Extensible Firmware Interface Library Specification

Draft for Review

Version 1.10.1

January 5, 2004

Revision History

Revision	Revision History	Date	Author
0.1	Initial review draft	3/24/99	Intel
0.2	Updated to match 0.91.007 Sample Implementation	7/14/99	Intel
0.9	Updated to match 0.91.009 Sample Implementation	11/17/99	Intel
0.99	Updated to match 0.99.12.20 Sample Implementation	4/24/00	Intel
1.1	Updated to match the 1.1 Sample Implementation Release Changed headers/footers.	7/31/01	Intel
1.10	Updated to match the EFI 1.10.14.54 Sample Implementation that matches the 0.9 draft of the EFI 1.10 Specification	12/16/01	Intel
1.10.1	Updated to match the EFI 1.10.14.62 Application Toolkit that matches the final version of the EFI 1.10 Specification.	1/5/04	Intel

intel

Contents

1 Introduction 11
- 1.1 Overview 11
- 1.2 Organization of this Document 11
- 1.3 Goals 12
- 1.4 Target Audience 12
- 1.5 Prerequisite Specifications 12
- 1.6 Conventions Used in This Document 12
  - 1.6.1 Data Structure Descriptions 13
  - 1.6.2 Procedure Descriptions 14
  - 1.6.3 Pseudo-Code Conventions 14
  - 1.6.4 Typographic Conventions 15

2 Constants 17
- 2.1 Constants 17

3 Global Variables 19
- 3.1 Global Variables 19
- 3.2 EFI_GUID Variables 19
  - 3.2.1 Exported by EFI Library 19
  - 3.2.2 For EFI IDs 20
  - 3.2.3 For File System Information IDs 20
  - 3.2.4 For Device Path Media Protocol IDs 20
  - 3.2.5 For Sample Implementation Vendor Device Path GUIDs 20
  - 3.2.6 For Disk Type Entries 20
  - 3.2.7 For EFI Configuration Table Entries 21
- 3.3 Device Path Data Structures 21
- 3.4 Device I/O Protocol Interface 21
- 3.5 Memory Allocation Type 21

4 Functions and Macros 23
- 4.1 Introduction 23
- 4.2 Initialization Functions 23
  - InitializeLib() 24
  - InitializeUnicodeSupport() 25
- 4.3 Linked List Support Macros 26
  - 4.3.1 Related Definitions 27
  - LIST_ENTRY 27
  - 4.3.2 Macros 28
  - InitializeListHead() 28
  - IsListEmpty() 29
  - RemoveEntryList() 30
  - InsertTailList() 31
  - InsertHeadList() 32

4.4 String Functions 33
StrCmp() 34
StrnCmp() 35
StriCmp() 36
StrCpy() 37
StrCat() 38
StrLen() 39
StrSize() 40
StrDuplicate() 41
StrLwr() 42
StrUpr() 43
strlena() 44
strcmpa() 45
strncmpa() 46
Xtoi() 47
Atoi() 48
MetaMatch() 49
MetaiMatch() 51
ValueToString() 53
ValueToHex() 54
TimeToString() 55
GuidToString() 56
StatusToString() 57
DevicePathToStr() 59
4.5 Memory Support Functions 61
ZeroMem() 62
SetMem() 63
CopyMem() 64
CompareMem() 65
AllocatePool() 66
AllocateZeroPool() 67
ReallocatePool() 68
FreePool() 69
GrowBuffer() 70
LibMemoryMap() 72
4.6 Text I/O Functions 73
Input() 75
Iinput() 76
Output() 77
Print() 78
PrintAt() 79
Iprint() 80
IprintAt() 81
APrint() 82
SPrint() 83
PoolPrint() 84
CatPrint() 85
DumpHex() 87

LibIsValidTextGraphics() 89
IsValidAscii() 90
IsValidEfiCntlChar() 91
4.7 Math Functions 92
LShiftU64() 93
RshiftU64() 94
MultU64x32() 95
DivU64x32() 96
4.8 Spin Lock Functions 97
4.8.1 Related Definitions 98
FLOCK 98
4.8.2 Functions 99
InitializeLock() 99
AcquireLock() 100
ReleaseLock() 101
4.9 Handle and Protocol Support Functions 102
LibLocateHandle() 103
LibLocateHandleByDiskSignature() 105
LibLocateProtocol() 107
LibInstallProtocolInterfaces() 108
LibUninstallProtocolInterfaces() 109
LibReinstallProtocolInterfaces() 110
LibScanHandleDatabase() 111
LibGetManagingDriverBindingHandles() 114
LibGetParentControllerHandles() 115
LibGetChildControllerHandles() 116
LibGetManagedControllerHandles() 117
LibGetManagedChildControllerHandles() 118
4.10 File I/O Support Functions 120
LibOpenRoot() 121
LibFileInfo() 122
LibFileSystemInfo() 123
LibFileSystemVolumeLabelInfo() 124
ValidMBR() 125
OpenSimpleReadFile() 126
ReadSimpleReadFile() 128
CloseSimpleReadFile() 130
4.11 Device Path Support Functions 131
DevicePathFromHandle() 132
DevicePathInstance() 133
DevicePathInstanceCount() 134
AppendDevicePath() 135
AppendDevicePathNode() 136
AppendDevicePathInstance() 137
FileDevicePath() 138
DevicePathSize() 139
DuplicateDevicePath() 140
LibDevicePathToInterface() 141

UnpackDevicePath() 142
LibMatchDevicePaths() 143
LibDuplicateDevicePathInstance() 144
4.12 PCI Functions and Macros 145
PciFindDeviceClass() 146
PciFindDevice() 148
InitializeGlobalIoDevice() 152
ReadPort() 154
WritePort() 155
ReadPciConfig() 156
WritePciConfig() 157
inp() 158
outp() 159
inpw() 160
outpw() 161
inpd() 162
outpd() 163
readpci8() 164
writepci8() 165
readpci16() 166
writepci16() 167
readpci32() 168
writepci32() 169
4.13 Miscellaneous Functions and Macros 170
LibGetVariable() 171
LibGetVariableAndSize() 172
LibDeleteVariable() 173
CompareGuid() 174
CR() 175
DecimaltoBCD() 176
BCDtoDecimal() 177
LibCreateProtocolNotifyEvent() 178
WaitForSingleEvent() 180
WaitForEventWithTimeout() 181
RtLibEnableVirtualMappings() 183
RtConvertList() 184
LibGetSystemConfigurationTable() 185

Figures

Figure 1-1. Memory Layout Conventions 13

Tables

Table 1-1. Organization and Contents of This Specification 11
Table 4-1. EFI Library Function and Macro Groups 23
Table 4-2. Initialization Functions 23
Table 4-3. Linked List Support Macros 26
Table 4-4. String Functions 33
Table 4-5. Syntax for Building the String *Pattern* (MetaMatch()) 49
Table 4-6. Examples of Patterns (MetaMatch()) 50
Table 4-7. Syntax for Building the String *Pattern* (MetaiMatch()) 51
Table 4-8. Examples of Patterns (MetaiMatch()) 52
Table 4-9. EFI_STATUS Output Formats 58
Table 4-10. Hardware Device Path Output Formats 59
Table 4-11. Messaging Device Path Output Formats 60
Table 4-12. Media Device Path Output Formats 60
Table 4-13. BIOS Boot Specification (BBS) Device Path Output Formats 60
Table 4-14. Memory Support Functions 61
Table 4-15. Format String Attribute and Argument Specification 73
Table 4-16. Text I/O Functions 74
Table 4-17. EFI Control Characters 91
Table 4-18. Math Functions 92
Table 4-19. Spin Lock Functions 97
Table 4-20. Handle and Protocol Support Functions 102
Table 4-21. File I/O Support Functions 120
Table 4-22. Device Path Support Functions 131
Table 4-23. PCI Functions and Macros 145
Table 4-24. Miscellaneous Functions and Macros 170

1 Introduction

1.1 Overview

The *Extensible Firmware Interface Specification* describes a set of application programming interfaces (APIs) and data structures that are exported by a system's firmware. During the development of a sample implementation of the *EFI Specification*, the need arose for a set of library functions to simplify the development process. These library functions are also useful in the implementation of the following:

- EFI Shells
- EFI Shell commands
- EFI applications
- EFI operating system (OS) loaders
- EFI device drivers

This document describes in detail each of the functions and macros that are present in the EFI Library, along with the constants and global variables that are exported.

1.2 Organization of this Document

This specification is organized as follows:

Table 1-1. Organization and Contents of This Specification

Chapter	Description
Chapter 1: Introduction	Provides an overview of the EFI Library Specification.
Chapter 2: Constants	Defines the constants that are exported by the EFI Library.
Chapter 3: Global Variables	Defines the global variables that are allocated by the EFI Library.
Chapter 4: Functions and Macros	Defines the functions and macros that are exported by the EFI Library.

1.3 Goals

The primary goal of the *Extensible Firmware Interface Library Specification* (hereafter referred to as the "*EFI Library Specification*") is to provide documentation for the collection of library functions that are available to developers of EFI firmware, the EFI Shell, EFI Shell applications, and shrink-wrapped OS boot loaders. These library functions complement the APIs described in the *EFI Specification.* The combination of the EFI APIs and the EFI Library functions provide all the functions that are required for the following:

- Basic console I/O
- Basic disk I/O
- Memory management
- Linked list management
- String manipulation

In addition, there are miscellaneous functions for the following:

- 64-bit math operations
- Spin locks
- Helper functions that are used to manage device handles, device protocols, and device paths

1.4 Target Audience

This document is intended for the following readers:

- Original equipment manufacturers (OEMs) who will be creating Intel® architecture–based platforms that are intended to boot shrink-wrap OSs
- BIOS developers, either those who create general-purpose BIOSes and other firmware products or those who modify these products for use in Intel architecture–based products
- OS developers who will be adapting their shrink-wrap OS products to run on Intel architecture–based platforms

1.5 Prerequisite Specifications

The following specifications are required reading for using this *EFI Library Specification:*

- *Extensible Firmware Interface Specification,* version 1.10 (final draft), Intel Corporation, 2002.
- *EFI 1.10 Specification Update*, version -001, Intel Corporation, 2003.

1.6 Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

1.6.1 Data Structure Descriptions

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

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

For the purposes of this specification, illustrations of data structures in memory will always show the lowest addresses at the bottom and the highest addresses at the top of the illustration, as shown in Figure 1-1 below. Bit positions are numbered from right to left.

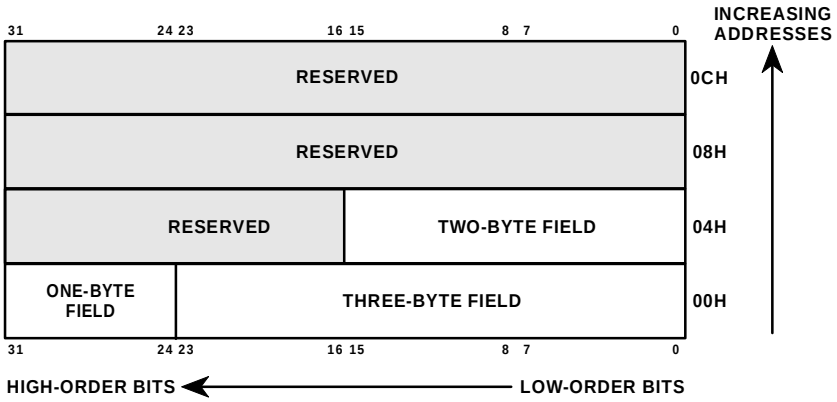

Figure 1-1. Memory Layout Conventions

The data structures described in this document generally have the following format:

STRUCTURE NAME: The formal name of the data structure.

Summary: A brief description of the data structure.

Prototype: A “C-style” type declaration for the data structure.

Parameters: A brief description of each field in the data structure prototype.

Description: A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this data structure.

1.6.2 Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

Prototype: A “C-style” procedure header defining the calling sequence.

Parameters: A brief description of each field in the procedure prototype.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this procedure.

Status Codes Returned: A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

1.6.3 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*.

1.6.4 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.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the 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 section in the *EFI 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 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 Constants

2.1 Constants

The following is the list of environment variable name constants that are exported by the EFI Library. These environment variable names are used by the *EFI Sample Implementation.*

```
#define VarLanguageCodes        L“LangCodes”;
#define VarLanguage             L“Lang”;
#define VarTimeout              L“Timeout”;
#define VarConsoleInp           L“ConIn”;
#define VarConsoleOut           L“ConOut”;
#define VarErrorOut             L“ErrOut”;
#define VarBootOption           L“Boot%04x”;
#define VarBootOrder            L“BootOrder”;
#define VarBootNext             L“BootNext”;
#define VarBootCurrent          L“BootCurrent”;
#define VarDriverOption         L“Driver%04x”;
#define VarDriverOrder          L“DriverOrder”;
#define VarSerialNumber         L“SerialNumber”;
#define VarSystemGuid           L“SystemGUID”;
#define VarConsoleInpDev        L“ConInDev”;
#define VarConsoleOutDev        L“ConOutDev”;
#define VarErrorOutDev          L“ErrOutDev”;

#define LanguageCodeEnglish     “eng"
```

3
Global Variables

3.1 Global Variables

The EFI Library exports three global variables that provide access to the following:

- EFI System Table (**ST**)
- EFI Boot Services (**BS**)
- EFI Runtime Services (**RT**)

The following is the declaration of these global variables:

```
extern EFI_SYSTEM_TABLE          *gST;
extern EFI_BOOT_SERVICES         *gBS;
extern EFI_RUNTIME_SERVICES      *gRT;
```

3.2 EFI_GUID Variables

3.2.1 Exported by EFI Library

The EFI Library also exports a group of **EFI_GUID** variables. These **EFI_GUID**s include protocol GUIDs and other miscellaneous GUIDs that are used by the *EFI Sample Implementation.*

The following is the declaration of these **EFI_GUID**s:

```
extern EFI_GUID DevicePathProtocol;
extern EFI_GUID LoadedImageProtocol;
extern EFI_GUID TextInProtocol;
extern EFI_GUID TextOutProtocol;
extern EFI_GUID BlockIoProtocol;
extern EFI_GUID DiskIoProtocol;
extern EFI_GUID FileSystemProtocol;
extern EFI_GUID LoadFileProtocol;
extern EFI_GUID DeviceIoProtocol;
extern EFI_GUID UnicodeCollationProtocol;
extern EFI_GUID SerialIoProtocol;

extern EFI_GUID VariableStoreProtocol;
extern EFI_GUID LegacyBootProtocol;
extern EFI_GUID VgaClassProtocol;
extern EFI_GUID TextOutSpliterProtocol;
extern EFI_GUID TextInSpliterProtocol;
extern EFI_GUID ErrorOutSpliterProtocol;
```

```
extern EFI_GUID SimpleNetworkProtocol;
extern EFI_GUID PxeBaseCodeProtocol;
extern EFI_GUID PxeCallbackProtocol;
extern EFI_GUID NetworkInterfaceIdentifierProtocol;
extern EFI_GUID UiProtocol;
extern EFI_GUID InternalShellProtocol;

extern EFI_GUID DriverBindingProtocol;
extern EFI_GUID BusSpecificDriverOverrideProtocol;
extern EFI_GUID PlatformDriverOverrideProtocol;
extern EFI_GUID PciRootBridgeIoProtocol;
extern EFI_GUID PciIoProtocol;
```

3.2.2 For EFI IDs

The following are the group of **EFI_GUID** variables for EFI IDs:

```
extern EFI_GUID EfiGlobalVariable;
extern EFI_GUID NullGuid;
```

3.2.3 For File System Information IDs

The following are the group of **EFI_GUID** variables for file system information IDs:

```
extern EFI_GUID GenericFileInfo;
extern EFI_GUID FileSystemInfo;
extern EFI_GUID FileSystemVolumeLabelInfo;
```

3.2.4 For Device Path Media Protocol IDs

The following are the group of **EFI_GUID** variables for device path media protocol IDs:

```
extern EFI_GUID PcAnsiProtocol;
extern EFI_GUID Vt100Protocol;
```

3.2.5 For Sample Implementation Vendor Device Path GUIDs

The following are the group of **EFI_GUID** variables for the *EFI Sample Implementation* vendor device path GUIDs:

```
extern EFI_GUID UnknownDevice;
```

3.2.6 For Disk Type Entries

The following are the group of **EFI_GUID** variables for the disk type entries:

```
extern EFI_GUID EfiPartTypeSystemPartitionGuid;
extern EFI_GUID EfiPartTypeLegacyMbrGuid;
```

3.2.7 For EFI Configuration Table Entries

The following are the group of **EFI_GUID** variables for the EFI Configuration Table entries:

```
extern EFI_GUID MpsTableGuid;
extern EFI_GUID AcpiTableGuid;
extern EFI_GUID SMBIOSTableGuid;
extern EFI_GUID SalSystemTableGuid;
```

3.3 Device Path Data Structures

The EFI Library also exports three device path data structures. These structures are used to build complete device paths.

```
extern EFI_DEVICE_PATH RootDevicePath[];
extern EFI_DEVICE_PATH EndDevicePath[];
extern EFI_DEVICE_PATH EndInstanceDevicePath[];
```

3.4 Device I/O Protocol Interface

The following is the global I/O Device I/O Protocol interface that is used to access the root PCI bus:

```
extern EFI_DEVICE_IO_INTERFACE  *GlobalIoFncs;
```

3.5 Memory Allocation Type

The following is the default memory allocation type for the EFI Library memory allocation functions:

```
extern EFI_MEMORY_TYPE PoolAllocationType;
```

4
Functions and Macros

4.1 Introduction

The functions and macros that are exported by the EFI Library are grouped as listed in Table 4-1.

Table 4-1. EFI Library Function and Macro Groups

Group	For more information, see Section…
Initialization functions	4.2
Linked list support macros	4.3
String functions	4.4
Memory support functions	4.5
Text I/O functions	4.6
Math functions	4.7
Spin lock functions	4.8
Handle and protocol functions	4.9
File I/O support functions	4.10
Device path support functions	4.11
PCI functions and macros	4.12
Miscellaneous functions	4.13

The following sections discuss each group in more detail and provide the API definitions.

4.2 Initialization Functions

The initialization functions in the EFI Library are used to initialize the execution environment so that other EFI Library functions may be used. Table 4-2 lists the initialization support functions that are described in the following sections.

Table 4-2. Initialization Functions

Name	Description
InitializeLib()	Initializes the EFI Library.
InitializeUnicodeSupport()	Initializes the use of language-dependent EFI Library functions.

InitializeLib()

Summary

Initializes the EFI Library.

Prototype

```
VOID
InitializeLib (
  IN EFI_HANDLE           ImageHandle,
  IN EFI_SYSTEM_TABLE     *SystemTable
  );
```

Parameters

ImageHandle

A handle for the image that is initializing the library. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

SystemTable

A pointer to the EFI System Table. Type **EFI_SYSTEM_TABLE** is defined in section 4.3 in the *EFI 1.10 Specification.*

Description

This function must be called to enable the use of all the EFI Library functions. Additional calls to this function are ignored. This function initializes all the global variables that are required by the EFI Library functions. In addition, it verifies the Cyclic Redundancy Checks (CRCs) for all the EFI System Tables.

InitializeUnicodeSupport()

Summary

Initializes the use of the language-dependent EFI Library functions.

Prototype

```
VOID
InitializeUnicodeSupport (
  IN CHAR8           *LangCode,
  );
```

Parameters

LangCode

The three-character ISO 639-2 language code.

Description

This function must be called to enable the use of all the language-dependent EFI Library functions. By default, the **InitializeLib()** function calls **InitializeUnicodeSupport()**. The only reason that this function would be called is to select a language other than the default one.

4.3 Linked List Support Macros

The EFI Library supplies a set of macros that allow doubly linked lists to be created and maintained. The head node of a doubly linked list is a **LIST_ENTRY** data structure. Each of the nodes in the linked list must also contain a **LIST_ENTRY** data structure. The **LIST_ENTRY** data structure simply contains a forward link and a backward link. The **LIST_ENTRY** data structure is defined in section 4.3.1, "Related Definitions."

Table 4-3 contains the list of macros that are described in the following sections.

Table 4-3. 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 double linked list.
InsertHeadList()	Adds a node to the beginning of a doubly linked list.

4.3.1 Related Definitions

LIST_ENTRY

Summary

Data structure for a doubly linked list entry.

Description

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY  *Flink;
  struct _LIST_ENTRY  *Blink;
} LIST_ENTRY;
```

Parameters

Flink

Pointer for the previous entry in the doubly linked list.

Blink

Pointer for the next entry in the doubly linked list.

Description

A doubly linked list entry uses *Flink* and *Blink* to link with each other.

4.3.2 Macros

InitializeListHead()

Summary

Initializes the head node of a doubly linked list.

Prototype

```
VOID
InitializeListHead(
  LIST_ENTRY *ListHead
  );
```

Parameters

ListHead

A pointer to the head node of a new doubly linked list. Type **LIST_ENTRY** is defined in section 4.3.1, “Related Definitions.”

Description

This macro initializes the forward and backward links of a new linked list. After initializing a linked list with this macro, the other linked list macros may be used to add and remove nodes from the linked list. It is up to the caller of this macro to allocate the memory for *ListHead*.

IsListEmpty()

Summary

Checks to see if a doubly linked list is empty or not.

Prototype

```
BOOLEAN
IsListEmpty(
  LIST_ENTRY *ListHead
  );
```

Parameters

ListHead

A pointer to the head node of a doubly linked list. Type **LIST_ENTRY** is defined in section 4.3.1, “Related Definitions.”

Description

This macro checks to see if the doubly linked list is empty. If the linked list does not contain any nodes, this macro returns **TRUE**. Otherwise, it returns **FALSE**.

Status Codes Returned

TRUE	The linked list is empty.
FALSE	The linked list is not empty.

RemoveEntryList()

Summary

Removes a node from a doubly linked list.

Prototype

```
VOID
RemoveEntryList(
  LIST_ENTRY *Entry
  );
```

Parameters

Entry

A pointer to a node in a linked list. Type **LIST_ENTRY** is defined in section 4.3.1, “Related Definitions.”

Description

This macro removes the node *Entry* from a doubly linked list. It is up to the caller of this macro to release the memory that is used by this node if releasing it is required.

InsertTailList()

Summary

Adds a node to the end of a doubly linked list.

Prototype

```
VOID
InsertTailList(
  LIST_ENTRY *ListHead,
  LIST_ENTRY *Entry
  );
```

Parameters

ListHead

A pointer to the head node of a doubly linked list. Type **LIST_ENTRY** is defined in section 4.3.1, “Related Definitions.”

Entry

A pointer to a node to add at the end of the doubly linked list.

Description

This macro adds the node *Entry* to the end of the doubly linked list denoted by *ListHead*.

InsertHeadList()

Summary

Adds a node to the beginning of a doubly linked list.

Prototype

```
VOID
InsertHeadList(
  LIST_ENTRY *ListHead,
  LIST_ENTRY *Entry
  );
```

Parameters

ListHead

A pointer to the head node of a doubly linked list. Type **LIST_ENTRY** is defined in section 4.3.1, “Related Definitions.”

Entry

A pointer to a node to insert at the beginning of a doubly linked list.

Description

This macro adds the node *Entry* at the beginning of the doubly linked list denoted by *ListHead*.

4.4 String Functions

The string functions in the EFI Library perform operations on Unicode and ASCII string. Table 4-4 contains the list of string support functions that are described in the following sections.

Table 4-4. 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.
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.

StrCmp()

Summary

Compares two Unicode strings.

Prototype

```
INTN
StrCmp (
  IN CHAR16   *s1,
  IN CHAR16   *s2
  );
```

Parameters

s1

Pointer to a **NULL**-terminated Unicode string.

s2

Pointer to a **NULL**-terminated Unicode string.

Description

This function compares the Unicode string *s1* to the Unicode string *s2*. If *s1* is identical to *s2*, then 0 is returned. Otherwise, the difference between the first mismatched Unicode characters is returned.

Status Codes Returned

0	*s1* is identical to *s2*.
≠ 0	*s1* is not identical to *s2*.

StrnCmp()

Summary

Compares a portion of two Unicode strings.

Prototype

```
INTN
StrnCmp (
  IN CHAR16    *s1,
  IN CHAR16    *s2,
  IN UINTN     len
  );
```

Parameters

s1

Pointer to a **NULL**-terminated Unicode string.

s2

Pointer to a **NULL**-terminated Unicode string.

len

Number of Unicode characters to compare.

Description

This function compares *len* Unicode characters from *s1* to *len* Unicode characters from *s2*. If all *len* characters from *s1* and *s2* are identical, then 0 is returned. Otherwise, the difference between the first mismatched ASCII characters is returned.

Status Codes Returned

0	*s1* is identical to *s2*.
≠ 0	*s1* is not identical to *s2*.

StriCmp()

Summary

Performs a case-insensitive comparison of two Unicode strings.

Prototype

```
INTN
StriCmp (
  IN CHAR16   *s1,
  IN CHAR16   *s2
  );
```

Parameters

s1

Pointer to a **NULL**-terminated Unicode string.

s2

Pointer to a **NULL**-terminated Unicode string.

Description

This function performs a case-insensitive comparison between the Unicode string *s1* and the Unicode string *s2* using the rules for the currently selected language code. If *s1* is equivalent to *s2*, then 0 is returned. If *s1* is lexically less than *s2*, then a negative number will be returned. If *s1* is lexically greater than *s2*, then a positive number will be returned. This function allows Unicode strings to be compared and sorted.

Status Codes Returned

0	*s1* is equivalent to *s2*.
> 0	*s1* is lexically greater than *s2*.
< 0	*s1* is lexically less than *s2*.

StrCpy()

Summary

Copies one Unicode string to another Unicode string.

Prototype

```
VOID
StrCpy (
  IN CHAR16   *Dest,
  IN CHAR16   *Src
  );
```

Parameters

Dest

Pointer to a **NULL**-terminated Unicode string.

Src

Pointer to a **NULL**-terminated Unicode string.

Description

This function copies the contents of the Unicode string *Src* to the Unicode string *Dest*.

StrCat()

Summary

Concatenates one Unicode string to another Unicode string.

Prototype

```
VOID
StrCat (
  IN CHAR16   *Dest,
  IN CHAR16   *Src
  );
```

Parameters

Dest

Pointer to a **NULL**-terminated Unicode string.

Src

Pointer to a **NULL**-terminated Unicode string.

Description

This function concatenates two Unicode strings. The contents of the Unicode string *Src* are concatenated to the end of the Unicode string *Dest*.

StrLen()

Summary

Determines the length of a Unicode string.

Prototype

```
UINTN
StrLen (
  IN CHAR16   *s1
  );
```

Parameters

s1

Pointer to a **NULL**-terminated Unicode string.

Description

This function returns the number of Unicode characters in the Unicode string *s1*.

StrSize()

Summary

Determines the size of a Unicode string in bytes.

Prototype

```
UINTN
StrSize (
  IN CHAR16   *s1
  );
```

Parameters

s1

Pointer to a **NULL**-terminated Unicode string.

Description

This function returns the size of the Unicode string *s1* in bytes.

StrDuplicate()

Summary

Duplicates a Unicode string.

Prototype

```
CHAR16 *
StrDuplicate (
  IN CHAR16   *Src
  );
```

Parameters

Src

Pointer to a **NULL**-terminated Unicode string.

Description

This function creates and returns a new copy of the Unicode string *Src*. The memory for the new string is allocated from pool.

StrLwr()

Summary

Converts all the characters in a Unicode string to lowercase.

Prototype

```
VOID
StrLwr (
  IN CHAR16   *Str
  );
```

Parameters

Str

Pointer to a **NULL**-terminated Unicode string.

Description

This function converts all the characters in the Unicode string *Str* to lowercase characters.

StrUpr()

Summary

Converts all the characters in a Unicode string to uppercase.

Prototype

```
VOID
StrUpr (
  IN CHAR16   *Str
  );
```

Parameters

Str

Pointer to a **NULL**-terminated Unicode string.

Description

This function converts all the characters in the Unicode string *Str* to uppercase characters.

strlena()

Summary

Determines the length of an ASCII string.

Prototype

```
UINTN
strlena (
  IN CHAR8    *s1
  );
```

Parameters

s1

Pointer to a **NULL**-terminated ASCII string.

Description

This function returns the length of the ASCII string *s1*.

strcmpa()

Summary

Compares two ASCII strings.

Prototype

```
UINTN
strcmpa (
  IN CHAR8    *s1,
  IN CHAR8    *s2
  );
```

Parameters

s1

Pointer to a **NULL**-terminated ASCII string.

s2

Pointer to a **NULL**-terminated ASCII string.

Description

This function compares the contents of the ASCII string *s1* to the contents of the ASCII string *s2*. If *s1* is identical to *s2*, then 0 is returned. Otherwise, the difference between the first mismatched ASCII characters is returned.

Status Codes Returned

0	*s1* is identical to *s2*.
≠ 0	*s1* is not identical to *s2*.

strncmpa()

Summary

Compares a portion of two ASCII strings.

Prototype

```
UINTN
strncmpa (
  IN CHAR8    *s1,
  IN CHAR8    *s2,
  IN UINTN    len
  );
```

Parameters

s1

Pointer to an ASCII string.

s2

Pointer to an ASCII string.

len

Number of ASCII characters to compare.

Description

This function compares *len* ASCII characters from *s1* to *len* ASCII characters from *s2*. If all *len* characters from *s1* and *s2* are identical, then 0 is returned. Otherwise, the difference between the first mismatched ASCII characters is returned.

Status Codes Returned

0	*s1* is identical to *s2* for the first *len* characters.
≠ 0	*s1* is not identical to *s2* for the first *len* characters.

Xtoi()

Summary

Converts a hexadecimal-formatted Unicode string to a value.

Prototype

```
UINTN
xtoi (
  IN CHAR16      *str
  );
```

Parameters

str

Pointer to a **NULL**-terminated Unicode string.

Description

This function converts the hexadecimal-formatted Unicode string *str* into an integer and returns that integer. Any preceding white space in *str* is ignored.

Atoi()

Summary

Converts a decimal-formatted Unicode string to a value.

Prototype

```
UINTN
Atoi (
  IN CHAR16  *str
  );
```

Parameters

str

Pointer to a **NULL**-terminated Unicode string.

Description

This function converts the decimal-formatted Unicode string *str* into an integer and returns that integer. Any preceding white space in *str* is ignored.

MetaMatch()

Summary

Checks to see if a Unicode string matches a given pattern.

Prototype

```
BOOLEAN
MetaMatch (
  IN CHAR16   *String,
  IN CHAR16   *Pattern
  );
```

Parameters

String

Pointer to a **NULL**-terminated Unicode string.

Pattern

Pointer to a **NULL**-terminated Unicode string.

Description

This function checks to see if the pattern of characters described by *Pattern* is found in *String*. If the pattern match succeeds, then **TRUE** is returned. Otherwise, **FALSE** is returned. The syntax shown in Table 4-5 can be used to build the string *Pattern*. Table 4-6 shows some examples using the syntax described in Table 4-5.

Table 4-5. Syntax for Building the String *Pattern* (MetaMatch())

Syntax	Description
*	Matches 0 or more characters.
?	Matches any one character.
[<char1><char2>…<charN>]	Matches any character in the set.
[<char1>-<char2>]	Matches any character between <char1> and <char2>.
<char>	Matches the character <char>.

Table 4-6. Examples of Patterns (MetaMatch())

Example Pattern	Description
`*.FW`	Matches all strings that end in "**`.FW`**."
`[a-z]`	Matches any lowercase character.
`[!@#$%^&*()]`	Matches any one of these symbols.
`z`	Matches the lowercase character **`z`**.
`DATA?.*`	Matches the string "**`DATA`**," followed by any character, followed by a "**`.`**", followed by any string.

Status Codes Returned

TRUE	*`Pattern`* was found in *`String`*.
FALSE	*`Pattern`* was not found in *`String`*.

MetaiMatch()

Summary

Performs a case-insensitive comparison of a Unicode pattern string and a Unicode string.

Prototype

```
BOOLEAN
MetaiMatch (
  IN CHAR16    *String,
  IN CHAR16    *Pattern
  );
```

Parameters

String

A pointer to a Unicode string.

Pattern

A pointer to a Unicode pattern string.

Description

This function checks to see if the pattern of characters described by *Pattern* are found in *String*. The pattern check is a case-insensitive comparison using the rules for the currently selected language code. If the pattern match succeeds, then **TRUE** is returned. Otherwise, **FALSE** is returned. The syntax shown in Table 4-7 can be used to build the string *Pattern*. Table 4-8 shows some examples using the syntax described in Table 4-7.

Table 4-7. Syntax for Building the String *Pattern* (MetaiMatch())

Syntax	Description
*	Matches 0 or more characters.
?	Matches any one character.
[<char1><char2>…<charN>]	Matches any character in the set.
[<char1>-<char2>]	Matches any character between <char1> and <char2>.
<char>	Matches the character <char>.

Table 4-8. Examples of Patterns (MetaiMatch())

Example Pattern	Description
`*.FW`	Matches all strings that end in "`.FW`" or "`.fw`" or "`.Fw`" or "`.fW`."
`[a-z]`	Matches any letter in the alphabet.
`[!@#$%^&*()]`	Matches any one of these symbols.
`z`	Matches the character '`z`' or '`Z`.'
`D?.*`	Matches the character '`D`' or '`d`,' followed by any character, followed by a "`.`", followed by any string.

Status Codes Returned

TRUE	*`Pattern`* was found in *`String`*.
FALSE	*`Pattern`* was not found in *`String`*.

ValueToString()

Summary

Converts an integer to decimal-formatted Unicode string.

Prototype

```
VOID
ValueToString (
  IN CHAR16   *Buffer,
  IN BOOLEAN  Comma,
  IN INT64    v
  );
```

Parameters

Buffer

Pointer to the Unicode string that will be returned by this function.

Comma

Tells if the converted string should be formatted with commas or not.

v

The integer value that is to be converted into a string.

Description

This function converts the integer *v* into a decimal formatted Unicode string. If *Comma* is **TRUE**, then the string is formatted with commas. Otherwise, it is not formatted with commas. The converted string is returned in *Buffer*.

ValueToHex()

Summary

Converts an integer to hexadecimal-formatted Unicode string.

Prototype

```
VOID
ValueToHex (
  IN CHAR16   *Buffer,
  IN UINT64   v
  );
```

Parameters

Buffer

Pointer to the Unicode string that will be returned by this function.

v

The integer value that is to be converted into a string.

Description

This function converts the integer *v* into a hexadecimal-formatted Unicode string. The converted string is returned in *Buffer*.

TimeToString()

Summary

Converts the time and date stored in a data structure into a Unicode string.

Prototype

```
VOID
TimeToString (
  OUT CHAR16      *Buffer,
  IN  EFI_TIME    *Time
  );
```

Parameters

Buffer

Pointer to the Unicode string that will be returned by this function.

Time

Pointer to a data structure containing the time and date. Type **EFI_TIME** is defined in **GetTime()** in the *EFI 1.10 Specification.*

Description

This function converts the **EFI_TIME** data structure *Time* in the Unicode string *Buffer*. The format of the Unicode string is as follows:

MM/DD/YY hh:mmA

MM	Month
DD	Day of the month
YY	Year
hh	Hour
mm	Minutes
A	AM/PM field. ‘a’ for AM and ‘p’ for PM.

GuidToString()

Summary

Converts a GUID data structure into a Unicode string.

Prototype

```
VOID
GuidToString (
  OUT CHAR16      *Buffer,
  IN EFI_GUID     *Guid
  );
```

Parameters

Buffer

Pointer to the Unicode string that will be returned by this function.

Guid

Pointer to a data structure containing the GUID. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function converts an **EFI_GUID** data structure *Guid* into the Unicode string *Buffer*. The format of the string is as follows:

LLLLLLLL-MMMM-HHHH-hh-ll-NNNNNNNNNNNN

LLLLLLLL	The time_low field.
MMMM	The time_mid field.
HHHH	The time_hi_and_version field.
hh	The clock_sequence_high_and_reserved field.
ll	The clock_seq_low field.
NNNNNNNNNNNN	The node identifier. This value is typically an Ethernet hardware ID.

StatusToString()

Summary

Converts an **EFI_STATUS** value into a Unicode string.

Prototype

```
VOID
StatusToString (
  OUT CHAR16      *Buffer,
  EFI_STATUS      Status
  );
```

Parameters

Buffer

Pointer to the Unicode string that will be returned by this function.

Status

EFI_STATUS value.

Description

This function converts the **EFI_STATUS** value *Status* into the Unicode string *Buffer*. Table 4-9 shows how **EFI_STATUS** values are converted to strings.

Table 4-9. EFI_STATUS Output Formats

EFI_STATUS Value	Output Format
EFI_SUCCESS	“Success”
EFI_LOAD_ERROR	“Load Error”
EFI_INVALID_PARAMETER	“Invalid Parameter”
EFI_UNSUPPORTED	“Unsupported”
EFI_BAD_BUFFER_SIZE	“Bad Buffer Size”
EFI_BUFFER_TOO_SMALL	“Buffer Too Small”
EFI_NOT_READY	“Not Ready”
EFI_DEVICE_ERROR	“Device Error”
EFI_WRITE_PROTECTED	“Write Protected”
EFI_OUT_OF_RESOURCES	“Out of Resources”
EFI_VOLUME_CORRUPTED	“Volume Corrupt”
EFI_VOLUME_FULL	“Volume Full”
EFI_NO_MEDIA	“No Media”
EFI_MEDIA_CHANGED	“Media Changed”
EFI_NOT_FOUND	“Not Found”
EFI_ACCESS_DENIED	“Access Denied”
EFI_NO_RESPONSE	“No Response”
EFI_NO_MAPPING	“No mapping”
EFI_TIMEOUT	“Time out”
EFI_NOT_STARTED	“Not started”
EFI_ALREADY_STARTED	“Already started”
EFI_ABORTED	“Aborted”
EFI_ICMP_ERROR	“ICMP Error”
EFI_TFTP_ERROR	“TFTP Error”
EFI_PROTOCOL_ERROR	“Protocol Error”
EFI_WARN_UNKOWN_GLYPH	“Warning Unknown Glyph”
EFI_WARN_DELETE_FAILED	“Warning Delete Failed”
EFI_WARN_WRITE_FAILURE	“Warning Write Failure”
EFI_WARN_BUFFER_TOO_SMALL	“Warning Buffer Too Small”

DevicePathToStr()

Summary

Converts a device path data structure into a printable **NULL**-terminated Unicode string.

Prototype

```
CHAR16 *
DevicePathToStr (
  EFI_DEVICE_PATH         *DevPath
  );
```

Parameters

DevPath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Description

This function converts a device path data structure into a **NULL**-terminated Unicode string. The memory for the Unicode string is allocated from pool, and a pointer to the Unicode string is returned. Table 4-10, Table 4-11, Table 4-12, and Table 4-13 show the conversions from the different device path types into printable strings. Device path nodes are separated by a '****' and device path instances are separated by a comma ('**,**').

Table 4-10. Hardware Device Path Output Formats

Hardware Device Path Type	Output Format
PCI Device Path	"PCI(<Device Number>|<Function Number>)"
PCCARD Device Path	"Pccard(Socket<Socket Number>)"
Memory Mapped Device Path	"MemMap(<Memory Type>:<Starting Address>-<Ending Address>)
Vendor Device Path	"VenHw(<Vendor GUID>)"
Controller Device Path	"Ctlr(<Controller>)"
Vendor Device Path(Legacy)	"VenHw(<Vendor GUID>:<Legacy Drive Letter>)"
ACPI Device Path	"Acpi(<ACPI HID>,<ACPI UID>)"
Extended ACPI Device Path	"Acpi(PNP<ACPI HID>,<ACPI UID>)"
Unknown	"?"

Table 4-11. Messaging Device Path Output Formats

Messaging Device Path Type	Output Format
ATAPI	“ATA(<Primary/Secondary>,<Master/Slave>)”
SCSI	“Scsi(<PUN>,<LUN>)”
Fibre Channel	“Fibre(<WWN>)”
1394	“1394(<GUID>)”
USB	“Usb(<Port>)”
I_2O	“I2O(<TID>)”
MAC Address	“MAC(<MAC address>)”
IPv4	“IPv4(not-done)”
IPv6	"IP-v6(not-done)"
InfiniBand*	“InfiniBand(not-done)”
UART	“Uart(<Baud Rate>,<Parity>,<Data Bits>,<Stop Bits>)”
Vendor-Defined	“VenMsg(<Vendor GUID>)”
Vendor(Legacy)	“VenMsg(<Vendor GUID>:<Legacy Drive Letter>)”
Unknown	“?”
UART Flow Control	“UartFlowCtrl(<Flow Control Map>)”

Table 4-12. Media Device Path Output Formats

Media Device Path Type	Output Format
Hard Drive	“HD(Part<Partition Number>,Sig<Signature>)”
CD-ROM	“CDROM(Entry<Boot Entry>)”
Vendor-Defined	“VenMedia(<Vendor GUID>)”
Vendor(Legacy)	“VenMedia(<Vendor GUID>:<Legacy Drive Letter>)”
File Path	“<File Name>”
Media Protocol	"<Protocol GUID>"
Unknown	“?”

Table 4-13. BIOS Boot Specification (BBS) Device Path Output Formats

BBS Device Path Type	Output Format
Floppy	“Floppy”
Hard Disk	“Harddrive”
CD-ROM	“CDROM”
PCMCIA	“PCMCIA”
USB	“USB”
Embedded Network	“Net”
Other	“?”
Unknown	“?”

4.5 Memory Support Functions

The EFI Library provides a set of functions that operate on buffers in memory. Buffers can either be allocated on the stack, as global variables, or from the memory pool. To prevent memory leaks, it is the caller's responsibility to maintain buffers that are allocated from pool. This maintenance means that the caller must free a buffer when that buffer is no longer needed. Table 4-14 contains the list of memory support functions that are described in the following sections.

Table 4-14. 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.

ZeroMem()

Summary

Fills a buffer with zeros.

Prototype

```
VOID
ZeroMem (
  IN VOID      *Buffer,
  IN UINTN     Size
  );
```

Parameters

Buffer

Pointer to the buffer to fill with zeros.

Size

Number of bytes in *Buffer* to fill with zeros.

Description

This functions fills *Size* bytes of *Buffer* with zeros.

SetMem()

Summary

Fills a buffer with a value.

Prototype

```
VOID
SetMem (
  IN VOID     *Buffer,
  IN UINTN    Size,
  IN UINT8    Value
  );
```

Parameters

Buffer

Pointer to the buffer to fill.

Size

Number of bytes in *Buffer* to fill.

Value

Value with which to fill *Buffer*.

Description

This function fills *Size* bytes of *Buffer* with *Value*.

CopyMem()

Summary

Copies the contents of one buffer to another buffer.

Prototype

```
VOID
CopyMem (
  IN VOID     *Dest,
  IN VOID     *Src,
  IN UINTN    len
  );
```

Parameters

Dest

Pointer to the destination buffer of the memory copy.

Src

Pointer to the source buffer of the memory copy.

Len

Number of bytes to copy from *Src* to *Dest*.

Description

This function copies *len* bytes from the buffer *Src* to the buffer *Dest*.

CompareMem()

Summary

Compares the contents of two buffers.

Prototype

```
INTN
CompareMem (
  IN VOID     *Dest,
  IN VOID     *Src,
  IN UINTN    len
  );
```

Parameters

Dest

Pointer to the buffer to compare.

Src

Pointer to the buffer to compare.

Len

Number of bytes to compare.

Description

This function compares *len* bytes of *Src* to *len* bytes of *Dest*. If the two buffers are identical for *len* bytes, then 0 is returned. Otherwise, the difference between the first two mismatched bytes is returned.

Status Codes Returned

0	*Dest* is identical to *Src* for *len* bytes.
≠ 0	*Dest* is not identical to *Src* for *len* bytes.

AllocatePool()

Summary

Allocates a buffer from memory with type *PoolAllocationType*.

Prototype

```
VOID *
AllocatePool (
  IN UINTN     Size
  );
```

Parameters

Size

The size of the buffer to allocate from pool.

Description

This function attempts to allocate *Size* bytes from memory with type *PoolAllocationType*. If the memory allocation fails, **NULL** is returned. Otherwise, a pointer to the allocated buffer is returned. Type *PoolAllocationType* is defined in section 3.5, “Memory Allocation Type.”

AllocateZeroPool()

Summary

Allocates and zeros buffer from memory.

Prototype

```
VOID *
AllocateZeroPool (
  IN UINTN     Size
  );
```

Parameters

Size

The size of the buffer to allocate from pool.

Description

This function attempts to allocate *Size* bytes from memory. If the memory allocation fails, **NULL** is returned. Otherwise, *Size* bytes of the allocated buffer are set to zero, and a pointer to the allocated buffer is returned.

ReallocatePool()

Summary

Adjusts the size of a previously allocated buffer.

Prototype

```
VOID *
ReallocatePool (
  IN VOID                  *OldPool,
  IN UINTN                 OldSize,
  IN UINTN                 NewSize
  );
```

Parameters

OldPool

A pointer to the buffer whose size is being adjusted.

OldSize

The size of the current buffer.

NewSize

The size of the new buffer.

Description

This function changes the size of a buffer allocated from pool from *OldSize* to *NewSize*. The contents of the old buffer are copied to the new buffer. If *NewSize* is zero, then *OldPool* is freed and **NULL** is returned. If *NewSize* is not zero and the new buffer cannot be allocated, then **NULL** is returned. If *NewSize* is not zero and the new buffer can be allocated, then the contents of *OldPool* are copied to the new buffer, *OldPool* is freed, and a pointer to the new buffer is returned.

FreePool()

Summary

Releases a previously allocated buffer.

Prototype

```
VOID
FreePool (
  IN VOID     *p
  );
```

Parameters

p

A pointer to the buffer to free.

Description

This function releases a previously allocated buffer. The freed memory is returned to the available pool. The buffer *p* must have been allocated with **AllocatePool()**.

GrowBuffer()

Summary

Either allocates a new buffer or increases the size of a previously allocated buffer.

Prototype

```
BOOLEAN
GrowBuffer(
  IN OUT EFI_STATUS   *Status,
  IN OUT VOID         **Buffer,
  IN UINTN            BufferSize
  );
```

Parameters

Status

Status from the last EFI API call.

Buffer

A pointer to the buffer to grow.

BufferSize

The new size of the buffer.

Description

This function either allocates a new buffer or increases the size of a previously allocated buffer.

If *Buffer* is **NULL** on entry, then this function will attempt to allocate a new buffer with size *BufferSize*. If the buffer is allocated, then *Buffer* will point to the new buffer, *Status* will be **EFI_SUCCESS**, and the function will return **TRUE**. If the buffer cannot be allocated, then *Buffer* will be set to **NULL**, *Status* will be set to **EFI_OUT_OF_RESOURCES**, and the function will return **FALSE**.

If *Buffer* is not **NULL** and *Status* is **EFI_BUFFER_TOO_SMALL**, then this function will free the old buffer, and attempt to reallocate a new buffer with size *BufferSize*. If that reallocation succeeds, then *Buffer* will point to the reallocated buffer, *Status* will be **EFI_SUCCESS**, and the function will return **TRUE**. If the buffer cannot be reallocated, then *Buffer* will be set to **NULL**, *Status* will be set to **EFI_OUT_OF_RESOURCES**, and the function will return **FALSE**.

If *Buffer* is not **NULL** and *Status* is **not** **EFI_BUFFER_TOO_SMALL**, then the buffer will be freed, *Buffer* will be set to **NULL**, and the function will return **FALSE**.

The main purpose of this function is to retry an EFI API call until a buffer of the appropriate size is allocated. The following is an example of how to use the **GrowBuffer()** function. The first

pass through the **while** loop uses the **GrowBuffer()** function to allocate a new 100-byte buffer. If the **GetVariable()** API call needs more than 100 bytes, it will return with status **EFI_BUFFER_TOO_SMALL**. Also, *BufferSize* will be set to the number of bytes that is required for the call to **GetVariable()** to succeed. So, the next iteration through the **while** loop will call **GrowBuffer()** again. This time, the buffer will be reallocated to the size that is required by **GetVariable()**. So, the next call to **GetVariable()** will succeed and the buffer that is used to store the variable will be exactly the right size.

```
EFI_STATUS              Status;
VOID                    *Buffer;
UINTN                   BufferSize;

Buffer = NULL;
BufferSize = 100;

while (GrowBuffer (&Status, &Buffer, BufferSize)) {
    Status = RT->GetVariable(Name,
                             VendorGuid,
                             NULL,
                             &BufferSize,
                             Buffer);
```

Status Codes Returned

TRUE	The buffer was reallocated.
FALSE	The buffer could not be reallocated.

LibMemoryMap()

Summary

Retrieves the system's current memory map.

Prototype

```
EFI_MEMORY_DESCRIPTOR *
LibMemoryMap (
  OUT UINTN                  *NoEntries,
  OUT UINTN                  *MapKey,
  OUT UINTN                  *DescriptorSize,
  OUT UINT32                 *DescriptorVersion
  );
```

Parameters

NoEntries

A pointer to the number of memory descriptors in the system.

MapKey

A pointer to the current memory map key.

DescriptorSize

A pointer to the size in bytes of a memory descriptor.

DescriptorVersion

A pointer to the version of the memory descriptor.

Description

This function retrieves and returns the system's current memory map. The number of memory map entries is returned in *NoEntries*, and the size of each entry in bytes is returned in *DescriptorSize*. Also, the version of the memory descriptor data structure is returned in *DescriptorVersion*, and the key for the current memory map is returned in *MapKey*. This function allocates the storage for the memory map from pool.

4.6 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. Many of the output functions use a *format string* to describe how to format the output of variable arguments. The format string consists of normal text and argument descriptors. There are no restrictions for how the normal text and argument descriptors can be mixed. Each argument descriptor is of the form **%w.lF**, where:

- **w** is an optional integer value that represents the argument width parameter.
- **l** is an optional integer value that represents the field width parameter.
- **F** is a set of optional field modifiers and the data type of the argument to print.

Table 4-15 lists the optional field modifiers and argument types.

Table 4-15. Format String Attribute and Argument Specification

Name	Description
0	Pad the field with zeros.
-	Left-justify the argument in the field.
,	Insert commas in a decimal-formatted integer.
*	The field width is provided as an argument.
n	Set the output attribute for this field to normal.
h	Set the output attribute for this field to highlight.
e	Set the output attribute for this field to error.
l	The argument value is 64 bits. The default is a 32-bit argument.
a	The argument is an ASCII string.
s	The argument is a Unicode string.
X	Print the argument as a hexadecimal value padded with zeros. The field width defaults to 8 for 32-bit arguments and 16 for 64-bit arguments.
x	Print the argument as a hexadecimal value.
D	Print the argument as a decimal value with optional commas.
C	The argument is a Unicode character.
T	The argument is a pointer to an **EFI_TIME** data structure. See the **TimeToString()** function for the output format of this field.
G	The argument is a pointer to a **EFI_GUID** data structure. See the **GuidToString()** function for the output format of this field.
R	The argument is an **EFI_STATUS** value. See the **StatusToString()** function for the output format of this field.
N	Set the output attribute to normal.
H	Set the output attribute to highlight.
E	Set the output attribute to error.

Table 4-16 contains the list of text I/O functions that are described in the following sections.

Table 4-16. 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.

Input()

Summary

Reads a Unicode string from the console-in device at the current cursor location.

Prototype

```
VOID
Input (
  IN CHAR16   *Prompt OPTIONAL,
  OUT CHAR16  *InStr,
  IN UINTN    StrLen
  );
```

Parameters

Prompt

A pointer to a Unicode string.

InStr

A pointer to the Unicode string that is used to store the string read from the console-in device.

StrLen

The maximum length of the Unicode string to read from the console-in device.

Description

This function reads a Unicode string from the console-in device at the current cursor location.

If *Prompt* is not **NULL**, then *Prompt* is displayed on the console-out device. Then, characters are read from the console-in device and displayed on the console-out device. In addition, these characters are stored in *InStr* until either a **\n** or a **\r** character is received. If the backspace key is pressed, then the last character is *InStr* is removed and the display is updated to show that the character has been erased. If more than *StrLen* characters are received, then the extra characters are ignored.

IInput()

Summary

Reads a Unicode string from the specified device at the current cursor location.

Prototype

```
VOID
IInput (
  IN SIMPLE_TEXT_OUTPUT_INTERFACE     *ConOut,
  IN SIMPLE_INPUT_INTERFACE           *ConIn,
  IN CHAR16                           *Prompt OPTIONAL,
  OUT CHAR16                          *InStr,
  IN UINTN                            StrLen
  );
```

Parameters

ConOut

A pointer to the output device's interface protocol.

ConIn

A pointer to the input device's interface protocol.

Prompt

A pointer to a Unicode string.

InStr

A pointer to the Unicode string that is used to store the string read from the input device.

StrLen

The maximum length of the Unicode string to read from the input device.

Description

This function reads a Unicode string from the specified device at the current cursor location.

If *Prompt* is not **NULL**, then *Prompt* is displayed on the *ConOut* device. Then, characters are read from the *ConIn* device and displayed on the *ConOut* device. In addition, these characters are stored in *InStr* until either a **\n** or a **\r** character is received. If the backspace key is pressed, then the last character is *InStr* is removed and the *ConOut* device is updated to show that the character has been erased. If more than *StrLen* characters are received from the *ConIn* device, then the extra characters are ignored.

Output()

Summary

Sends a Unicode string to the console-out device at the current cursor location.

Prototype

```
VOID
Output (
  IN CHAR16   *Str
  );
```

Parameters

Str

A pointer to a Unicode string.

Description

This function sends the Unicode string *Str* to the console-out device that is specified in the EFI System Table.

Print()

Summary

Sends a formatted Unicode string to the console-out device at the current cursor location.

Prototype

```
UINTN
Print (
  IN CHAR16   *fmt,
  ...
  );
```

Parameters

Fmt

A pointer to a Unicode string containing format information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. This formatted Unicode string is then sent to the console-out device. The length of the formatted Unicode string is returned.

PrintAt()

Summary

Sends a formatted Unicode string to the console-out device at the specified cursor location.

Prototype

```
UINTN
PrintAt (
  IN UINTN    Column,
  IN UINTN    Row,
  IN CHAR16   *fmt,
  ...
  );
```

Parameters

Column

The column number on the console-out device.

Row

The row number on the console-out device.

fmt

A pointer to a Unicode string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. This formatted Unicode string is then sent to the console-out device at the cursor location specified by *Column* and *Row*. The length of the formatted Unicode string is returned.

Iprint()

Summary

Sends a formatted Unicode string to the specified device at the current cursor location.

Prototype

```
UINTN
IPrint (
  IN SIMPLE_TEXT_OUTPUT_INTERFACE     *Out,
  IN CHAR16                           *fmt,
  ...
  );
```

Parameters

Out

A pointer to the output device's interface protocol.

fmt

A pointer to a Unicode string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. This formatted Unicode string is then sent to the device specified by *Out* at the current cursor location. The length of the formatted Unicode string is returned.

IprintAt()

Summary

Sends a formatted Unicode string to the specified device at the specified cursor location.

Prototype

```
UINTN
IPrintAt (
  IN SIMPLE_TEXT_OUTPUT_INTERFACE     *Out,
  IN UINTN                            Column,
  IN UINTN                            Row,
  IN CHAR16                           *fmt,
  ...
  );
```

Parameters

Out

A pointer to the output devices interface protocol.

Column

The column number on the console-out device.

Row

The row number on the console-out device.

fmt

A pointer to a Unicode string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. This formatted Unicode string is then sent to the device specified by *Out* at the cursor location specified by *Column* and *Row*. The length of the formatted Unicode string is returned.

APrint()

Summary

Sends a formatted Unicode string to the console-out device at the current cursor location.

Prototype

```
UINTN
APrint (
  IN char     *fmt,
  ...
  );
```

Parameters

fmt

A pointer to an ASCII string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. This formatted Unicode string is then sent to the console-out device. The length of the formatted Unicode string is returned.

SPrint()

Summary

Sends a formatted Unicode string to the specified buffer.

Prototype

```
UINTN
SPrint (
  OUT CHAR16  *Str,
  IN UINTN    StrSize,
  IN CHAR16   *fmt,
  ...
  );
```

Parameters

Str

A pointer to a Unicode string.

StrSize

The maximum length of the Unicode string *Str*.

fmt

A pointer to a Unicode string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. Up to *StrSize* characters of this formatted Unicode string is then stored in *Str*. The length of the formatted Unicode string is returned.

PoolPrint()

Summary

Sends a formatted Unicode string to a buffer allocated from pool.

Prototype

```
CHAR16 *
PoolPrint (
  IN CHAR16           *fmt,
  ...
  );
```

Parameters

fmt

A pointer to a Unicode string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. Storage for the formatted Unicode string is allocated from pool. A pointer to the formatted Unicode string is returned. It is the caller's responsibility to free the allocated buffer.

CatPrint()

Summary

Concatenates a formatted Unicode string to a Unicode string allocated from pool.

Prototype

```
CHAR16 *
CatPrint (
  IN OUT POOL_PRINT   *Str,
  IN CHAR16           *fmt,
  ...
  );
```

Parameters

Str

A pointer to the **POOL_PRINT** data structure that contains a Unicode string. Type **POOL_PRINT** is defined in “Related Definitions” below.

fmt

A pointer to a Unicode string that contains formatting information.

...

Variable-length argument list.

Description

This function uses the format string *fmt* and the variable argument list to build a formatted Unicode string. *Str* is grown to accommodate the formatted Unicode string, and the formatted Unicode string is appended to the end of *Str*. A pointer to the concatenated Unicode string is returned.

Related Definitions

```
//*************************************
// POOL_PRINT
//*************************************
typedef struct {
  CHAR16      *str;
  UINTN       len;
  UINTN       maxlen;
} POOL_PRINT;
```

str

A Unicode string.

len

Length of the Unicode string.

maxlen

Maximum length of the Unicode string.

DumpHex()

Summary

Prints the contents of a buffer in hexadecimal format.

Prototype

```
VOID
DumpHex (
  IN UINTN        Indent,
  IN UINTN        Offset,
  IN UINTN        DataSize,
  IN VOID         *UserData
  );
```

Parameters

Indent

Number of spaces to indent the text output.

Offset

Byte offset within *UserData* at which to start printing.

DataSize

The number of bytes of data to print from *UserData*.

UserData

A pointer to the buffer of data to print.

Description

This function prints the contents of *UserData*. The format of each line of output is a 4-byte address printed in hexadecimal, followed by 16 bytes of data printed in hexadecimal, followed by 16 ASCII characters. If the ASCII characters are not printable, then they are substituted with a period. The entire output is indented by *Indent* spaces. The output starts *Offset* bytes into *UserData*, and a total of *DataSize* bytes are printed. The following is a sample output with an *Indent* of 0, *Offset* of 1, a *DataSize* of 100, and *UserData* pointing at the EFI System Table.

Sample Output:

```
00000001: 49 42 49 20 53 59 53 54-00 00 01 00 60 00 00 00  *EFI SYST....`...*
00000011: EC 95 4A D3 00 00 00 00-88 D9 DA 01 64 DB DA 01  *..J.........d...*
00000021: 88 D9 DA 01 28 DB DA 01-08 D6 DA 01 28 D7 DA 01  *....(.......(...*
00000031: 00 D0 41 00 60 7B 41 00-00 00 00 00 00 00 00 00  *..A.`.A.........*
00000041: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
00000051: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  *................*
00000061: 00 00 00 00                                     *....*
```

LibIsValidTextGraphics()

Summary

Determines if a graphic is a supported Unicode box drawing character.

Prototype

```
BOOLEAN
LibIsValidTextGraphics (
  IN  CHAR16   Graphic,
  OUT CHAR8   *PcAnsi,    OPTIONAL
  OUT CHAR8   *Ascii      OPTIONAL
  );
```

Parameters

Graphic

Unicode character to test.

PcAnsi

Optional pointer to return the PC ANSI equivalent of *Graphic*.

Ascii

Optional pointer to return the ASCII equivalent of *Graphic*.

Description

This function returns **TRUE** if the character *Graphic* adheres to the range of legal Unicode box drawing characters. The criteria is that the upper byte contains a 0x25 or 0x21. If the value is **TRUE** and the character also has a mapping for either the *PcAnsi* or *Ascii* character set, these mappings will be returned. Otherwise, the function is not deemed to be a box drawing character and **FALSE** is returned.

Status Codes Returned

TRUE	The character is a Unicode box drawing character.
FALSE	The character is not a Unicode box drawing character.

IsValidAscii()

Summary

Determines if the argument is a legal ASCII character.

Prototype

```
BOOLEAN
IsValidAscii (
  IN  CHAR16   Ascii
  );
```

Parameters

Ascii

Number of spaces to indent the text output.

Description

This function returns **TRUE** if *Ascii* lies within the legal range of ASCII elements, namely 0x20 and 0x7F, respectively. Otherwise, the function returns **FALSE**.

Status Codes Returned

TRUE	The character is a legal ASCII element.
FALSE	The character is not a legal ASCII element.

IsValidEfiCntlChar()

Summary

Determines if the character is one of the four EFI control characters.

Prototype

```
BOOLEAN
IsValidEfiCntlChar (
  IN  CHAR16   c
  );
```

Parameters

c

Character to determine if it is also a control character.

Description

This function returns **TRUE** if the input character *c* is a legal EFI control character. Otherwise, the value **FALSE** is returned. Table 4-17 lists the possible EFI control characters.

Table 4-17. EFI Control Characters

Name	Value
CHAR_NULL	0x0000
CHAR_BACKSPACE	0x0008
CHAR_TAB	0x0009
CHAR_LINEFEED	0x000A
CHAR_CARRIAGE_RETURN	0x000D

Status Codes Returned

TRUE	The character is an EFI control character.
FALSE	The character is not an EFI control character.

4.7 Math Functions

The EFI Library provides a few math functions to operate on 64-bit operands. These math functions include shift operations, multiplication, and division. Table 4-18 lists the set of 64-bit math functions that are described in the following sections.

Table 4-18. 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.
MultU64x32()	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.

LShiftU64()

Summary

Shifts a 64-bit integer left between 0 and 63 bits.

Prototype

```
UINT64
LShiftU64 (
  IN UINT64   Operand,
  IN UINTN    Count
  );
```

Parameters

Operand

The 64-bit operand to shift left.

Count

The number of bits to shift left.

Description

This function shifts the 64-bit value *Operand* to the left by *Count* bits. The shifted value is returned.

RshiftU64()

Summary

Shifts a 64-bit integer right between 0 and 63 bits.

Prototype

```
UINT64
RShiftU64 (
  IN UINT64   Operand,
  IN UINTN    Count
  );
```

Parameters

Operand

The 64-bit operand to shift right.

Count

The number of bits to shift right.

Description

This function shifts the 64-bit value *Operand* to the right by *Count* bits. The shifted value is returned.

MultU64x32()

Summary

Multiplies a 64-bit unsigned integer by a 32-bit unsigned integer and generates a 64-bit unsigned result.

Prototype

```
UINT64
MultU64x32 (
  IN UINT64   Multiplicand,
  IN UINTN    Multiplier
  );
```

Parameters

Multiplicand

A 64-bit unsigned value.

Multiplier

A 32-bit unsigned value.

Description

This function multiplies the 64-bit unsigned value *Multiplicand* by the 32-bit unsigned value *Multiplier* and generates a 64-bit unsigned result. This 64-bit unsigned result is returned.

DivU64x32()

Summary

Divides a 64-bit unsigned integer by a 32-bit unsigned integer and generates a 64-bit unsigned result and a 32-bit unsigned remainder.

Prototype

```
UINT64
DivU64x32 (
  IN UINT64   Dividend,
  IN UINTN    Divisor,
  OUT UINTN   *Remainder OPTIONAL
  );
```

Parameters

Dividend

A 64-bit unsigned value.

Divisor

A 32-bit unsigned value.

Remainder

A pointer to a 32-bit value.

Description

This function divides the 64-bit unsigned value *Dividend* by the 32-bit unsigned value *Divisor* and generates a 32-bit unsigned quotient. If *Remainder* is not **NULL**, then the 32-bit unsigned remainder is returned in *Remainder*. This function returns the 32-bit unsigned quotient.

4.8 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 at several different priority levels. A spin lock is stored in an **FLOCK** data structure, which is defined in section 4.8.1, "Related Definitions."

Table 4-19 lists the support functions for creating and maintaining spin locks. These functions are described in the following sections.

Table 4-19. Spin Lock Functions

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

4.8.1 Related Definitions

FLOCK

Summary

Data structure for a lock.

Prototype

```
typedef struct _FLOCK {
  EFI_TPL     Tpl;
  EFI_TPL     OwnerTpl;
  UINTN       Lock;
} FLOCK;
```

Parameters

Tpl

Task Priority Level (TPL) for this lock.

OwnerTpl

The owner's TPL for this lock.

Lock

Count of the lock.

Description

This structure is used to maintain the related information for a lock.

4.8.2 Functions

InitializeLock()

Summary

Initializes a basic mutual exclusion lock.

Prototype

```
VOID
InitializeLock (
  IN OUT FLOCK    *Lock,
  IN EFI_TPL      Priority
  );
```

Parameters

Lock

A pointer to the lock data structure to initialize. Type **FLOCK** is defined in section 4.8.1, “Related Definitions.”

Priority

The task priority level of the lock. Type **EFI_TPL** is defined in **RaiseTpl()** in the *EFI 1.10 Specification.*

Description

This function initializes a basic mutual exclusion lock. Each lock provides mutual exclusion access at its task priority level. Because there is no preemption or multiprocessor support in EFI, acquiring the lock consists only of raising to the lock’s task priority level.

AcquireLock()

Summary

Acquires ownership of a lock.

Prototype

```
VOID
AcquireLock (
  IN FLOCK    *Lock
  );
```

Parameters

Lock

A pointer to the lock to acquire.

Description

This function raises the system's current task priority level to the task priority level of the mutual exclusion lock. Then, it acquires ownership of the lock.

ReleaseLock()

Summary

Releases ownership of a lock.

Prototype

```
VOID
ReleaseLock (
  IN FLOCK    *Lock
  );
```

Parameters

Lock

A pointer to the lock to release.

Description

This function releases ownership of the mutual exclusion lock and restores the system's task priority level to its previous level.

4.9 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 4-20 lists the set of helper functions that are described in the following sections.

Table 4-20. 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.

LibLocateHandle()

Summary

Returns an array of handles that support the requested protocol in a buffer allocated from pool.

Prototype

```
EFI_STATUS
LibLocateHandle (
  IN EFI_LOCATE_SEARCH_TYPE   SearchType,
  IN EFI_GUID                 *Protocol OPTIONAL,
  IN VOID                     *SearchKey OPTIONAL,
  IN OUT UINTN                *NoHandles,
  OUT EFI_HANDLE              **Buffer
  );
```

Parameters

SearchType

Specifies which handle(s) are to be returned. Type **EFI_LOCATE_SEARCH_TYPE** is defined in **LocateHandle()** in the *EFI 1.10 Specification.*

Protocol

Provides the protocol to search by. This parameter is only valid for *SearchType* **ByProtocol**. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

SearchKey

Supplies the search key depending on the *SearchType*.

NoHandles

The number of handles that is returned in *Buffer*.

Buffer

A pointer to the buffer to return the requested array of handles that support *Protocol*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

The **LibLocateHandle()** function returns one or more handles that match the *SearchType* request. *Buffer* is allocated from pool, and the number of entries in *Buffer* is returned in *NoHandles*. Each *SearchType* is described below:

AllHandles	*Protocol* and *SearchKey* are ignored and the function returns an array of every handle in the system.
ByRegisterNotify	*SearchKey* supplies the *Registration* value that is returned by **RegisterProtocolNotify()**. The function returns the next handle that is new for the *Registration*. Only one handle is returned at a time, and the caller must loop until no more handles are returned. *Protocol* is ignored for this search type.
ByProtocol	All handles that support *Protocol* are returned. *SearchKey* is ignored for this search type.

Status Codes Returned

EFI_SUCCESS	The resulting array of handles was returned.
EFI_NOT_FOUND	No handles match the search.
EFI_OUT_OF_RESOURCES	There is not enough pool memory to store the matching results.

LibLocateHandleByDiskSignature()

Summary

Returns an array of handles that support the requested protocol in a buffer allocated from pool.

Prototype

```
EFI_STATUS
LibLocateHandleByDiskSignature (
  IN UINT8                MBRType,
  IN UINT8                SignatureType,
  IN VOID                 *Signature,
  IN OUT UINTN            *NoHandles,
  OUT EFI_HANDLE          **Buffer
  );
```

Parameters

MBRType

Specifies the type of Master Boot Record (MBR) for which to search. It can either be the PC-AT*-compatible MBR or an EFI Partition Table Header.

SignatureType

Specifies the type of signature to look for in the MBR. This signature can either be a 32-bit signature or a GUID signature.

Signature

A pointer to a 32-bit disk signature or a pointer to a GUID disk signature. The type depends on *SignatureType*.

NoHandles

The number of handles returned in *Buffer*.

Buffer

A pointer to the buffer to return the requested array of handles that support *Protocol*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

The **LibLocateHandleByDiskSignature()** function returns one or more handles to disk devices that match the specified *MBRType*, *SignatureType*, and *Signature*. *Buffer* is allocated from pool, and the number of entries in *Buffer* is returned in *NoHandles*. The valid values for *MBRType* and *SignatureType* are defined in “Related Definitions” below.

Related Definitions

```
//*******************************************************
// Valid values for MBRType
//*******************************************************
#define MBR_TYPE_PCAT                       0x01
#define MBR_TYPE_EFI_PARTITION_TABLE_HEADER 0x02

//*******************************************************
// Valid values for SignatureType
//*******************************************************
#define SIGNATURE_TYPE_MBR                  0x01
#define SIGNATURE_TYPE_GUID                 0x02
```

Status Codes Returned

EFI_SUCCESS	The resulting array of handles was returned.
EFI_NOT_FOUND	No handles match the search.
EFI_OUT_OF_RESOURCES	There is not enough pool memory to store the matching results.

LibLocateProtocol()

Summary

Returns the first protocol instance that matches the given protocol.

Prototype

```
EFI_STATUS
LibLocateProtocol (
  IN EFI_GUID                 *Protocol,
  OUT VOID                    **Interface
  );
```

Parameters

Protocol

The protocol for which to search. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Interface

On return, a pointer to the first interface that matches *Protocol*.

Description

The **LibLocateProtocol()** function finds all the device handles that support *Protocol* and returns a pointer to the protocol instance from the first handle in *Interface*. If no protocol instances are found, then *Interface* is set to **NULL**.

Status Codes Returned

EFI_SUCCESS	A protocol instance matching *Protocol* was found.
EFI_NOT_FOUND	No protocol instances were found that match *Protocol*.

LibInstallProtocolInterfaces()

Summary

Installs one or more protocol interfaces in the EFI Boot Services environment.

Prototype

```
EFI_STATUS
LibInstallProtocolInterfaces (
  IN OUT EFI_HANDLE       *Handle,
  ...
  );
```

Parameters

Handle

The handle on which to install the new protocol interfaces, or **NULL** if a new handle is to be allocated. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

...

A variable argument list containing pairs of protocol GUIDs and protocol interfaces.

Description

This function installs a set of protocol interfaces in the EFI Boot Services environment. It removes arguments from the variable argument list in pairs. The first item is always a pointer to the protocol's GUID, and the second item is always a pointer to the protocol's interface. These pairs are used to call the EFI Boot Services routine **InstallProtocolInterface()** to add one protocol interface to *Handle*. If *Handle* is **NULL** on entry, then a new handle will be allocated. The pairs of arguments are removed from the variable argument list until a **NULL** protocol GUID value is found. If any errors are generated while the protocol interfaces are being installed, then all the protocols added in this call will be removed.

Status Codes Returned

EFI_SUCCESS	All the protocol interfaces were installed.
EFI_OUT_OF_RESOURCES	There was not enough memory in pool to install all the protocols.

LibUninstallProtocolInterfaces()

Summary

Removes one or more protocol interfaces from the EFI Boot Services environment.

Prototype

```
VOID
LibUninstallProtocolInterfaces (
  IN EFI_HANDLE           Handle,
  ...
  );
```

Parameters

Handle

The handle from which to remove the protocol interfaces. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

...

A variable argument list containing pairs of protocol GUIDs and protocol interfaces.

Description

This function removes a set of protocol interfaces from the EFI Boot Services environment. It removes arguments from the variable argument list in pairs. The first item is always a pointer to the protocol's GUID, and the second item is always a pointer to the protocol's interface. These pairs are used to call the EFI Boot Services routine **UninstallProtocolInterface()** to remove one protocol interface from *Handle*. The pairs of arguments are removed from the variable argument list until a **NULL** protocol GUID value is found.

Status Codes Returned

EFI_SUCCESS	All the protocol interfaces were removed.
EFI_NOT_FOUND	One of the protocol interfaces could not be found.

LibReinstallProtocolInterfaces()

Summary

Replaces one or more protocol interfaces in the EFI Boot Services environment.

Prototype

```
EFI_STATUS
LibReinstallProtocolInterfaces (
  IN OUT EFI_HANDLE           *Handle,
  ...
  );
```

Parameters

Handle

The handle to remove the protocol interfaces from. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

...

A variable argument list containing triplets of protocol GUIDs, old protocol interfaces, and new protocol interfaces.

Description

This function replaces a set of protocol interfaces in the EFI Boot Services environment. It removes arguments from the variable argument list in triplets. The first item is always a pointer to the protocol's GUID, the second item is always a pointer to the current protocol interface, and the third item is always a pointer to the new protocol interface. These triplets are used to call the EFI Boot Services routine **ReinstallProtocolInterface()** to replace one protocol interface in *Handle*. The triplets of arguments are removed from the variable argument list until a **NULL** protocol GUID value is found. If there are any errors in this process, then the EFI Boot Services environment is restored to the state it had just before the call to this function was made.

Status Codes Returned

EFI_SUCCESS	All the protocol interfaces were replaced.
EFI_NOT_FOUND	One of the protocol interfaces could not be found.

LibScanHandleDatabase()

Summary

Scans the handle database and collects EFI Driver Model–related information.

Prototype

```
EFI_STATUS
LibScanHandleDatabase (
  EFI_HANDLE  DriverBindingHandle,        OPTIONAL
  UINT32      *DriverBindingHandleIndex,  OPTIONAL
  EFI_HANDLE  ControllerHandle,           OPTIONAL
  UINT32      *ControllerHandleIndex,     OPTIONAL
  UINTN       *HandleCount,
  EFI_HANDLE  **HandleBuffer,
  UINT32      **HandleType
  );
```

Parameters

DriverBindingHandle

If this parameter is not **NULL**, then information is collected about the EFI driver that produced the Driver Binding Protocol on this handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DriverBindingHandleIndex

If this parameter is not **NULL** and *DriverBindingHandle* is not **NULL**, then this parameter returns the index of *HandleBuffer* that contains *DriverBindingHandle*.

ControllerHandle

If this parameter is not **NULL**, then information is collected about this controller.

ControllerHandleIndex

If this parameter is not **NULL** and *ControllerHandle* is not **NULL**, then this parameter returns the index of *HandleBuffer* that contains *ControllerHandle*.

HandleCount

Returns the number of handles in *HandleBuffer*.

HandleBuffer

Returns the array of handles in the handle database. This buffer is allocated by this call, and the caller is responsible for freeing this buffer.

HandleType

Returns the array of attributes of the handles in the handle database. See "Related Definitions" for the list of available attributes. This buffer is allocated by this call, and the caller is responsible for freeing this buffer.

Description

This function collects EFI Driver Model related information about handles in the handle database. This function always allocated and returns the array of handles in the handle database and an array of attributes that describe EFI Driver Model related information about each handle. The number of handles is returned in *HandleCount*. The list of handles is returned in *HandleBuffer*. The list of attributes is returned in *HandleType*. The index of the handle associated with *DriverBindingHandle* is returned in *DriverBindingHandleIndex* if both are not **NULL**, and the index of the handle associated with *ControllerHandle* is returned in *ControllerHandleIndex* if both are not **NULL**.

Related Definitions

```
//*******************************************************
// Attributes for HandleType
//*******************************************************
#define EFI_HANDLE_TYPE_UNKNOWN                    0x000
#define EFI_HANDLE_TYPE_IMAGE_HANDLE               0x001
#define EFI_HANDLE_TYPE_DRIVER_BINDING_HANDLE      0x002
#define EFI_HANDLE_TYPE_DEVICE_DRIVER              0x004
#define EFI_HANDLE_TYPE_BUS_DRIVER                 0x008
#define EFI_HANDLE_TYPE_DRIVER_CONFIGURATION_HANDLE 0x010
#define EFI_HANDLE_TYPE_DRIVER_DIAGNOSTICS_HANDLE  0x020
#define EFI_HANDLE_TYPE_COMPONENT_NAME_HANDLE      0x040
#define EFI_HANDLE_TYPE_DEVICE_HANDLE              0x080
#define EFI_HANDLE_TYPE_PARENT_HANDLE              0x100
#define EFI_HANDLE_TYPE_CONTROLLER_HANDLE          0x200
#define EFI_HANDLE_TYPE_CHILD_HANDLE               0x400
```

Following is a description of the fields in the above definition.

EFI_HANDLE_TYPE_IMAGE_HANDLE	This bit is set if the handle supports the Loaded Image Protocol.
EFI_HANDLE_TYPE_DRIVER_ BINDING_HANDLE	This bit is set if the handle supports the Driver Binding Protocol.
EFI_HANDLE_TYPE_DRIVER_ CONFIGURATION_HANDLE	This bit is set if the handle supports the Driver Configuration Protocol.
EFI_HANDLE_TYPE_DRIVER_ DIAGNOSTICS_HANDLE	This bit is set if the handle supports the Driver Diagnostics Protocol.
EFI_HANDLE_TYPE_COMPONENT_ NAME_HANDLE	This bit is set if the handle supports the Component Name Protocol.
EFI_HANDLE_TYPE_DEVICE_HANDLE	This bit is set if the handle supports the Device Path Protocol.

EFI_HANDLE_TYPE_DEVICE_DRIVER	This bit is set if the handle specified by *DriverBindingHandle* is managing at least one controller or the handle is currently managing the controller specified by *ControllerHandle*.
EFI_HANDLE_TYPE_BUS_DRIVER	This bit is set if the handle specified by *DriverBindingHandle* has produced at least one child controller or the handle produced the child handle specified by *ControllerHandle*.
EFI_HANDLE_TYPE_PARENT_HANDLE	This bit is set if the handle is a device handle, and it is a parent of the controller specified by *ControllerHandle*.
EFI_HANDLE_TYPE_CONTROLLER_ HANDLE	This bit is set if the handle is a device handle that is being managed by the driver specified by *DriverBindingHandle*.
EFI_HANDLE_TYPE_CHILD_HANDLE	This bit is set if the handle is a device handle that was produced by the bus driver specified by *DriverBindingHandle*.

Status Codes Returned

EFI_SUCCESS	The list of handles was returned in *HandleBuffer*, the list of attributes was returned in *HandleType*, and the number of handles was returned in *HandleCount*.
EFI_OUT_OF_RESOURCES	There are not enough resources to return the EFI Driver Model–related information from the handle database.

LibGetManagingDriverBindingHandles()

Summary

Retrieves the set of driver handles that are currently managing a specific controller.

Prototype

```
EFI_STATUS
LibGetManagingDriverBindingHandles (
  EFI_HANDLE  ControllerHandle,
  UINTN       *DriverBindingHandleCount,
  EFI_HANDLE  **DriverBindingHandleBuffer
  );
```

Parameters

ControllerHandle

A handle that represents a controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DriverBindingHandleCount

Returns the number of handles in *DriverBindingHandleBuffer*.

DriverBindingHandleBuffer

Returns the array of handles from the handle database that support the Driver Binding Protocol and are currently managing the controller specified by *ControllerHandle*.

Description

This function retrieves the subset of handles that support the Driver Binding Protocol and are also currently managing the controller handle specified by *ControllerHandle*. The number of managing handles is returned in *DriverBindingHandleCount*, and the array of managing handles is returned in *DriverBindingHandleBuffer*.

Status Codes Returned

EFI_SUCCESS	The list of handles managing *ControllerHandle* was returned in *DriverBindingHandleBuffer*, and the number of driver handles was returned in *DriverBindingHandleCount*.
EFI_OUT_OF_RESOURCES	There are not enough resources to return the EFI Driver Model–related information from the handle database.
EFI_NOT_FOUND	No handles are currently managing *ControllerHandle*.

LibGetParentControllerHandles()

Summary

Retrieves the set of device handles that are parents of a specific controller.

Prototype

```
EFI_STATUS
LibGetParentControllerHandles (
  EFI_HANDLE  ControllerHandle,
  UINTN       *ParentControllerHandleCount,
  EFI_HANDLE  **ParentControllerHandleBuffer
  );
```

Parameters

ControllerHandle

A handle that represents a controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

ParentControllerHandleCount

Returns the number of handles in *ParentControllerHandleBuffer*.

ParentControllerHandleBuffer

Returns the array of device handles from the handle database that are parents of the controller specified by *ControllerHandle*.

Description

This function retrieves the subset of handles that are device handles and are also currently the parents of the controller specified by *ControllerHandle*. The number of parent handles is returned in *ParentControllerHandleCount*, and the array of parent controllers is returned in *ParentControllerHandleBuffer*.

Status Codes Returned

EFI_SUCCESS	The list of parent handles of *ControllerHandle* was returned in *ParentControllerHandleBuffer*, and the number of parent handles was returned in *ParentControllerHandleCount*.
EFI_OUT_OF_RESOURCES	There are not enough resources to return the EFI Driver Model–related information from the handle database.
EFI_NOT_FOUND	No handles are currently parents of *ControllerHandle*.

LibGetChildControllerHandles()

Summary

Retrieves the set of device handles that are children of a specific controller.

Prototype

```
EFI_STATUS
LibGetChildControllerHandles (
  EFI_HANDLE  ControllerHandle,
  UINTN       *ChildControllerHandleCount,
  EFI_HANDLE  **ChildControllerHandleBuffer
  );
```

Parameters

ControllerHandle

A handle that represents a controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

ChildControllerHandleCount

Returns the number of handles in *ChildControllerHandleBuffer*.

ChildControllerHandleBuffer

Returns the array of device handles from the handle database that are children of the controller specified by *ControllerHandle*.

Description

This function retrieves the subset of handles that are device handles and are also currently children of the controller specified by *ControllerHandle*. The number of child handles is returned in *ChildControllerHandleCount*, and the array of child controllers is returned in *ChildControllerHandleBuffer*.

Status Codes Returned

EFI_SUCCESS	The list of children of *ControllerHandle* was returned in *ChildControllerHandleBuffer*, and the number of child handles was returned in *ChildControllerHandleCount*.
EFI_OUT_OF_RESOURCES	There are not enough resources to return the EFI Driver Model–related information from the handle database.
EFI_NOT_FOUND	No handles are currently children of *ControllerHandle*.

LibGetManagedControllerHandles()

Summary

Retrieves the set of device handles that a driver is currently managing.

Prototype

```
EFI_STATUS
LibGetManagedControllerHandles (
  EFI_HANDLE  DriverBindingHandle,
  UINTN       *ControllerHandleCount,
  EFI_HANDLE  **ControllerHandleBuffer
  );
```

Parameters

DriverBindingHandle

A handle that represents a driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

ControllerHandleCount

Returns the number of handles in *ControllerHandleBuffer*.

ControllerHandleBuffer

Returns the array of device handles from the handle database that are currently being managed by *DriverBindingHandle*.

Description

This function retrieves the subset of handles that are device handles and are also currently being managed by the driver specified by *DriverBindingHandle*. The number of device handles is returned in *ControllerHandleCount*, and the array of controller handles is returned in *ControllerHandleBuffer*.

Status Codes Returned

EFI_SUCCESS	The list of controller handles being managed by *DriverBindingHandle* was returned in *ControllerHandleBuffer*, and the number of handles was returned in *ControllerHandleCount*.
EFI_OUT_OF_RESOURCES	There are not enough resources to return the EFI Driver Model–related information from the handle database.
EFI_NOT_FOUND	No controller handles are currently being managed by *DriverBindingHandle*.

LibGetManagedChildControllerHandles()

Summary

Retrieves the set of device handles that a driver produced as children of a specific controller.

Prototype

```
EFI_STATUS
LibGetManagedControllerHandles (
  EFI_HANDLE  DriverBindingHandle,
  EFI_HANDLE  ControllerHandle,
  UINTN       *ChildControllerHandleCount,
  EFI_HANDLE  **ChildControllerHandleBuffer
  );
```

Parameters

DriverBindingHandle

A handle that represents a driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

ControllerHandle

A handle that represents a controller.

ChildControllerHandleCount

Returns the number of handles in *ChildControllerHandleBuffer*.

ChildControllerHandleBuffer

Returns the array of child device handles from the handle database that were produced by the driver specified by *DriverBindingHandle* and that are children of the controller specified by *ControllerHandle*.

Description

This function retrieves the subset of handles that are child handles that were produced by the driver specified by *DriverBindingHandle* and that are children of the controller specified by *ControllerHandle*. The number of child handles is returned in *ChildControllerHandleCount*, and the array of child handles is returned in *ChildControllerHandleBuffer*.

Status Codes Returned

EFI_SUCCESS	The list of child handles that were produced by *DriverBindingHandle* and that are children of *ControllerHandle* was returned in *ControllerHandleBuffer*, and the number of child handles was returned in *ControllerHandleCount*.
EFI_OUT_OF_RESOURCES	There are not enough resources to return the EFI Driver Model–related information from the handle database.
EFI_NOT_FOUND	No child handles were produced by *DriverBindingHandle* and that are currently children of *ControllerHandle*.

4.10 File I/O Support Functions

Table 4-21 lists some helper function related to files and a set of functions and macros that facilitate the manipulating of files.

Table 4-21. 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 Master Boot Record (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()`**.

LibOpenRoot()

Summary

Opens and returns a file handle to the root directory of a volume.

Prototype

```
EFI_FILE_HANDLE
LibOpenRoot (
  IN EFI_HANDLE           DeviceHandle
  );
```

Parameters

DeviceHandle

A handle for a device. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function looks for a *FileSystemProtocol* that is attached to *DeviceHandle*. If one is found, then an attempt is made to open a volume on that device. If this attempt succeeds, then a valid file handle is returned. Otherwise, **NULL** is returned.

LibFileInfo()

Summary

Gets the file information from an open file descriptor and stores it in a buffer allocated from pool.

Prototype

```
EFI_FILE_INFO *
LibFileInfo (
  IN EFI_FILE_HANDLE      FHand
  );
```

Parameters

FHand

A file handle.

Description

This function retrieves the **EFI_FILE_INFO** data structure for the file handle *Fhand* and stores it in a buffer allocated from pool. A pointer to this buffer is returned. If the file information cannot be retrieved or if there is not enough memory in pool to store the data structure, **NULL** will be returned.

LibFileSystemInfo()

Summary

Gets the file system information from an open file descriptor and stores it in a buffer allocated from pool.

Prototype

```
EFI_FILE_SYSTEM_INFO *
LibFileSystemInfo (
  IN EFI_FILE_HANDLE      FHand
  );
```

Parameters

FHand

A file handle.

Description

This function retrieves the **EFI_FILE_SYSTEM_INFO** data structure for the file handle *Fhand* and stores it in a buffer allocated from pool. A pointer to this buffer is returned. If the file information cannot be retrieved or if there is not enough memory in pool to store the data structure, **NULL** will be returned.

LibFileSystemVolumeLabelInfo()

Summary

Gets the file system information from an open file descriptor and stores it in a buffer allocated from pool.

Prototype

```
EFI_FILE_SYSTEM_VOLUME_LABEL_INFO *
LibFileSystemVolumeLabelInfo (
  IN EFI_FILE_HANDLE      FHand
  );
```

Parameters

FHand

A file handle.

Description

This function retrieves the 11-byte **EFI_FILE_SYSTEM_LABEL_INFO** data structure for the file handle *FHand* and stores it in a buffer allocated from pool. A pointer to this buffer is returned. If the file information cannot be retrieved or if there is not enough memory in pool to store the data structure, **NULL** will be returned.

ValidMBR()

Summary

Determines if a hard drive's Master Boot Record (MBR) is valid.

Prototype

```
BOOLEAN
ValidMBR(
  IN  MASTER_BOOT_RECORD  *Mbr,
  IN  EFI_BLOCK_IO        *BlkIo
  );
```

Parameters

Mbr

A pointer to a hard drive's Master Boot Record.

BlkIo

A pointer to a Block I/O Protocol handle. Type **EFI_BLOCK_IO** is defined in section 11.6 in the *EFI 1.10 Specification.*

Description

This function verifies that the layout of partitions that are described in the MBR are valid. The MBR is in the buffer pointed to by *Mbr*. Additional information about the physical disk is contained in *BlkIo*. The size of the partitions are compared to the size of the physical drive, and checks are also made for overlapping partitions. If the MBR is valid, then **TRUE** is returned. Otherwise, **FALSE** is returned.

Status Codes Returned

TRUE	The Master Boot Record is valid.
FALSE	The Master Boot Record is not valid.

OpenSimpleReadFile()

Summary

Opens a file from several possible sources and returns a file handle.

Prototype

```
EFI_STATUS
OpenSimpleReadFile (
  IN BOOLEAN                  BootPolicy,
  IN VOID                     *SourceBuffer   OPTIONAL,
  IN UINTN                    SourceSize,
  IN OUT EFI_DEVICE_PATH      **FilePath,
  OUT EFI_HANDLE              *DeviceHandle,
  OUT SIMPLE_READ_FILE        *SimpleReadHandle
  );
```

Parameters

BootPolicy

If **TRUE**, indicates that the request originates from the boot manager and that the boot manager is attempting to load *FilePath* as a boot selection.

SourceBuffer

A pointer to a buffer containing the file.

SourceSize

The size of the buffer containing the file to access.

FilePath

Pointer to the device-specific path of the file to load. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

DeviceHandle

Pointer to the device handle of the device to open. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

SimpleReadHandle

A pointer to the file handle to return. Type **SIMPLE_READ_FILE** is defined in “Related Definitions” below.

Description

This function opens a file from one of three possible sources and returns a file handle. The first source is a file on a device through the file system interface. The second source is through a file on a device through the load file interface, and the third source is from a buffer in memory. If the optional parameter *SourceBuffer* is not **NULL**, then it is assumed that the file is in a buffer in memory and a file handle for this file is returned in *SimpleReadHandle*. If the root of the device specified by *DeviceHandle* can be opened and *FilePath* is a valid file path on the device, then the file specified by the combination of *DeviceHandle* and *FilePath* is opened and a file handle is returned in *SimpleReadHandle*. If access to the file is not allowed through the file system interface, then an attempt is made to open the file through the load file interface. If this succeeds, then a copy of the file is loaded into memory and a file handle is returned in *SimpleReadHandle*.

Related Definitions

```
//*******************************************
// SIMPLE_READ_FILE
//*******************************************
typedef VOID        *SIMPLE_READ_FILE;
```

Status Codes Returned

EFI_SUCCESS	The file was opened and a valid file handle was returned.
EFI_OUT_OF_RESOURCES	The file handle could not be allocated from memory.
EFI_UNSUPPORTED	The Load File Protocol is not supported from this file.
EFI_BUFFER_TOO_SMALL	A buffer for the file could not be allocated.
EFI_NO_MEDIA	No media was present to load the file.
EFI_DEVICE_ERROR	The file was not loaded due to a device error.
EFI_NO_RESPONSE	The remote system did not respond.
EFI_NOT_FOUND	The file was not found.

ReadSimpleReadFile()

Summary

Reads data from a file that was opened with **OpenSimpleReadFile()**.

Prototype

```
EFI_STATUS
ReadSimpleReadFile (
  IN SIMPLE_READ_FILE     SimpleReadHandle,
  IN UINTN                Offset,
  IN OUT UINTN            *ReadSize,
  OUT VOID                *Buffer
  );
```

Parameters

SimpleReadHandle

A file handle. Type **SIMPLE_READ_FILE** is defined in **OpenSimpleReadFile()**.

Offset

Offset in bytes within the file to begin the read operation.

ReadSize

A pointer to the number of bytes to read from the file.

Buffer

A pointer to the buffer to store the read data.

Description

This function reads data from the file that is specified by the file handle *SimpleReadHandle*. If the file handle describes a file image in memory, then a memory copy is performed to copy the read data into *Buffer*. Otherwise, a file system read call is made to read the data from a device into *Buffer*. If *Offset* is beyond the end of the file, then *ReadSize* is set to zero and an error is returned. Otherwise, *ReadSize* will be set to the number of bytes that was actually read from the device.

Status Codes Returned

EFI_SUCCESS	The data was read.
EFI_NO_MEDIA	No media was present to load the file.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_BUFFER_TOO_SMALL	*ReadSize* is too small to read the current file. *ReadSize* has been updated with the size needed to complete the request.

CloseSimpleReadFile()

Summary

Closes a file that was opened with **OpenSimpleReadFile()**.

Prototype

```
VOID
CloseSimpleReadFile (
  IN SIMPLE_READ_FILE     SimpleReadHandle
  );
```

Parameters

SimpleReadHandle

A file handle. Type **SIMPLE_READ_FILE** is defined in **OpenSimpleReadFile()**.

Description

This function closes the file that is specified by *SimpleReadHandle* and frees the memory used by *SimpleReadHandle*. If any data buffers were allocated when *SimpleReadHandle* was opened, then those buffers are also freed.

4.11 Device Path Support Functions

Table 4-22 lists the support functions for creating and maintaining device path data structures. These functions are described in the following sections.

Table 4-22. 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.
FileDevicePath()	Appends a file path to a device path.
DevicePathSize()	Returns the size of a device path in bytes.
DuplicateDevicePath()	Creates a new copy of a device path.
LibDevicePathToInterface()	Retrieves a protocol interface for a device.
UnpackDevicePath()	Naturally aligns all the nodes in a device path.
LibMatchDevicePaths()	Reports membership of a single-instance device path in a possible multiple-instance device path.
LibDuplicateDevicePathInstance()	Creates a second corresponding instance of a given device path.

DevicePathFromHandle()

Summary

Retrieves the device path for the specified handle.

Prototype

```
EFI_DEVICE_PATH *
DevicePathFromHandle (
  IN EFI_HANDLE           Handle
  );
```

Parameters

Handle

A handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function retrieves the device path for a handle specified by *Handle*. If *Handle* is valid, then a pointer to the device path is returned. If *Handle* is not valid, then **NULL** is returned.

DevicePathInstance()

Summary

Retrieves the next device path instance from a device path data structure.

Prototype

```
EFI_DEVICE_PATH *
DevicePathInstance (
  IN OUT EFI_DEVICE_PATH  **DevicePath,
  OUT UINTN               *Size
  );
```

Parameters

DevicePath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Size

A pointer to the size of a device path instance in bytes.

Description

This function is used to parse device path instances from the device path *DevicePath*. This function returns a pointer to the current device path instance. In addition, it returns the size in bytes of the current device path instance in *Size* and a pointer to the next device path instance in *DevicePath*. If there are no more device path instances in *DevicePath*, then *DevicePath* will be set to **NULL**.

DevicePathInstanceCount()

Summary

Determines the number of device path instances that exist in a device path.

Prototype

```
UINTN
DevicePathInstanceCount (
  IN EFI_DEVICE_PATH      *DevicePath
  );
```

Parameters

DevicePath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Description

This function counts and returns the number of device path instances in *DevicePath*.

AppendDevicePath()

Summary

Appends a device path to all the instances in another device path.

Prototype

```
EFI_DEVICE_PATH *
AppendDevicePath (
  IN EFI_DEVICE_PATH      *Src1,
  IN EFI_DEVICE_PATH      *Src2
  );
```

Parameters

Src1

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Src2

A pointer to a device path data structure.

Description

This function appends the device path *Src2* to every device path instance in *Src1*. A pointer to the new device path is returned. **NULL** is returned if space for the new device path could not be allocated from pool. It is up to the caller to free the memory used by *Src1* and *Src2* if they are no longer needed.

AppendDevicePathNode()

Summary

Appends a device path node to all the instances in another device path.

Prototype

```
EFI_DEVICE_PATH *
AppendDevicePathNode (
  IN EFI_DEVICE_PATH      *Src1,
  IN EFI_DEVICE_PATH      *Src2
  );
```

Parameters

Src1

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Src2

A pointer to a single device path node.

Description

This function appends the device path node *Src2* to every device path instance in *Src1*. This function returns a pointer to the new device path. If there is not enough temporary pool memory available to complete this function, then **NULL** is returned. It is up to the caller to free the memory used by *Src1* and *Src2* if they are no longer needed.

AppendDevicePathInstance()

Summary

Adds a device path instance to a device path.

Prototype

```
EFI_DEVICE_PATH *
AppendDevicePathInstance (
  IN EFI_DEVICE_PATH      *Src,
  IN EFI_DEVICE_PATH      *Instance
  );
```

Parameters

Src

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Instance

A pointer to a device path instance.

Description

This function appends the device path instance *Instance* to the device path *Src*. If *Src* is **NULL**, then a new device path with one instance is created. This function returns a pointer to the new device path.. If there is not enough temporary pool memory available to complete this function, then **NULL** is returned. It is up to the caller to free the memory used by *Src* and *Instance* if they are no longer needed.

FileDevicePath()

Summary

Allocates a device path for a file and appends it to an existing device path.

Prototype

```
EFI_DEVICE_PATH *
FileDevicePath (
  IN EFI_HANDLE           Device  OPTIONAL,
  IN CHAR16               *FileName
  );
```

Parameters

Device

A pointer to a device handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

FileName

A pointer to a **NULL**-terminated Unicode string.

Description

If *Device* is a valid device handle, then a device path for the file specified by *FileName* is allocated and appended to the device path associated with the handle *Device*. If *Device* is not a valid device handle, then a device path for the file specified by *FileName* is allocated and returned.

DevicePathSize()

Summary

Returns the size of a device path in bytes.

Prototype

```
UINTN
DevicePathSize (
  IN EFI_DEVICE_PATH      *DevPath
  );
```

Parameters

DevPath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Description

This function determines the size of a data path data structure in bytes. This size is returned.

DuplicateDevicePath()

Summary

Creates a duplicate copy of an existing device path.

Prototype

```
EFI_DEVICE_PATH *
DuplicateDevicePath (
  IN EFI_DEVICE_PATH      *DevPath
  );
```

Parameters

DevPath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Description

This function allocates space for a new copy of the device path *DevPath*. If the memory is successfully allocated, then the contents of *DevPath* are copied to the newly allocated buffer and a pointer to that buffer is returned. Otherwise, **NULL** is returned.

LibDevicePathToInterface()

Summary

Retrieves a protocol interface for a device.

Prototype

```
EFI_STATUS
LibDevicePathToInterface (
  IN EFI_GUID              *Protocol,
  IN EFI_DEVICE_PATH       *FilePath,
  OUT VOID                 **Interface
  );
```

Parameters

Protocol

The published unique identifier of the protocol. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

FilePath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Interface

Supplies an address where a pointer to the requested protocol interface is returned.

Description

This function finds all the devices that support the interface protocol specified by *Protocol*. It then searches that list of devices for the one that best matches the device path specified by *FilePath*. If a match is found, then the protocol interface of that device is returned in *Interface*. Otherwise, *Interface* is set to **NULL**.

Status Codes Returned

EFI_SUCCESS	A matching protocol interface was found.
EFI_NOT_FOUND	A matching protocol interface was not found.
EFI_UNSUPPORTED	The device does not support the requested protocol.
EFI_INVALID_PARAMETER	*FilePath* contains more than one device path instance.

UnpackDevicePath()

Summary

Unpacks a device path data structure so that all the nodes of a device path are naturally aligned.

Prototype

```
EFI_DEVICE_PATH *
UnpackDevicePath (
  IN EFI_DEVICE_PATH      *DevPath
  );
```

Parameters

DevPath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Description

This function allocates space for a new copy of the device path *DevPath*. The new copy of *DevPath* is modified so that every node of the device path is naturally aligned. If the memory for the device path is successfully allocated, then a pointer to the new device path is returned. Otherwise, **NULL** is returned.

LibMatchDevicePaths()

Summary

Compares a device path data structure to that of all the nodes of a second device path instance.

Prototype

```
BOOLEAN
LibMatchDevicePaths (
  IN EFI_DEVICE_PATH      *Multi,
  IN EFI_DEVICE_PATH      *Single
  );
```

Parameters

Multi

A pointer to a multiple-instance device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Single

A pointer to a single-instance device path data structure.

Description

This function compares the *Single* instance device path against the various device path instances in *Multi*. The function returns **TRUE** if the *Single* is contained within *Multi*. Otherwise, **FALSE** is returned.

Status Codes Returned

TRUE	*Single* was found in *Multi*.
FALSE	*Single* was not found in *Multi*.

LibDuplicateDevicePathInstance()

Summary

Creates a device path data structure that identically matches the device path that is passed in.

Prototype

```
EFI_DEVICE_PATH *
LibDuplicateDevicePathInstance (
  IN EFI_DEVICE_PATH      *DevPath
  );
```

Parameters

DevPath

A pointer to a device path data structure. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Description

This function allocates space for a new copy of the device path *DevPath*. The new copy of *DevPath* is created to identically match the input. Otherwise, **NULL** is returned.

4.12 PCI Functions and Macros

Table 4-23 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 4-23. 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.
Outpd()	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.

PciFindDeviceClass()

Summary

Finds the first PCI device with the specified class.

Prototype

```
EFI_STATUS
PciFindDeviceClass (
  IN  OUT EFI_PCI_ADDRESS_UNION   *Address,
  IN      UINT8                   BaseClass,
  IN      UINT8                   SubClass
  );
```

Parameters

Address

A pointer to the data structure containing the bus, device, and function of the PCI device that matches the specified class. Type **EFI_PCI_ADDRESS_UNION** is defined in "Related Definitions" below.

BaseClass

The PCI base class of the device for which to search.

SubClass

The PCI subclass of the device for which to search.

Description

This function searches all the PCI buses for a device with a matching *BaseClass* and *SubClass* in the device's standard PCI header. If a matching device is found, the device's PCI bus number, PCI device number, and PCI function number are returned in *Address*.

Related Definitions

```
//*****************************************
// EFI_PCI_ADDRESS_UNION
//*****************************************
typedef union {
  UINT64          Address;
  EFI_ADDRESS     EfiAddress;
} EFI_PCI_ADDRESS_UNION;
```

Address

64-bit PCI address.

EfiAddress

64-bit PCI address of **EFI_ADDRESS**. Type **EFI_ADDRESS** is defined below.

```
//*****************************************
// EFI_ADDRESS
//*****************************************
typedef struct {
  UINT8   Register;
  UINT8   Function;
  UINT8   Device;
  UINT8   Bus;
  UINT32  Reserved;
} EFI_ADDRESS;
```

Register

Register number of a PCI device.

Function

Function number of a PCI device.

Device

Device number of a PCI device.

Bus

Bus number of a PCI device.

Reserved

Not used. Reserved for future extension.

Status Codes Returned

EFI_SUCCESS	A corresponding PCI device was found.
EFI_NOT_FOUND	A corresponding PCI device was not found.

PciFindDevice()

Summary

Finds the first PCI device with the specified device ID and vendor ID.

Prototype

```
EFI_STATUS
PciFindDevice (
  IN  OUT EFI_PCI_ADDRESS_UNION   *DeviceAddress,
  IN      UINT16                  VendorId,
  IN      UINT16                  DeviceId,
  IN OUT  PCI_TYPE00              *Pci
  )
```

Parameters

DeviceAddress

A pointer to the data structure containing the bus, device, and function of the PCI device that matches the specified class. Type **EFI_PCI_ADDRESS_UNION** is defined in **PciFindDeviceClass()**.

VendorId

The PCI base class of the device for which to search.

DeviceId

The PCI subclass of the device for which to search.

Pci

A pointer to the configuration space header of the device. Type **PCI_TYPE00** is defined in “Related Definitions” below.

Description

This function searches all the PCI buses for a device with a matching *VendorId* and *DeviceId* in the device’s standard PCI header. If a matching device is found, the device’s PCI bus number, PCI device number, and PCI function number are returned in *Address*, as is the Type 0 configuration space returned in *Pci*. If the device cannot be discovered, **EFI_NOT_FOUND** is returned.

Related Definitions

```
//*********************************************
// PCI_TYPE00
//*********************************************
typedef struct {
  PCI_DEVICE_INDEPENDENT_REGION   Hdr;
  PCI_DEVICE_HEADER_TYPE_REGION   Device;
} PCI_TYPE00;
```

Hdr

Contains device-independent information in a configuration space. Type **PCI_DEVICE_INDEPENDENT_REGION** is defined below.

Device

Contains header information in a configuration space. Type **PCI_DEVICE_HEADER_TYPE_REGION** is defined below.

```
//*********************************************
// PCI_DEVICE_INDEPENDENT_REGION
//*********************************************
typedef struct {
  UINT16      VendorId;
  UINT16      DeviceId;
  UINT16      Command;
  UINT16      Status;
  UINT8       RevisionID;
  UINT8       ClassCode[3];
  UINT8       CacheLineSize;
  UINT8       LatencyTimer;
  UINT8       HeaderType;
  UINT8       BIST;
} PCI_DEVICE_INDEPENDENT_REGION;
```

VendorId

Identifies the manufacturer of the device.

DeviceId

Identifies the particular device.

Command

Provides coarse control over a device's ability to generate and respond to PCI cycles.

Status

This register is used to record status information for PCI-bus-related events.

RevisionID

Specifies a device-specific revision identifier.

ClassCode

Identifies the generic function of the device.

CacheLineSize

Specifies the system cache line size in units of 32-bit words.

LatencyTimer

Specifies, in units of PCI bus clocks, the value of the latency timer for this PCI bus master.

HeaderType

Identifies the layout of the second part of the predefined header (beginning at byte 10h in the PCI configuration space) and also whether the device contains multiple functions.

BIST

Specifies Built-in Self Test (BIST) information.

```
//*******************************************
// PCI_DEVICE_HEADER_TYPE_REGION
//*******************************************
typedef struct {
  UINT32      Bar[6];
  UINT32      CISPtr;
  UINT16      SubsystemVendorID;
  UINT16      SubsystemID;
  UINT32      ExpansionRomBar;
  UINT32      Reserved[2];
  UINT8       InterruptLine;
  UINT8       InterruptPin;
  UINT8       MinGnt;
  UINT8       MaxLat;
} PCI_DEVICE_HEADER_TYPE_REGION;
```

Bar

Base address registers.

CISPtr

This optional register is used by those devices that want to share silicon between CardBus* and PCI. The field is used to point to the Card Information Structure (CIS) for the CardBus card.

SubsystemVendorID

Identifies the vendor of add-in board or subsystem where the PCI device resides.

SubsystemID

Identifies the add-in board or subsystem where the PCI device resides.

ExpansionRomBar

This register is defined to handle the base address and size information for expansion ROM.

Reserved

Not used. Reserved for future extension.

InterruptLine

An 8-bit register that is used to communicate interrupt line routing information.

InterruptPin

Indicates which interrupt pin the device (or device function) uses.

MinGnt

Specifies the length of the burst period that is needed by the device, assuming a clock rate of 33 MHz.

MaxLat

Specifies how often the device needs to gain access to the PCI bus.

Status Codes Returned

EFI_SUCCESS	A corresponding PCI device was found.
EFI_NOT_FOUND	A corresponding PCI device was not found.

InitializeGlobalIoDevice()

Summary

Returns a Device I/O Protocol instance that is supported by the given device.

Prototype

```
EFI_STATUS
InitializeGlobalIoDevice (
  IN EFI_DEVICE_PATH              *DevicePath,
  IN EFI_GUID                     *Protocol,
  IN CHAR8                        *ErrorStr,
  OUT EFI_DEVICE_IO_INTERFACE     **GlobalIoFncs
  );
```

Parameters

DevicePath

A pointer to a device path. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

Protocol

The protocol that a device driver is attempting to register for this device. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

ErrorStr

Error message to display if the device specified by *DevicePath* already supports *Protocol*.

GlobalIoFncs

A pointer to the Device I/O Protocol instance that is supported by the device specified by *DevicePath*. Type **EFI_DEVICE_IO_INTERFACE** is defined in section 18.2 in the *EFI 1.10 Specification.*

Description

This function checks to see if the device specified by *DevicePath* already supports *Protocol*. If it does, then an error message is displayed using *ErrorStr*. If the device specified by *DevicePath* does not support *Protocol*, then a check is made to see if the device specified by *DevicePath* supports the Device I/O Protocol. If it does, then the Device I/O Protocol instance is returned in *GlobalIoFncs*.

Status Codes Returned

EFI_SUCCESS	A Device I/O Protocol instance was returned..
EFI_LOAD_ERROR	The device already supports *Protocol*.
EFI_NOT_FOUND	A Device I/O Protocol instance was not found.

ReadPort()

Summary

Reads an I/O port using a Device I/O Protocol instance.

Prototype

```
UINT32
ReadPort (
  IN EFI_DEVICE_IO_INTERFACE    *GlobalIoFncs,
  IN EFI_IO_WIDTH               Width,
  IN UINTN                      Port
  );
```

Parameters

GlobalIoFncs

The Device I/O Protocol instance to use to perform the I/O read. Type **EFI_DEVICE_IO_INTERFACE** is defined in section 18.2 in the *EFI 1.10 Specification.*

Width

The width of the I/O read operation. Type **EFI_IO_WIDTH** is defined in the **DEVICE_IO** Protocol definition in the *EFI 1.10 Specification.*

Port

The address of the I/O read operation.

Description

This function reads the I/O port specified by *Port* and *Width* using the protocol interface functions in *GlobalIoFncs*. The data returned by the I/O read operation is returned.

WritePort()

Summary

The **WritePort()** function writes to an I/O port using a Device I/O Protocol instance.

Prototype

```
UINT32
WritePort (
  IN EFI_DEVICE_IO_INTERFACE    *GlobalIoFncs,
  IN EFI_IO_WIDTH               Width,
  IN UINTN                      Port,
  IN UINTN                      Data
);
```

Parameters

GlobalIoFncs

The Device I/O Protocol instance to use to perform the I/O write. Type **EFI_DEVICE_IO_INTERFACE** is defined in section 18.2 in the *EFI 1.10 Specification.*

Width

The width of the I/O write operation. Type **EFI_IO_WIDTH** is defined in the **DEVICE_IO** Protocol definition in the *EFI 1.10 Specification.*

Port

The address of the I/O write operation.

Data

The data to use for the I/O write operation.

Description

This function writes *Data* to the I/O port specified by *Port* and *Width* using the protocol interface functions in *GlobalIoFncs*. *Data* is returned.

ReadPciConfig()

Summary

Reads from PCI configuration space using a Device I/O Protocol instance.

Prototype

```
UINT32
ReadPciConfig (
  IN EFI_DEVICE_IO_INTERFACE    *GlobalIoFncs,
  IN EFI_IO_WIDTH               Width,
  IN UINTN                      Port
  );
```

Parameters

GlobalIoFncs

The Device I/O Protocol instance to use to perform the PCI configuration read. Type **EFI_DEVICE_IO_INTERFACE** is defined in section 18.2 in the *EFI 1.10 Specification.*

Width

The width of the PCI configuration read operation. Type **EFI_IO_WIDTH** is defined in the **DEVICE_IO** Protocol definition in the *EFI 1.10 Specification.*

Port

The address of the PCI configuration read operation.

Description

This function reads from PCI configuration space at the address specified by *Port* and *Width* using the protocol interface functions in *GlobalIoFncs*. The data returned by the PCI configuration read operation is returned.

WritePciConfig()

Summary

Writes to PCI configuration space using a Device I/O Protocol instance.

Prototype

```
UINT32
WritePciConfig (
  IN EFI_DEVICE_IO_INTERFACE    *GlobalIoFncs,
  IN EFI_IO_WIDTH               Width,
  IN UINTN                      Port,
  IN UINTN                      Data
);
```

Parameters

GlobalIoFncs

The Device I/O Protocol instance to use to write to PCI configuration space. Type **EFI_DEVICE_IO_INTERFACE** is defined in section 18.2 in the *EFI 1.10 Specification.*

Width

The width of the PCI configuration write operation. Type **EFI_IO_WIDTH** is defined in the **DEVICE_IO** Protocol definition in the *EFI 1.10 Specification.*

Port

The address of the PCI configuration write operation.

Data

The data to use for the PCI configuration write operation.

Description

This function writes *Data* to the PCI configuration space at the address specified by *Port* and *Width*, using the protocol interface functions in *GlobalIoFncs*. *Data* is returned.

inp()

Summary

Reads an 8-bit value from an I/O port using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
UINT8
inp (
  IN UINTN    Port
);
```

Parameters

Port

The address of the I/O read operation.

Description

This function reads an 8-bit value from the I/O port specified by *Port*.

outp()

Summary

Writes an 8-bit value to an I/O port using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
VOID
outp (
  IN UINTN    Port,
  IN UINT8    Data
);
```

Parameters

Port

The address of the I/O write operation.

Data

The 8-bit value to write.

Description

This function writes the 8-bit value *Data* to the I/O port specified by *Port*.

inpw()

Summary

Reads a 16-bit value from an I/O port using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
UINT16
inpw (
  IN UINTN    Port
);
```

Parameters

Port

The address of the I/O read operation.

Description

This function reads a 16-bit value from the I/O port specified by *Port*.

outpw()

Summary

Writes a 16-bit value to an I/O port using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
VOID
outpw (
  IN UINTN    Port,
  IN UINT16   Data
);
```

Parameters

Port

The address of the I/O write operation.

Data

The 16-bit value to write.

Description

This function writes the 16-bit value *Data* to the I/O port specified by *Port*.

inpd()

Summary

Reads a 32-bit value from an I/O port using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
UINT32
inpd (
  IN UINTN    Port
);
```

Parameters

Port

The address of the I/O read operation.

Description

This function reads a 32-bit value from the I/O port specified by *Port*.

outpd()

Summary

Writes a 32-bit value to an I/O port using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
VOID
outpd (
  IN UINTN    Port,
  IN UINT32   Data
);
```

Parameters

Port

The address of the I/O write operation.

Data

The 32-bit value to write.

Description

This function writes 32-bit value *Data* to the I/O port specified by *Port*.

readpci8()

Summary

Reads an 8-bit value from the PCI configuration space using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
UINT8
readpci8 (
  IN UINTN    Port
);
```

Parameters

Port

The address of the PCI configuration read operation.

Description

This function reads an 8-bit value from the PCI configuration space at the address specified by *Port*.

writepci8()

Summary

Writes an 8-bit value to the PCI configuration space using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
VOID
writepci8 (
  IN UINTN    Port,
  IN UINT8    Data
);
```

Parameters

Port

The address of the PCI configuration write operation.

Data

The 8-bit value to write.

Description

This function writes the 8-bit value *Data* to the PCI configuration space at the address specified by *Port*.

readpci16()

Summary

Reads a 16-bit value from the PCI configuration space using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
UINT8
readpci16 (
  IN UINTN    Port
);
```

Parameters

Port

The address of the PCI configuration read operation.

Description

This function reads a 16-bit value from the PCI configuration space at the address specified by *Port*.

writepci16()

Summary

Writes a 16-bit value to the PCI configuration space using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
VOID
writepci16 (
  IN UINTN    Port,
  IN UINT8    Data
);
```

Parameters

Port

The address of the PCI configuration write operation.

Data

The 16-bit value to write.

Description

This function writes the 16-bit value *Data* to the PCI configuration space at the address specified by *Port*.

readpci32()

Summary

Reads a 32-bit value from the PCI configuration space using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
UINT8
readpci32 (
  IN UINTN    Port
);
```

Parameters

Port

The address of the PCI configuration read operation.

Description

This function reads a 32-bit value from the PCI configuration space at the address specified by *Port*.

writepci32()

Summary

Writes a 32-bit value to the PCI configuration space using the Device I/O Protocol instance **GlobalIoFncs**.

Prototype

```
VOID
writepci32 (
  IN UINTN    Port,
  IN UINT8    Data
);
```

Parameters

Port

The address of the PCI configuration write operation.

Data

The 32-bit value to write.

Description

This function writes the 32-bit value *Data* to the PCI configuration space at the address specified by *Port*.

4.13 Miscellaneous Functions and Macros

Table 4-24 lists some miscellaneous helper functions that are described in the following sections.

Table 4-24. 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 Code 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.
RtLibEnableVirtualMappings()	Converts internal library pointers to virtual runtime pointers.
RtConvertList()	Converts pointers in a linked list to virtual runtime pointers.
LibGetSystemConfigurationTable()	Retrieves a system configuration table from the EFI System Table.

LibGetVariable()

Summary

Returns the value of the specified variable.

Prototype

```
VOID *
LibGetVariable (
  IN CHAR16                *Name,
  IN EFI_GUID              *VendorGuid
  );
```

Parameters

Name

A **NULL**-terminated Unicode string that is the name of the vendor's variable.

VendorGuid

A unique identifier for the vendor. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function retrieves the value of the variable specified by *Name* and *VendorGuid*. If the variable exists, space for storing the variable's value is allocated from pool and a pointer to the variable's value is returned. Otherwise, **NULL** is returned.

LibGetVariableAndSize()

Summary

Returns the value of the specified variable and its size in bytes.

Prototype

```
VOID *
LibGetVariableAndSize (
  IN CHAR16                  *Name,
  IN EFI_GUID                *VendorGuid,
  OUT UINTN                  *VarSize
  );
```

Parameters

Name

A **NULL**-terminated Unicode string that is the name of the vendor's variable.

VendorGuid

A unique identifier for the vendor. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

VarSize

The size of the returned environment variable in bytes.

Description

This function retrieves the value of the variable specified by *Name* and *VendorGuid*. If the variable exists, space for storing the variable's value is allocated from pool, and a pointer to the variable's value is returned. Otherwise, **NULL** is returned. The size of the variable's value is returned in *VarSize*.

LibDeleteVariable()

Summary

Returns the value of the specified variable and its size in bytes.

Prototype

```
VOID *
LibDeleteVariable (
  IN CHAR16                 *VarName,
  IN EFI_GUID               *VarGuid
);
```

Parameters

VarName

A **NULL**-terminated Unicode string that is the name of the vendor's variable.

VarGuid

A unique identifier for the vendor. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function deletes the variable specified by *Name* and *VendorGuid*. If the variable exists, space for storing the variable's value is allocated from pool and a zero value is written.

Status Codes Returned

EFI_SUCCESS	The variable was found and removed.
EFI_UNSUPPORTED	The variable store was inaccessible.
EFI_OUT_OF_RESOURCES	The temporary buffer was not available.
EFI_NOT_FOUND	The variable was not found.

CompareGuid()

Summary

Compares two GUIDs.

Prototype

```
INTN
CompareGuid(
  IN EFI_GUID     *Guid1,
  IN EFI_GUID     *Guid2
  );
```

Parameters

Guid1

A pointer to a 128-bit GUID. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Guid2

A pointer to a 128-bit GUID.

Description

This function compares two128-bit GUIDs. If the GUIDs are identical then 0 is returned. If there are any bit differences in the two GUIDs, a nonzero value is returned.

Status Codes Returned

0	The two GUIDs are identical.
≠ 0	The two GUIDs are not identical.

CR()

Summary

Returns a pointer to an element's containing record .

Prototype

```
TYPE *
CR(
  VOID    *Record,
          TYPE,
          Field,
  UINTN   Signature
  );
```

Parameters

Record

A pointer to a field within the containing record.

TYPE

The name of the containing record's data structure.

Field

The name of the field from the containing record to which *Record* points.

Signature

The signature for the containing record's data structure.

Description

This macro returns a pointer to a data structure from one of the data structure's elements.

DecimaltoBCD()

Summary

Converts a decimal value to a Binary Coded Decimal (BCD) value.

Prototype

```
UINT8
DecimaltoBCD(
  IN  UINT8 DecValue
  );
```

Parameters

DecValue

An 8-bit decimal value.

Description

This function converts an 8-bit decimal value to an 8-bit BCD value and returns the BCD value.

BCDtoDecimal()

Summary

Converts a BCD value to a decimal value.

Prototype

```
UINT8
BCDtoDecimal(
  IN  UINT8 BcdValue
  );
```

Parameters

BcdValue

An 8-bit BCD value.

Description

This function converts an 8-bit BCD value to an 8-bit decimal value and returns the decimal value.

LibCreateProtocolNotifyEvent()

Summary

Creates a notification event and registers that event with all the protocol instances specified by *ProtocolGuid*.

Prototype

```
EFI_EVENT
LibCreateProtocolNotifyEvent(
  IN EFI_GUID              *ProtocolGuid,
  IN EFI_TPL                NotifyTpl,
  IN EFI_EVENT_NOTIFY       NotifyFunction,
  IN VOID                  *NotifyContext,
  OUT VOID                 *Registration
  );
```

Parameters

ProtocolGuid

Supplies the GUID of the protocol upon whose installation the event is fired. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

NotifyTpl

Supplies the task priority level of the event notifications. Type **EFI_TPL** is defined in **RaiseTpl()** in the *EFI 1.10 Specification.*

NotifyFunction

Supplies the function to notify when the event is signaled. Type **EFI_EVENT_NOTIFY** is defined in **CreateEvent()** in the *EFI 1.10 Specification.*

NotifyContext

The context parameter to pass to *NotifyFunction*.

Registration

A pointer to a memory location to receive the registration value. This value is passed to **LocateHandle()** to obtain new handles that have been added that support the *ProtocolGuid*-specified protocol.

Description

This function causes the notification function to be executed for every instance of a protocol of type *ProtocolGuid* that exists in the system when this function is invoked. In addition, every time an instance of a protocol of type *ProtocolGuid* is added, the notification function is also executed. This function returns the notification event that was created.

WaitForSingleEvent()

Summary

Waits for a given event to fire or for an optional timeout to expire.

Prototype

```
EFI_STATUS
WaitForSingleEvent(
  IN EFI_EVENT            Event,
  IN UINT64               Timeout OPTIONAL
  );
```

Parameters

Event

The event for which to wait. Type **EFI_EVENT** is defined in **CreateEvent()** in the *EFI 1.10 Specification.*

Timeout

An optional timeout value in 100 ns units.

Description

This function waits for *Event* to fire. If *Event* does fire, then **EFI_SUCCESS** is returned. If *Timeout* is zero, then this function will wait indefinitely for *Event* to fire. If *Timeout* is not zero, then this function will wait for both *Event* and the *Timeout* period. If the *Timeout* expires, then **EFI_TIME_OUT** will be returned.

Status Codes Returned

EFI_SUCCESS	*Event* fired before *Timeout* expired.
EFI_TIME_OUT	*Timeout* expired before *Event* fired.

WaitForEventWithTimeout()

Summary

Prints a string for the given number of seconds until either the timeout expires or the user presses a key.

Prototype

```
VOID
WaitForEventWithTimeout (
  IN  EFI_EVENT       Event,
  IN  UINTN           Timeout,
  IN  UINTN           Row,
  IN  UINTN           Column,
  IN  CHAR16          *String,
  IN  EFI_INPUT_KEY   TimeoutKey,
  OUT EFI_INPUT_KEY   *Key
  )
```

Parameters

Event

The **SIMPLE_TEXT_INPUT** event for which to wait. Type **EFI_EVENT** is defined in **CreateEvent()** in the *EFI 1.10 Specification.*

Timeout

Timeout value in 1 second units

Row

The row to print *String*.

Column

The column to print *String*.

String

The string to display on the standard output device.

TimeoutKey

The key to return in *Key* if a timeout occurs. Type **EFI_INPUT_KEY** is defined in **SIMPLE_INPUT.ReadKeyStroke()** in the *EFI 1.10 Specification.*

Key

Either the key the user pressed or the *TimeoutKey* if the *Timeout* expired.

Description

This function waits for *Event* to fire or *Timeout* to expire. If *Event* does fire, then a keystroke is read from the standard input device returned in *Key*. If the *Timeout* in seconds does expire, then *TimeoutKey* is returned in *Key*. For each second that passes while this function is waiting, *String* is displayed on the standard output device at (*Row*, *Column*).

RtLibEnableVirtualMappings()

Summary

Converts runtime pointers that are internal to the EFI Library to a new virtual base address.

Prototype

```
VOID
RtLibEnableVirtualMappings (
  VOID
  );
```

Parameters

None.

Description

This function converts any runtime pointers that are internal to the EFI Library to a new virtual address base. This function should be called only once as an OS transitions the EFI firmware from a flat physical memory model to a virtual runtime memory model.

RtConvertList()

Summary

Converts all the pointers in a doubly linked list to a new virtual base address.

Prototype

```
VOID
RtConvertList (
  IN UINTN              DebugDisposition,
  IN OUT LIST_ENTRY     *ListHead
  );
```

Parameters

DebugDisposition

A bit mask that describes the pointer types in the linked list. See “Related Definitions” below for the defined bit masks.

ListHead

A pointer to a doubly linked list. Type **LIST_ENTRY** is defined in section 4.3.1, “Related Definitions.”

Description

This function converts all the *Flink* and *Blink* fields of the doubly linked list *ListHead* to a new virtual base address.

Related Definitions

```
//*******************************************************
// DebugDisposition values
//*******************************************************
#define EFI_OPTIONAL_PTR            0x00000001
#define EFI_INTERNAL_FNC            0x00000002
#define EFI_INTERNAL_PTR            0x00000004
```

Following is a description of the fields in the above definition.

EFI_OPTIONAL_PTR	If this flag is specified, the pointer being converted is allowed to be **NULL**.
EFI_INTERNAL_FNC	The pointer is to an internal runtime function.
EFI_INTERNAL_PTR	The pointer is to an internal runtime data.

LibGetSystemConfigurationTable()

Summary

Returns a system configuration table that is stored in the EFI System Table based on the provided GUID.

Prototype

```
EFI_STATUS
LibGetSystemConfigurationTable (
  IN EFI_GUID          *TableGuid,
  IN OUT VOID          **Table
  );
```

Parameters

TableGuid

A pointer to the table's GUID type. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Table

On exit, a pointer to a system configuration table.

Description

This function searches the list of configuration tables that are stored in the EFI System Table for a table with a GUID that matches *TableGuid*. If one is found, then a pointer to the configuration table is returned in *Table* and **EFI_SUCCESS** is returned. If a matching GUID cannot be found, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	A configuration table matching *TableGuid* was found.
EFI_NOT_FOUND	A configuration table matching *TableGuid* could not be found.