: 00 00 00 8B FC 1E 57 8B-F5 CB 00 00 00 00 00 00  *......W.........*
00000190: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
000001A0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
000001B0: 00 00 00 00 00 00 00 00-A3 F4 00 00 00 00 80 01  *................*
000001C0: 01 00 0B 7F 7F EF 3F 00-00 00 C1 07 3D 00 00 00  *......?.....=...*
000001D0: 41 F0 05 7F FF DF 00 08-3D 00 00 08 3D 00 00 00  *A.......=...=...*
000001E0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
000001F0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA  *..............U.*

Press [ENTER] to continue...

Block #0 of device
00000000: EB 58 90 4D 53 57 49 4E-34 2E 31 00 02 08 20 00  *.X.MSWIN4.1... .*
00000010: 02 00 00 00 00 F8 00 00-3F 00 80 00 3F 00 00 00  *........?...?...*
00000020: C1 07 3D 00 3F 0F 00 00-00 00 00 00 02 00 00 00  *..=.?...........*
00000030: 01 00 06 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
00000040: 80 00 29 EB 17 22 1D 4E-4F 20 4E 41 4D 45 20 20  *..)..".NO NAME  *
00000050: 20 20 46 41 54 33 32 20-20 20 33 C9 8E D1 BC F4  *  FAT32   3.....*
00000060: 7B 8E C1 8E D9 BD 00 7C-88 4E 02 8A 56 40 B4 08  *.........N..V@..*
00000070: CD 13 73 05 B9 FF FF 8A-F1 66 0F B6 C6 40 66 0F  *..s......f...@f.*
00000080: B6 D1 80 E2 3F F7 E2 86-CD C0 ED 06 41 66 0F B7  *....?.......Af..*
00000090: C9 66 F7 E1 66 89 46 F8-83 7E 16 00 75 38 83 7E  *.f..f.F.....u8..*
000000A0: 2A 00 77 32 66 8B 46 1C-66 83 C0 0C BB 00 80 B9  **.w2f.F.f.......*
000000B0: 01 00 E8 2B 00 E9 48 03-A0 FA 7D B4 7D 8B F0 AC  *...+..H.........*
000000C0: 84 C0 74 17 3C FF 74 09-B4 0E BB 07 00 CD 10 EB  *..t.<.t.........*
000000D0: EE A0 FB 7D EB E5 A0 F9-7D EB E0 98 CD 16 CD 19  *................*
000000E0: 66 60 66 3B 46 F8 0F 82-4A 00 66 6A 00 66 50 06  *f`f;F...J.fj.fP.*
000000F0: 53 66 68 10 00 01 00 80-7E 02 00 0F 85 20 00 B4  *Sfh.......... ..*
00000100: 41 BB AA 55 8A 56 40 CD-13 0F 82 1C 00 81 FB 55  *A..U.V@........U*
00000110: AA 0F 85 14 00 F6 C1 01-0F 84 0D 00 FE 46 02 B4  *.............F..*
00000120: 42 8A 56 40 8B F4 CD 13-B0 F9 66 58 66 58 66 58  *B.V@......fXfXfX*
00000130: 66 58 EB 2A 66 33 D2 66-0F B7 4E 18 66 F7 F1 FE  *fX.*f3.f..N.f...*
00000140: C2 8A CA 66 8B D0 66 C1-EA 10 F7 76 1A 86 D6 8A  *...f..f....v....*
00000150: 56 40 8A E8 C0 E4 06 0A-CC B8 01 02 CD 13 66 61  *V@............fa*
00000160: 0F 82 54 FF 81 C3 00 02-66 40 49 0F 85 71 FF C3  *..T.....f@I..q..*
00000170: 4E 54 4C 44 52 20 20 20-20 20 20 0D 0A 4E 54 4C  *NTLDR      ..NTL*
00000180: 44 52 20 69 73 20 6D 69-73 73 69 6E 67 FF 0D 0A  *DR is missing...*
00000190: 44 69 73 6B 20 65 72 72-6F 72 FF 0D 0A 50 72 65  *Disk error...Pre*
000001A0: 73 73 20 61 6E 79 20 6B-65 79 20 74 6F 20 72 65  *ss any key to re*
000001B0: 73 74 61 72 74 0D 0A 00-00 00 00 00 00 00 00 00  *start...........*
000001C0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
```

```
000001D0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
000001E0: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
000001F0: 00 00 00 00 00 00 00 00-00 7B 8E 9B 00 00 55 AA  *..............U.*

Press [ENTER] to continue...

Block #0 of device
00000000: EB 3C 90 4D 53 57 49 4E-34 2E 31 00 02 04 01 00  *.<.MSWIN4.1.....*
00000010: 02 00 02 00 00 F8 67 00-3F 00 80 00 3F 00 00 00  *......g.?...?...*
00000020: 41 99 01 00 80 00 29 E3-18 61 2D 4E 4F 20 4E 41  *A.....)..a-NO NA*
00000030: 4D 45 20 20 20 20 46 41-54 31 36 20 20 20 33 C9  *ME    FAT16   3.*
00000040: 8E D1 BC FC 7B 16 07 BD-78 00 C5 76 00 1E 56 16  *........x..v..V.*
00000050: 55 BF 22 05 89 7E 00 89-4E 02 B1 0B FC F3 A4 06  *U.".....N.......*
00000060: 1F BD 00 7C C6 45 FE 0F-38 4E 24 7D 20 8B C1 99  *.....E..8N$. ...*
00000070: E8 7E 01 83 EB 3A 66 A1-1C 7C 66 3B 07 8A 57 FC  *.....:f...f;..W.*
00000080: 75 06 80 CA 02 88 56 02-80 C3 10 73 ED 33 C9 FE  *u.....V....s.3..*
00000090: 06 D8 7D 8A 46 10 98 F7-66 16 03 46 1C 13 56 1E  *....F...f..F..V.*
000000A0: 03 46 0E 13 D1 8B 76 11-60 89 46 FC 89 56 FE B8  *.F....v.`.F..V..*
000000B0: 20 00 F7 E6 8B 5E 0B 03-C3 48 F7 F3 01 46 FC 11  * ....^...H...F..*
000000C0: 4E FE 61 BF 00 07 E8 28-01 72 3E 38 2D 74 17 60  *N.a....(.r>8-t.`*
000000D0: B1 0B BE D8 7D F3 A6 61-74 3D 4E 74 09 83 C7 20  *.......at=Nt... *
000000E0: 3B FB 72 E7 EB DD FE 0E-D8 7D 7B A7 BE 7F 7D AC  *;.r.............*
000000F0: 98 03 F0 AC 98 40 74 0C-48 74 13 B4 0E BB 07 00  *.....@t.Ht......*
00000100: CD 10 EB EF BE 82 7D EB-E6 BE 80 7D EB E1 CD 16  *................*
00000110: 5E 1F 66 8F 04 CD 19 BE-81 7D 8B 7D 1A 8D 45 FE  *^.f...........E.*
00000120: 8A 4E 0D F7 E1 03 46 FC-13 56 FE B1 04 E8 C2 00  *.N....F..V......*
00000130: 72 D7 EA 00 02 70 00 52-50 06 53 6A 01 6A 10 91  *r....p.RP.Sj.j..*
00000140: 8B 46 18 A2 26 05 96 92-33 D2 F7 F6 91 F7 F6 42  *.F..&...3......B*
00000150: 87 CA F7 76 1A 8A F2 8A-E8 C0 CC 02 0A CC B8 01  *...v............*
00000160: 02 80 7E 02 0E 75 04 B4-42 8B F4 8A 56 24 CD 13  *.....u..B...V$..*
00000170: 61 61 72 0A 40 75 01 42-03 5E 0B 49 75 77 C3 03  *aar.@u.B.^.Iuw..*
00000180: 18 01 27 0D 0A 49 6E 76-61 6C 69 64 20 73 79 73  *..'..Invalid sys*
00000190: 74 65 6D 20 64 69 73 6B-FF 0D 0A 44 69 73 6B 20  *tem disk...Disk *
000001A0: 49 2F 4F 20 65 72 72 6F-72 FF 0D 0A 52 65 70 6C  *I/O error...Repl*
000001B0: 61 63 65 20 74 68 65 20-64 69 73 6B 2C 20 61 6E  *ace the disk, an*
000001C0: 64 20 74 68 65 6E 20 70-72 65 73 73 20 61 6E 79  *d then press any*
000001D0: 20 6B 65 79 0D 0A 00 00-49 4F 20 20 20 20 20 20  * key....IO      *
000001E0: 53 59 53 4D 53 44 4F 53-20 20 20 53 59 53 7F 01  *SYSMSDOS   SYS..*
000001F0: 00 41 BB 00 07 60 66 6A-00 E9 3B FF 00 00 55 AA  *.A...`fj..;...U.*

Press [ENTER] to continue...

System Configuration Tables:

  ACPI Table is at address                    : 00000000
  SMBIOS Table is at address                  : 00000000
  MPS Table is at address                     : 00000000

Press [ENTER] to continue...

Memory Descriptor List:

  Type        Start Address     End Address       Attributes
  ==========  ================  ================  ================
  BS_data     0000000000000000  0000000000000FFF  0000000000000008
  available   0000000000001000  000000000001FFFF  0000000000000008
  BS_data     0000000000020000  0000000000021FFF  0000000000000008
  available   0000000000022000  0000000000089FFF  0000000000000008
  BS_data     000000000008A000  000000000009AFFF  0000000000000008
  BS_code     000000000009B000  000000000009EFFF  0000000000000008
  BS_data     000000000009F000  000000000009FFFF  0000000000000008
  BS_data     00000000000F0000  00000000000FFFFF  0000000000000008
  available   0000000000100000  00000000001DFFFF  0000000000000008
  BS_data     00000000001E0000  00000000001FFFFF  0000000000000008
  available   0000000000200000  0000000000200FFF  0000000000000008
  BS_code     0000000000201000  0000000000215FFF  0000000000000008
  RT_code     0000000000216000  0000000000218FFF  8000000000000008
  BS_data     0000000000219000  000000000021BFFF  0000000000000008
  RT_data     000000000021C000  000000000021CFFF  8000000000000008
  BS_data     000000000021D000  000000000021DFFF  0000000000000008
  available   000000000021E000  0000000007E67FFF  0000000000000008
  LoaderData  0000000007E68000  0000000007E68FFF  0000000000000008
  LoaderCode  0000000007E69000  0000000007E6CFFF  0000000000000008
  LoaderData  0000000007E6D000  0000000007E74FFF  0000000000000008
  available   0000000007E75000  0000000007E7AFFF  0000000000000008
  LoaderData  0000000007E7B000  0000000007E7BFFF  0000000000000008
  BS_data     0000000007E7C000  0000000007E85FFF  0000000000000008
  RT_data     0000000007E86000  0000000007E86FFF  8000000000000008
  BS_data     0000000007E87000  0000000007E87FFF  0000000000000008
  BS_code     0000000007E88000  0000000007E93FFF  0000000000000008
  BS_data     0000000007E94000  0000000007E99FFF  0000000000000008
  LoaderData  0000000007E9A000  0000000007E9BFFF  0000000000000008
  BS_data     0000000007E9C000  0000000007E9CFFF  0000000000000008
  LoaderData  0000000007E9D000  0000000007E9DFFF  0000000000000008
  LoaderCode  0000000007E9E000  0000000007EA1FFF  0000000000000008
  LoaderData  0000000007EA2000  0000000007EA4FFF  0000000000000008
  BS_data     0000000007EA5000  0000000007FC6FFF  0000000000000008
  RT_data     0000000007FC7000  0000000007FC8FFF  8000000000000008
  BS_data     0000000007FC9000  0000000007FCEFFF  0000000000000008
  RT_data     0000000007FCF000  0000000007FCFFFF  8000000000000008
  BS_data     0000000007FD0000  0000000007FFFFFF  0000000000000008
  BS_data     00000000FFFE0000  00000000FFFFFFFF  0000000000000008

Press [ENTER] to continue...

Environment Variable List:

GUID                                 Variable Name        Value
==================================== ==================== ========
Efi                                  Lang                 07E9AF08
Efi                                  Timeout              07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 DevIo                07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 fs                   07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 diskio               07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 blkio                07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 txtin                07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 txtout               07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 load                 07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 image                07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 varstore             07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 dpath                07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 unicode              07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 pxe                  07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 ShellInt             07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 SEnv                 07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 ShellProtId          07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 ShellDevPathMap      07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 ShellAlias           07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 G0                   07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 Efi                  07E9AF08
```

```
47C7B226-C42A-11D2-8E57-00A0C969723 GenFileInfo         07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 FileSysInfo         07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 PcAnsi              07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 Unknown Device      07E9AF08
47C7B227-C42A-11D2-8E57-00A0C969723 dir                 07E9AF08
47C7B227-C42A-11D2-8E57-00A0C969723 md                  07E9AF08
47C7B227-C42A-11D2-8E57-00A0C969723 rd                  07E9AF08
47C7B227-C42A-11D2-8E57-00A0C969723 del                 07E9AF08
47C7B227-C42A-11D2-8E57-00A0C969723 copy                07E9AF08
47C7B226-C42A-11D2-8E57-00A0C969723 LegacyBoot          07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 fs0                 07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 fs1                 07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 fs2                 07E7BF08
47C7B225-C42A-11D2-8E57-00A0C969723 fs3                 07E7BF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk0                07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk1                07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk2                07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk3                07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk4                07E7BF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk5                07E7BF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk6                07E7BF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk7                07E9AF08
47C7B225-C42A-11D2-8E57-00A0C969723 blk8                07E9AF08
47C7B224-C42A-11D2-8E57-00A0C969723 path                07E7BF08
Efi                                 LangCodes           07E9AF08

Press [ENTER] to continue...

Call ExitBootServices()

Press [ENTER] to continue...
```

Appendix G Glossary

ACPI	Advanced Configuration and Power Interface.
API	Application programming interface.
ARP	Address Resolution Protocol.
ASCII	American Standard Code for Information Interchange.
BAR	Base Address Register.
BCD	Binary Coded Decimal.
BIOS	Basic input/output system.
BIS	Boot Integrity Services.
BS	Boot Services.
CD-ROM	Compact Disk – Read-Only Memory.
CRC	Cyclic Redundancy Check.
DHCP	Dynamic Host Configuration Protocol.
DMA	Direct Memory Access.
D.O.R.A.	Discover / offer / request / acknowledge.
EBC	EFI Byte Code.
EFI	Extensible Firmware Interface.
FAT	File allocation table.
FIFO	First In First Out.
FW	Firmware.
GUID	Globally Unique Identifier.
IA-32	32-bit Intel® architecture.
ID	Identifier.
IP	Internet Protocol.
IPv4	Internet Protocol Version 4.
IPv6	Internet Protocol Version 6.
I/O	Input/output.
Lib	Library.
LUN	Logical Unit Number.
MAC	Media Access Controller.

MBR	Master Boot Record.
MTFTP	Multicast TFTP.
NVRAM	Nonvolatile RAM.
OEM	Original equipment manufacturer.
OS	Operating system.
PCI	Peripheral Component Interconnect.
P-code	Pseudo code.
PE	Portable Executable.
PE/COFF	PE32, PE32+, or Common Object File Format.
PXE	Preboot Execution Environment.
RAM	Random Access Memory.
ROM	Read-Only Memory.
RT	Runtime Services.
RTC	Real Time Clock.
SAL	System Abstraction Layer.
S.A.R.R.	Solicit / advertise / request / reply.
SCSI	Small Computer System Interface.
SDK	Software Development Kit.
SMBIOS	System Management BIOS.
ST	System Table.
TFTP	Trivial File Transfer Protocol.
TPL	Task Priority Level.
UDP	User Datagram Protocol.
USB	Universal Serial Bus.
UGA	Universal Graphics Adapter.