Extensible Firmware Interface Library Specification

Draft for Review

Version 1.10

December 16, 2001

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Except for a limited copyright license to copy this specification for internal use only, no license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. Intel does not warrant or represent that such implementation(s) will not infringe such rights.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

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	Update 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

Contents

Figures

Tables

1 Introduction

The *Extensible Firmware Interface (EFI) 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 EFI Shells, EFI Shell commands, EFI Applications, EFI OS loaders, and EFI Device Drivers. This document describes in detail each of the functions and macros present in the EFI Library along with the constants and global variables that are exported.

1.1 Organization of this Document

This specification is organized as follows:

Table 1-1. Specification Organization and Contents

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

1.2 Goals

The primary goal of the EFI Library Specification is to provide documentation for the collection of library functions that are available to EFI firmware developers, EFI shell developers, EFI shell application developers, and shrink wrapped operating system boot loader developers. These library functions complement those APIs described in the EFI Specification. The combination of the EFI APIs and the EFI Library functions provide all the functions required for basic console I/O, basic disk I/O, memory management, linked list management, and string manipulation. In addition, there are miscellaneous functions for 64 bit math operations, spin locks, and helper functions used to managing device handle, device protocols, and device paths.

1.3 Target Audience

This document is intended for the following readers:

- OEMs who will be creating Intel Architecture-based platforms intended to boot shrink-wrap operating systems.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in Intel Architecture-based products.

- Operating system developers who will be adapting their shrink-wrap operating system products to run on Intel Architecture-based platforms.

1.4 Prerequisite Specifications

Extensible Firmware Interface Specification Version 0.9, Intel Corporation, 1999.

1.5 Conventions Used in This Document

This document uses typographic and illustrative conventions described below.

1.5.1 Data Structure Illustrations

The Intel Architecture processors of the IA-32 family are “little endian” machines. This 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 IA-64 family may be configured for both “little endian” and “big endian” operation.

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 1-2. Bit positions are numbered from right to left.

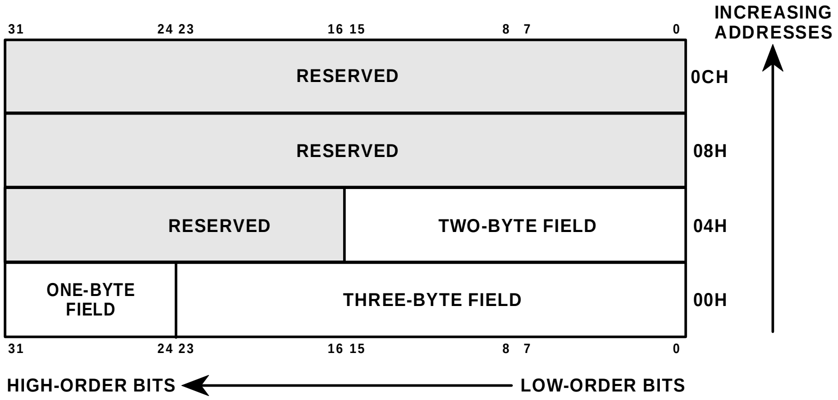

Figure 1-2. Memory Layout Conventions

In some memory layout descriptions, certain fields are marked **RESERVED**. Software should initialize these fields as binary zeros, but should otherwise treat them as having a future, though unknown effect. Software should avoid any dependence on the values in the reserved fields.

1.5.2 Typographic Conventions

The following typographic conventions are used in this document to illustrate programming concepts:

`Prototype` This typeface is use to indicate prototype code.

`Argument` This typeface is used to indicate arguments.

`Name` This typeface is used to indicate actual code or a programming construct.

register This typeface is used to indicate a processor register.

2
Constants

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

There are three global variables exported by the EFI Library that provide access to the EFI System Table (ST), EFI Boot Time Services (BS), and the EFI Run Time Services (RT). The declaration of these global variables is shown below.

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

A group of EFI_GUID variables are also exported by the EFI Library. These include protocol GUIDs and other miscellaneous GUIDs used by the EFI sample implementation.

```
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;
```

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

```
extern EFI_GUID EfiGlobalVariable;
extern EFI_GUID GenericFileInfo;
extern EFI_GUID FileSystemInfo;
extern EFI_GUID FileSystemVolumeLabelInfo;
extern EFI_GUID PcAnsiProtocol;
extern EFI_GUID Vt100Protocol;
extern EFI_GUID NullGuid;
extern EFI_GUID UnknownDevice;
```

The following are the group of `EFI_GUID` variables for the disk type entries.

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

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;
```

There are also three Device Path data structures that are exported by the EFI Library. These are used to build complete device paths.

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

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

```
extern EFI_DEVICE_IO_INTERFACE  *GlobalIoFncs;
```

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

```
extern EFI_MEMORY_TYPE PoolAllocationType;
```

4
Functions and Macros

The functions and macros exported by the EFI Library are grouped as follows:

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

4.1 Initialization Functions

The initialization functions in the EFI Library are used to initialize the execution environment so that other EFI Library function may be used. Table 4-1 lists the initialization support functions that are described in the following sections.

Table 4-1. Initialization Functions

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

4.1.1 InitializeLib Function

The **InitializeLib()** function initializes the EFI Library.

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

Parameters

ImageHandle	A handle for the image that is initializing the library.
SystemTable	A pointer to the EFI system table.

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 required by the EFI Library functions. In addition, it verifies the CRCs for all the EFI system tables.

4.1.2 InitializeUnicodeSupport Function

The **`InitializeUnicodeSupport()`** function initializes the use of the language dependant EFI Library functions.

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

Parameters

`LangCode` The 3 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.

4.2 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 following is the definition of the **LIST_ENTRY** data structure.

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

Table 4-2 contains the list of macros that are described in the following sections.

Table 4-2. Linked List Support Macros

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

4.2.1 InitializeListHead Macro

The **`InitializeListHead()`** macro initializes the head node of a doubly linked list.

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

Parameters

`ListHead`	A pointer to the head node of a new doubly linked list.

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`*.

4.2.2 IsListEmpty Macro

The **`IsListEmpty()`** macro checks to see if a doubly linked list is empty or not.

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

Parameters

`ListHead`	A pointer to the head node of a doubly linked list.

Description

This macro checks to see if the doubly linked list is empty. If the linked list contains zero 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

4.2.3 RemoveEntryList Macro

The **`RemoveEntryList()`** macro removes a node from a doubly linked list.

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

Parameters

`Entry`	A pointer to a node in a linked list

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 used by this node if that is required.

4.2.4 InsertTailList Macro

The **`InsertTailList()`** macro adds a node to the end of a doubly linked list.

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

Parameters

`ListHead`	A pointer to the head node of a doubly linked list.
`Entry`	A pointer to a node that is to be added 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`*.

4.2.5 InsertHeadList Macro

The **InsertHeadList()** macro adds a node to the beginning of a doubly linked list.

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

Parameters

ListHead A pointer to the head node of a doubly linked list.

Entry A pointer to a node that is to be inserted 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*.

4.3 String Functions

The string functions in the EFI Library perform operations on Unicode and ASCII string. Table 4-3 contains the list of string support functions that are described in the following sections.

Table 4-3. String Functions

Name	Description
StrCmp	Compares two Unicode strings.
StrnCmp	Compares a portion of two Unicode strings.
StriCmp	Performs a case insensitive comparison of two Unicode strings.
StrCpy	Copies one Unicode string to another Unicode string.
StrCat	Concatenates two Unicode strings.
StrLen	Determines the length of a Unicode string.
StrSize	Determines the size of a Unicode string in bytes.
StrDuplicate	Creates a duplicate of a Unicode string.
StrLwr	Converts characters in a Unicode string to upper case characters.
StrUpr	Converts characters in a Unicode string to lower case 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.

4.3.1 StrCmp Function

The **StrCmp()** function compares two Unicode strings.

```
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.

4.3.2 StrnCmp Function

The **StrnCmp()** function compares a portion of two Unicode strings.

```
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.

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.

4.3.3 StriCmp Function

The **StriCmp()** function performs a case insensitive comparison of two Unicode strings.

```
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

4.3.4 StrCpy Function

The **`StrCpy()`** function copies one Unicode string to another Unicode string.

```
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`*.

4.3.5 StrCat Function

The **`StrCat()`** function concatenates one Unicode string to another Unicode string.

```
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 string. The contents of Unicode string *`Src`* are concatenated to the end of Unicode string *`Dest`*.

4.3.6 StrLen Function

The **`StrLen()`** function determines the length of a Unicode string.

```
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*.

4.3.7 StrSize Function

The **StrSize()** function determines the size of a Unicode string in bytes.

```
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.

4.3.8 StrDuplicate Function

The **StrDulicate()** function duplicates a Unicode string.

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

Parameters

Src Pointer to a Null-terminated Unicode string.

Description

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

4.3.9 StrLwr Function

The **StrLwr()** function converts all the characters in a Unicode string to lower case.

```
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 lower case characters.

4.3.10 StrUpr Function

The **`StrUpr()`** function converts all the characters in a Unicode string to upper case.

```
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 upper case characters.

4.3.11 strlena Function

The **`strlena()`** function determines the length of an ASCII string.

```
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*.

4.3.12 strcmpa Function

The **strcmpa()** function compares two ASCII strings.

```
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

4.3.13 strncmpa Function

The **strncmpa()** function compares a portion of two ASCII strings.

```
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 character 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.

4.3.14 xtoi Function

The **xtoi()** function converts a hexadecimal formatted Unicode string to a value.

```
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.

4.3.15 Atoi Function

The **Atoi()** function converts a decimal formatted Unicode string to a value.

```
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.

4.3.16 MetaMatch Function

The **MetaMatch()** function checks to see if a Unicode string matches a given pattern.

```
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 following syntax can be used to build the string *Pattern*.

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

Examples patterns:

***.FW**	Match all strings that end in ".FW".
[a-z]	Match any lower case character.
[!@#$%^&*()]	Match any one of these symbols.
z	Match the lower case character z.
DATA?.*	Match 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

4.3.17 MetaiMatch Function

The `MetaiMatch()` function performs a case insensitive comparison of a Unicode pattern string and a Unicode string.

```
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 following syntax can be used to build the string *Pattern*.

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

Examples patterns (for English):

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

4.3.18 ValueToString Function

The **ValueToString()** function converts an integer to decimal formatted Unicode string.

```
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*.

4.3.19 ValueToHex Function

The **`ValueToHex()`** function converts an integer to hexadecimal formatted Unicode string.

```
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`*.

4.3.20 TimeToString Function

The **TimeToString()** function converts the time and date stored in a data structure into a Unicode string.

```
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.

Description

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

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.

4.3.21 GuidToString Function

The **GuidToString()** function converts a GUID data structure into a Unicode string.

```
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.

Description

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

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 is typically an ethernet hardware ID.

4.3.22 StatusToString Function

The **StatusToString()** function converts an EFI_STATUS value into a Unicode string.

```
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 4-4 shows how **EFI_STATUS** values are converted to strings.

Table 4-4. EFI_STATUS Output Formats

EFI_STATUS	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”

4.3.23 DevicePathToStr Function

The **DevicePathToStr()** function converts a device path data structure into a printable Null-terminated Unicode string.

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

Parameters

DevPath A pointer to a device path data structure.

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. Tables 4-5, 4-6, 4-7, and 4-8 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 ','.

Table 4-5. 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 Device Path	"MemMap(<Memory Type>:<Starting Address>-<Ending Address>)
Controller Device Path	"Ctlr(<Controller>)"
Vendor Device Path	"VenHw(<Vendor GUID>)"
Vendor Device Path(Legacy)	"VenHw(<Vendor GUID>:<Legacy Drive Letter>)"
ACPI Device Path	"Acpi(<ACPI HID>,<ACPI UID>)"
ACPI Device Path(EISA)	"Acpi(PNP<ACPI HID>,<ACPI UID>)"
Unknown	"?"

Table 4-6. Messaging Device Path Output Formats

Messaging Device Path Type	Output Format
ATAPI	“ATA(<Primary/Secondary>,<Master/Slave>)”
SCSI	“Scsi(<PUN>,<LUN>)”
FibreChannel	“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	“VenMsg(<Vendor GUID>)”
Vendor(Legacy)	“VenMsg(<Vendor GUID>:<Legacy Drive Letter>)”
Unknown	“?”

Table 4-7. Media Device Path Output Formats

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

Table 4-8. BSS Device Path Output Formats

BSS Device Path Type	Output Format
Floppy	“Floppy”
HardDrive	“Harddrive”
CDROM	“CDROM”
PCMCIA	“PCMCIA”
USB	“USB”
Embedded Network	“Net”
Other	“?”
Unknown	“?”

4.4 Memory Support Functions

The EFI Library provides a set of functions that operating on buffers in memory. Buffers can either be allocated on the stack, as global variables, or from the memory pool. To prevent memory leaks, it is the caller's responsibility to maintain buffers allocated from pool. This means that the caller must free a buffer when that buffer is no longer needed. Table 4-9 contains the list of memory support functions that are described in the following sections.

Table 4-9. 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.

4.4.1 ZeroMem Function

The **`ZeroMem()`** function fills a buffer with zeros.

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

Parameters

`Buffer`	Pointer to the buffer to zero.
`Size`	Number of bytes to zero in *`Buffer`*.

Description

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

4.4.2 SetMem Function

The **SetMem()** function fills a buffer with a value..

```
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 to fill *Buffer* with.

Description

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

4.4.3 CopyMem Function

The **CopyMem()** function copies the contents of one buffer to another buffer.

```
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*.

4.4.4 CompareMem Function

The **CompareMem()** function compares the contents of two buffers.

```
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	Desk is not identical to Src for len bytes

4.4.5 AllocatePool Function

The **AllocatePool()** function allocates a buffer from memory with type **PoolAllocationType**.

```
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.

4.4.6 AllocateZeroPool Function

The **`AllocatePool()`** function allocates and zeros buffer from memory.

```
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.

4.4.7 ReallocatePool Function

The **ReallocatePool()** function adjusts the size of a previously allocated buffer.

```
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 can not 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.

4.4.8 FreePool Function

The **FreePool()** function releases a previously allocated buffer.

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

Parameters

p	A pointer to the buffer to free.

Description

The free memory is returned to the available pool. The buffer *p* must have been allocated with **AllocatePool().**

4.4.9 GrowBuffer Function

The **GrowBuffer()** function either allocates a new buffer or it increases the size of a previously allocated buffer.

```
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

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 can not 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 can not 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 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 required by **GetVariable()**. So, the next call to **GetVariable()** will succeed, and the buffer 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.

4.4.10 LibMemoryMap Function

The **LibMemoryMap()** function retrieves the systems current memory map.

```
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 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*. The storage for the memory map is allocated by this function from pool.

4.5 Text I/O Functions

The Text I/O functions in the EFI Library provide a simple means to get input and output from a console device. 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, and 'F' is a set of optional field modifiers, and the data type of the argument to print. Table 4-11 lists the optional field modifiers and arguments types.

Table 4-11. 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. Please see the TimeToString() function for the output format of this field.
G	The argument is a pointer to a EFI_GUID data structure. Please see the GuidToString() function for the output format of this field.
R	The argument is an EFI_STATUS value. Please 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 4-12 contains the list of Text I/O functions that are described in the following sections.

Table 4-12. Text I/O Functions

Name	Description
Input	Input a Unicode string at the current cursor location using the console in and console out device.
Iinput	Input a Unicode string at the current cursor location using the specified input and output devices.
Output	Send 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	Decide if Graphic is a supported Unicode Box Drawing character.
IsValidAscii	Decide if character is legal ASCII element.
IsValidEfiCntlChar	Decide if character is an EFI Control character.

4.5.1 Input Function

The **Input()** function reads a Unicode string form the console in device at the current cursor location.

```
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 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

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.

4.5.2 IInput Function

The **IInput()** function reads a Unicode string form the specified device at the current cursor location.

```
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 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

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.

4.5.3 Output Function

The **`Output()`** function sends a Unicode string to the console out device at the current cursor location.

```
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 specified in the system table.

4.5.4 Print Function

The **Print()** function sends a formatted Unicode string to the console out device at the current cursor location.

```
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.

4.5.5 PrintAt Function

The **PrintAt()** function sends a formatted Unicode string to the console out device at the specified cursor location.

```
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 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 at the cursor location specified by *Column* and *Row*. The length of the formatted Unicode string is returned.

4.5.6 IPrint Function

The **IPrint()** function sends a formatted Unicode string to the specified device at the current cursor location.

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

Parameters

Out A pointer to the output devices interface protocol..

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 device specified by *Out* at the current cursor location. The length of the formatted Unicode string is returned.

4.5.7 IPrintAt Function

The **IPrintAt()** function sends a formatted Unicode string to the specified device at the specified cursor location.

```
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 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 device specified by Out at the cursor location specified by *Column* and *Row*. The length of the formatted Unicode string is returned.

4.5.8 APrint Function

The **`APrint()`** function sends a formatted Unicode string to the console out device at the current cursor location.

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

Parameters

`fmt`	A pointer to an ASCII 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.

4.5.9 SPrint Function

The **SPrint()** function sends a formatted Unicode string to the specified buffer.

```
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 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. Up to *StrSize* characters of this formatted Unicode string is then stored in *Str*. The length of the formatted Unicode string is returned.

4.5.10 PoolPrint Function

The **PoolPrint()** function sends a formatted Unicode string to a buffer allocated from pool.

```
CHAR16 *
PoolPrint (
    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. 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.

4.5.11 CatPrint Function

The **CatPrint()** function concatenates a formatted Unicode string to a Unicode string allocated from pool.

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

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

Parameters

Str A pointer to **POOL_PRINT** data structure containing a Unicode string.

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. 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.

4.5.12 DumpHex Function

The **DumpHex()** function prints the contents of a buffer in hexadecimal format.

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

Parameters

Indent Number of spaces to indent text output.

Offset Byte offset within *UserData* 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 the 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 SystemTable.

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                                      *....*
```

4.5.13 LibIsValidTextGraphics Function

The **LibIsValidTextGraphics()** function returns **TRUE** if Graphic is a supported Unicode Box Drawing character.

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

Parameters

Graphic	Unicode character to test.
PcAnsi	Optional pointer to return PCANSI equivalent of Graphic.
Ascii	Optional pointer to return ASCII equivalent of Graphic.

Description

This function returns a **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 has also 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.

4.5.14 IsValidAscii Function

The **`IsValidAscii()`** function determines if the argument is a legal ASCII character.

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

Parameters

`Ascii`	Number of spaces to indent 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.

4.5.15 IsValidEfiCntlChar Function

The **`IsValidEfiCntlChar()`** function determines if the character is one of the four EFI control characters.

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

Parameters

C	Character to decide if is also a control character.

Description

This function returns **`TRUE`** if the input character *`c`* is a legal EFI control character. The control characters included in Table 4-13.

Table 4-13. EFI Control Characters

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

Otherwise, the value **`FALSE`** is returned.

Status Codes Returned

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

4.6 Math Functions

The EFI Library provides a few math functions to operate on 64 bit operands. These include shift operations, multiplication and division. Table 4-14 lists the set of 64 bit math functions that are described in the following sections.

Table 4-14. Math Functions

Name	Description
LShiftU64	Shift a 64 bit integer left between 0 and 63 bits.
RShiftU64	Shift a 64 bit integer right between 0 and 63 bits.
MultU64x32	Multiply a 64 bit unsigned integer by a 32 bit unsigned integer and generate a 64 bit unsigned result.
DivU64x32	Divide a 64 bit unsigned integer by a 32 bit unsigned integer and generate a 64 bit unsigned result with an optional 32 bit unsigned remainder.

4.6.1 LShiftU64 Function

The `LShiftU64()` function shifts a 64 bit integer left between 0 and 63 bits.

```
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.

4.6.2 RshiftU64 Function

The **`LshiftU64()`** function shifts a 64 bit integer right between 0 and 63 bits.

```
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.

4.6.3 MultU64x32 Function

The **MultU64x32()** function multiples a 64 bit unsigned integer by a 32 bit unsigned integer and generates a 64 bit unsigned result.

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

Parameters

Multiplicand A 64 bit unsigned value.

Multiplier A 32 bit unsigned value.

Description

This function multiples 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.

4.6.4 DivU64x32 Function

The **DivU64x32()** function 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.

```
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.

4.7 Spin Lock Functions

Spin locks are used to protect data structures that may be updated by more than one processor at a time, or a single processor that may update the same data structure while running a several different priority levels. A spin lock is stored in an FLOCK data structure.

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

Table 4-15 lists the support functions for creating and maintaining spin locks. These functions are described in the following sections.

Table 4-15. Spin Lock Functions

Name	Description
InitializeLock	Initialize a spin lock.
AcquireLock	Acquire a spin lock.
ReleaseLock	Release a spin lock.

4.7.1 InitializeLock Function

The **InitializeLock()** function initializes a basic mutual exclusion lock.

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

Parameters

Lock	A pointer to the lock data structure to initialize.
Priority	The task priority level of the lock.

Description

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

4.7.2 AcquireLock Function

The **AcquireLock()** function acquires ownership of a lock

```
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.

4.7.3 ReleaseLock Function

The **ReleaseLock()** function releases ownership of a lock

```
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.

4.8 Handle and Protocol Support Functions

The EFI Library contains a set of functions that help drivers maintain the protocol interfaces in the boot services environment. Table 4-16 lists the set of helper functions that are described in the following sections.

Table 4-16. 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 boot services environment..
LibUninstallProtocolInterfaces	Removes one or more protocol interfaces from the boot services environment.
LibReinstallProtocolInterfaces	Reinstalls one or more protocol interfaces into the 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.

4.8.1 LibLocateHandle Function

The **LibLocateHandle()** function returns an array of handles that support the requested protocol in a buffer allocated from pool.

```
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.
Protocol	Provides the protocol to search by. This parameter is only valid for *SearchType* **ByProtocol**.
SearchKey	Supplies the search key depending on the *SearchType*.
NoHandles	The number of handles returned in *Buffer*.
Buffer	A pointer to the buffer to return the requested array of handles that support *Protocol*.

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 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 result 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.

4.8.2 LibLocateHandleByDiskSignature Function

The **LibLocateHandleByDiskSignature()** function returns an array of handles that support the requested protocol in a buffer allocated from pool.

```
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 MBR to search for. This 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 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*.

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 following are definitions for the valid values for *MBRType* and *SignatureType*.

```
#define MBR_TYPE_PCAT                       0x01
#define MBR_TYPE_EFI_PARTITION_TABLE_HEADER 0x02

#define SIGNATURE_TYPE_MBR                  0x01
#define SIGNATURE_TYPE_GUID                 0x02
```

Status Codes Returned

EFI_SUCCESS	The result 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.

4.8.3 LibLocateProtocol Function

The **LibLocateProtocol()** function returns the first protocol instance that matches the given protocol.

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

Parameters

Protocol Provides the protocol to search for.

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 Instance 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*.

4.8.4 LibInstallProtocolInterfaces Function

The **LibInstallProtocolInterfaces()** function installs one or more protocol interfaces into the boot services environment.

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

Parameters

Handle The handle to install the new protocol interfaces on, or **NULL** if a new handle is to be allocated.

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

Description

This function installs a set of protocol interfaces into the 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 boot services routine **InstallProtoclInterface()** 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.

4.8.5 LibUninstallProtocolInterfaces Function

The **LibInstallProtocolInterfaces()** function removes one or more protocol interfaces into the boot services environment.

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

Parameters

Handle The handle to remove the protocol interfaces from.

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

Description

This function removes a set of protocol interfaces from the 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 boot services routine **UninstallProtoclInterface()** 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.

4.8.6 LibReinstallProtocolInterfaces Function

The **`LibReinstallProtocolInterfaces()`** function replaces one or more protocol interfaces into the boot services environment.

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

Parameters

Handle	The handle to remove the protocol interfaces from.
...	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 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, and 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 boot services routine **`ReinstallProtoclInterface()`** 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 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.

4.8.7 LibScanHandleDatabase Function

The **LibScanHandleDatabase()** function scans the handle database and collects EFI Driver Model related information.

```
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.

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 *HandeBuffer*.

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 Defintions” for the list of available attributes. This buffer is allocated by this call, and the caller is responsible for freeing this buffer.

Related Definitions

```
#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
```

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*.

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**.

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.

4.8.8 LibGetManagingDriverBindingHandles Function

The **`LibGetManagingDriverBindingHandles()`** function retrieves the set of driver handles that are currently managing a specific controller.

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

Parameters

ControllerHandle

A handle that represents a controller.

DriverBindingHandleCount

Returns the number of handles in *DriverBindingHandeBuffer*.

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 the 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*.

4.8.9 LibGetParentControllerHandles Function

The **`LibGetParentControllerHandles()`** function retrieves the set of device handles that are parents of a specific controller.

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

Parameters

ControllerHandle

A handle that represents a controller.

ParentControllerHandleCount

Returns the number of handles in *ParentControllerHandeBuffer*.

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*.

4.8.10 LibGetChildControllerHandles Function

The **`LibGetChildControllerHandles()`** function retrieves the set of device handles that are children of a specific controller.

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

Parameters

ControllerHandle

A handle that represents a controller.

ChildControllerHandleCount

Returns the number of handles in *ChildControllerHandeBuffer*.

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*.

4.8.11 LibGetManagedControllerHandles Function

The **LibGetManagedControllerHandles()** function retrieves the set of device handles that a driver is currently managing.

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

Parameters

DriverBindingHandle

A handle that represents a driver.

ControllerHandleCount

Returns the number of handles in *ControllerHandeBuffer*.

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*.

4.8.12 LibGetManagedChildControllerHandles Function

The **`LibGetManagedChildControllerHandles()`** function retrieves the set of device handles that a driver produced as children of a specific controller.

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

Parameters

DriverBindingHandle

A handle that represents a driver.

ControllerHandle

A handle that represents a controller.

ChildControllerHandleCount

Returns the number of handles in *ChildControllerHandeBuffer*.

ChildControllerHandleBuffer

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

4.9 File I/O Support Functions

Table 4-17 lists some helper function related to Files and a set of functions and macros that facilitate the manipulating of Files.

```
typedef VOID        *SIMPLE_READ_FILE;
```

Table 4-17. 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 is valid.
OpenSimpleReadFile	Opens a file from several possible sources and returns a file handle.
ReadSimpleReadFile	Read from a file opened with OpenSimpleReadFile.
CloseSimpleReadFile	Close a file opened with OpenSimpleReadFile.

4.9.1 LibOpenRoot Function

The **LibOpenRoot()** function opens and returns a file handle to the root directory of a volume.

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

Parameters

DeviceHandle A handle for a device.

Description

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

4.9.2 LibFileInfo Function

The **LibFileInfo()** function gets the file information from an open file descriptor, and stores it in a buffer allocated from pool.

```
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 can not be retrieved, or there is not enough memory in pool to store the data structure, **NULL** will be returned.

4.9.3 LibFileSystemInfo Function

The **LibFileSystemInfo()** function gets the file system information from an open file descriptor, and stores it in a buffer allocated from pool.

```
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 can not be retrieved, or there is not enough memory in pool to store the data structure, **NULL** will be returned.

4.9.4 LibFileSystemVolumeLabelInfo Function

The **`LibFileSystemVolumeLabelInfo()`** function gets the file system information from an open file descriptor, and stores it in a buffer allocated from pool.

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

Parameters

FHand	A file handle.

Description

This function retrieves the eleven-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 can not be retrieved, or there is not enough memory in pool to store the data structure, **`NULL`** will be returned.

4.9.5 ValidMBR Function

The **`ValidMBR()`** function determines if a hard drive’s Master Boot Record is valid.

```
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_IO Protocol handle.

Description

This function verifies that the layout of partitions described in the master boot record are valid. The master boot record 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.

4.9.6 OpenSimpleReadFile Function

The **`OpenSimpleReadFile()`** function opens a file from several possible sources and returns a file handle.

```
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.

`DeviceHandle` Pointer to the device handle of the device to open.

`SimpleReadHandle` A pointer to the file handle to return.

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`*.

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 form 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.

4.9.7 ReadSimpleReadFile Function

The **ReadSimpleReadFile()** reads data from a file opened with **OpenSimpleReadFile()**

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

Parameters

SimpleReadHandle	A file handle.
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 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 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	The *ReadSize* is too small to read the current file. *ReadSize* had been updated with the size needed to complete the request.

4.9.8 CloseSimpleReadFile Function

The **`CloseSimpleReadFile()`** closes a file opened with **`OpenSimpleReadFile().`**

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

Parameters

`SimpleReadHandle`	A file handle.

Description

This function closes the file 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.

4.10 Device Path Support Functions

Table 4-18 lists the support functions for creating and maintaining device path data structures. These functions are described in the following sections.

Table 4-18. 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 multi-instance device path.
LibDuplicateDevicePathInstance	Creates a second corresponding instance of a given device path.

4.10.1 DevicePathFromHandle Function

The **DevicePathFromHandle()** function retrieves the device path for the specified handle.

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

Parameters

Handle	A handle.

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.

4.10.2 DevicePathInstance Function

The **DevicePathInstance()** function retrieves the next device path instance from a device path data structure.

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

Parameters

DevicePath	A pointer to a device path data structure.
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**.

4.10.3 DevicePathInstanceCount Function

The **DevicePathInstanceCount()** function is used to determine the number of device path instances that exist in a device path.

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

Parameters

DevicePath A pointer to a device path data structure.

Description

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

4.10.4 AppendDevicePath Function

The **AppendDevicePath()** function is used to append a device path to all the instances in another device path.

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

Parameters

Src1	A pointer to a device path data structure.
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.

4.10.5 AppendDevicePathNode Function

The **AppendDevicePathNode()** function is used to append a device path node to all the instances in another device path.

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

Parameters

Src1	A pointer to a device path data structure.
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.

4.10.6 AppendDevicePathInstance Function

The **AppendDevicePathInstance()** function is used to add a device path instance to a device path.

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

Parameters

Src	A pointer to a device path data structure.
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.

4.10.7 FileDevicePath Function

The **FileDevicePath()** allocates a device path for a file and appends it to an existing device path.

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

Parameters

Device	A pointer to a device handle.
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.

4.10.8 DevicePathSize Function

The **DevicePathSize()** function returns the size of a device path in bytes.

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

Parameters

DevPath	A pointer to a device path data structure.

Description

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

4.10.9 DuplicateDevicePath Function

The **DuplicateDevicePath()** function creates a duplicate copy of an existing device path.

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

Parameters

DevPath	A pointer to a device path data structure.

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.

4.10.10 LibDevicePathToInterface Function

The **LibDevicePathToInterface()** function retrieves a protocol interface for a device.

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

Parameters

Protocol	The published unique identifier of the protocol.
FilePath	A pointer to a device path data structure.
Interface	Supplies and 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.

4.10.11 UnpackDevicePath Function

The **UnpackDevicePath()** function unpacks a device path data structure so that all the nodes of a device path are naturally aligned.

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

Parameters

DevPath	A pointer to a device path data structure.

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.

4.10.12 LibMatchDevicePaths Function

The **LibMatchDevicePaths()** function compares a device path data structure to that of all the nodes of a second device path instance.

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

Parameters

Multi	A pointer to a multi-instance device path data structure.
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

4.10.13 LibDuplicateDevicePathInstance Function

The **`LibDuplicateDevicePathInstance()`** function creates a device path data structure that identically matches the device path passed in.

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

Parameters

`DevPath`	A pointer to a device path data structure.

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.

4.11 PCI Functions and Macros

Table 4-19 lists some helper function related to PCI devices and a set of functions and macros that are used to access PCI I/O and PCI Configuration Space.

Table 4-19. PCI Functions and Macros

Name	Description
PCIFindDeviceClass	Finds a PCI device that matches the PCI BaseClass and SubClass.
PCIFindDevice	Finds a PCI device that matches the PCI Device ID and Vendor ID.
InitializeGlobalIoDevice	Retrieves the DEVICE_IO 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	Read an 8 bit value from an I/O port.
Outp	Write an 8 bit value to an I/O port.
Inpw	Read a 16 bit value from an I/O port.
Outpw	Write a 16 bit value to an I/O port.
Inpd	Read a 32 bit value from an I/O port.
Outpd	Write a 32 bit value to an I/O port.
Readpci8	Read an 8 bit value from PCI Configuration Space.
Writepci8	Write an 8 bit value to PCI Configuration Space.
readpci16	Read a 16 bit value from PCI Configuration Space.
writepci16	Write a 16 bit value to PCI Configuration Space.
readpci32	Read a 32 bit value from PCI Configuration Space.
writepci32	Write a 32 bit value to PCI Configuration Space.

4.11.1 PciFindDeviceClass Function

The **PciFindDeviceClass()** function finds the first PCI device with the specified class.

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

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

```
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.

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

SubClass The PCI sub class of the device to search for.

Description

This function search all the PCI busses 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*.

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

4.11.2 PciFindDevice Function

The **PciFindDevice()** function finds the first PCI device with the specified Device ID and Vendor ID.

```
typedef struct {
    UINT16      VendorId;
    UINT16      DeviceId;
    UINT16      Command;
    UINT16      Status;
    UINT8       RevisionID;
    UINT8       ClassCode[3];
    UINT8       CacheLineSize;
    UINT8       LaytencyTimer;
    UINT8       HeaderType;
    UINT8       BIST;
} PCI_DEVICE_INDEPENDENT_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;

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

```
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.
VendorId	The PCI base class of the device to search for.
DeviceId	The PCI sub class of the device to search for.
Pci	A pointer to the configuration space header of the device.

Description

This function search all the PCI busses 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.

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

4.11.3 InitializeGlobalIoDevice Function

The **InitializeGlobalIoDevice()** function returns a DEVICE_IO protocol instance that is supported by the given device.

```
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.

Protocol The protocol that a device driver is attempting to register for this device.

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

GlobalIoFuncs A pointer to the DEVICE_IO protocol instance that is supported by the device specified by *DevicePath*.

Description

This function check to see if 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_IO protocol. If it does, then the DEVICE_IO protocol instance is returned in *GlobalIoFncs*.

Status Codes Returned

EFI_SUCCESS	A DEVICE_IO protocol instance was returned..
EFI_LOAD_ERROR	The device already supports *Protocol*.
EFI_NOT_FOUND	A DEVICE_IO protocol instance was not found.

4.11.4 ReadPort Function

The **ReadPort()** function reads an I/O port using a DEVICE_IO protocol instance.

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

Parameters

GlobalIoFncs The DEVICE_IO protocol instance to use to perform the I/O read.

Width The width of the I/O read operation.

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.

4.11.5 WritePort Function

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

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

Parameters

GlobalIoFncs	The DEVICE_IO protocol instance to use to perform the I/O write.
Width	The width of the I/O write operation.
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.

4.11.6 ReadPciConfig Function

The **ReadPciConfig()** function reads from PCI Configuration Space using a DEVICE_IO protocol instance.

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

Parameters

GlobalIoFncs The DEVICE_IO protocol instance to use to perform the PCI Configuration read.

Width The width of the PCI Configuration read operation.

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.

4.11.7 WritePciConfig Function

The **WritePciConfig()** function writes to PCI Configuration Space using a DEVICE_IO protocol instance.

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

Parameters

GlobalIoFncs	The DEVICE_IO protocol instance to use to perform the I/O write.
Width	The width of the PCI Configuration write operation.
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 PCI Configuration Space at the address specified by *Port* and *Width* using the protocol interface functions in *GlobalIoFncs*. *Data* is returned.

4.11.8 inp Macro

The **inp()** macro reads an 8 bit value from an I/O port using the DEVICE_IO protocol instance **GlobalIoFcns**.

```
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*.

4.11.9 outp Macro

The **outp()** macro writes an 8 bit value to an I/O port using the DEVICE_IO protocol instance **GlobalIoFcns**.

```
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*.

4.11.10 inpw Macro

The **`inpw()`** macro reads a 16 bit value from an I/O port using the DEVICE_IO protocol instance **`GlobalIoFcns`**.

```
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`*.

4.11.11 outpw Macro

The **`outpw()`** macro writes a 16 bit value to an I/O port using the DEVICE_IO protocol instance **`GlobalIoFcns`**.

```
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`*.

4.11.12 inpd Macro

The **inpd()** macro reads a 32 bit value from an I/O port using the DEVICE_IO protocol instance **GlobalIoFcns**.

```
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*.

4.11.13 outpd Macro

The **`outpd()`** macro writes a 32 bit value to an I/O port using the DEVICE_IO protocol instance **`GlobalIoFcns`**.

```
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`*.

4.11.14 readpci8 Macro

The **`readpci8()`** macro reads an 8 bit value from PCI Configuration Space using the DEVICE_IO protocol instance **`GlobalIoFcns`**.

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

Parameters

`Port` The address of the PCI Configuration read operation.

Description

This function reads an 8 bit value from PCI Configuration Space at the address specified by *`Port`*.

4.11.15 writepci8 Macro

The **`writepci8()`** macro writes an 8 bit value to PCI Configuration Space using the DEVICE_IO protocol instance **`GlobalIoFcns`**.

```
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 PCI Configuration Space at the address specified by *`Port`*.

4.11.16 readpci16 Macro

The **readpci16()** macro reads a 16 bit value from PCI Configuration Space using the DEVICE_IO protocol instance **GlobalIoFcns**.

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

Parameters

Port	The address of the PCI Configuration read operation.

Description

This function reads a 16 bit value from PCI Configuration Space at the address specified by *Port*.

4.11.17 writepci16 Macro

The **`writepci16()`** macro writes a 16 bit value to PCI Configuration Space using the DEVICE_IO protocol instance **`GlobalIoFcns`**.

```
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 PCI Configuration Space at the address specified by *`Port`*.

4.11.18 readpci32 Macro

The **readpci32()** macro reads a 32 bit value from PCI Configuration Space using the DEVICE_IO protocol instance **GlobalIoFcns**.

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

Parameters

Port	The address of the PCI Configuration read operation.

Description

This function reads a 32 bit value from PCI Configuration Space at the address specified by *Port*.

4.11.19 writepci32 Macro

The **writepci32()** macro writes a 32 bit value to PCI Configuration Space using the DEVICE_IO protocol instance **GlobalIoFcns**.

```
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 PCI Configuration Space at the address specified by *Port*.

4.12 Miscellaneous Functions and Macros

Table 4-20 lists some miscellaneous helper functions that are described in the following sections.

Table 4-20. Miscellaneous Functions and Macros

Name	Description
LibGetVariable	Retrieves and environment variable's value.
LibGetVariableAndSIze	Retrieves and environment variable's value and its size in bytes.
LibDeleteVariable	Remove a given variable from the variable store
CompareGuid	Compares two 128 bit GUIDs.
CR	Returns a pointer to a element's containing record.
DecimaltoBCD	Converts a decimal value to a 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.

4.12.1 LibGetVariable Function

The **LibGetVariable()** function returns the value of the specified variable.

```
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.

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.

4.12.2 LibGetVariableAndSize Function

The **LibGetVariableAndSize()** function returns the value of the specified variable and its size in bytes.

```
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.

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*.

4.12.3 LibDeleteVariable Function

The **LibDeleteVariable()** function returns the value of the specified variable and its size in bytes.

```
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.

VarrGuid A unique identifier for the vendor.

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 writter.

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

4.12.4 CompareGuid Function

The **CompareGuid()** function compares two GUIDs.

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

Parameters

Guid1 A pointer to a 128 bit GUID.

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 non zero value is returned.

Status Codes Returned

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

4.12.5 CR Macro

The **CR()** macro returns a pointer to an elements containing record .

```
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 type. record.
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.

4.12.6 DecimaltoBCD Function

The **`DecimaltoBcd()`** function converts a decimal value to a BCD value..

```
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.

4.12.7 BCDtoDecimal Function

The **`BCDtoDecimal()`** function converts a BCD value to a decimal value.

```
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.

4.12.8 LibCreateProtocolNotifyEvent Function

The **`LibCreateProtocolNotifyEvent()`** function creates a notification event and registers that event with all the protocol instances specified by *`ProtocolGuid`*.

```
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 GUID of the protocol upon whose installation the event is fired.
`NotifyTpl`	Supplies the task priority level of the event notifications.
`NotifyFunction`	Supplies the function to notify when the event is signaled.
`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 protocol of type *`ProtocolGuid`* instance that exists in the system when this function is invoked. In addition, every time a protocol of type *`ProtocolGuid`* instance is added, the notification function is also executed. This function returns the notification event that was created.

4.12.9 WaitForSingleEvent Function

The **WaitForSingleEvent()** function waits for a given event to fire, or for an optional timeout to expire.

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

Parameters

Event	The event to wait for.
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	*Timout* expired before *Event* fired..

4.12.10 WaitForEventWithTimeout Function

The `WaitForEventWithTimeout()` function prints a string for the given number of seconds until either the timeout expires, or the user presses a key.

```
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 to wait for.

`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.

`Key` Either the key the user pressed or *`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 a 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`*).

4.12.11 RtLibEnableVirtualMappings Function

The **`RtLibEnableVirtualMappings()`** function converts runtime pointers internal to the EFI Library to a new virtual base address.

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

Parameters

Description

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

4.12.12 RtConvertList Function

The **RtConvertList()** function converts all the pointers in a doubly linked list to a new virtual base address.

```
#define EFI_OPTIONAL_PTR             0x00000001
#define EFI_INTERNAL_FNC             0x00000002
#define EFI_INTERNAL_PTR             0x00000004

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

Parameters

DebugDisposition	A bitmask that describes the pointer types in the linked list.
ListHead	A pointer to a doubly linked list.

Description

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

4.12.13 LibGetSystemConfigurationTable Function

The **LibGetSystemConfigurationTable()** function returns a system configuration table that is stored in the EFI System Table based on the provided GUID.

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

Parameters

TableGuid	A pointer to the table's GUID type.
Table	On exit, a pointer to a system configuration table.

Description

This function searches the list of configuration tables 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.