EFI Driver Library Specification

Draft for Review

Version 1.11

November 26, 2003

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 Corrected minor typographical errors. Fixed headers and footers.	7/31/01	Intel
1.10	Updated to match the EFI 1.10 Sample Implementation Release that matches the 0.9 draft of the EFI 1.10 Specification.	12/16/01	Intel
1.11	Updated to match the EFI 1.10 Sample Implementation Release 1.10.14.62 that matches the final draft of the EFI 1.10 Specification	11/26/03	Intel

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 Driver 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 Driver Library Specification.
Chapter 2: Functions and Macros	Definition of functions and macros exported by the EFI Driver Library.

1.2 Goals

The primary goal of the EFI Driver 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 Driver 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 Related Information

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

- *Extensible Firmware Interface Specification*, Version 1.02, Intel Corporation, 2000.

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.

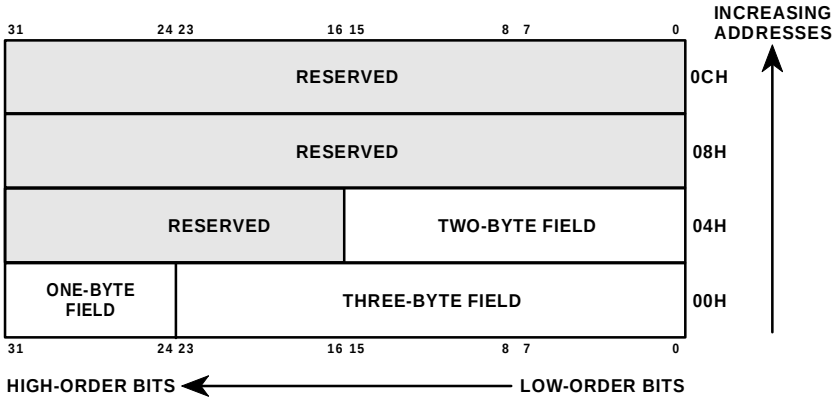

Figure 1-2. Memory Layout Conventions

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
Functions and Macros

The functions and macros exported by the EFI Driver 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

2.1 Initialization Functions

The initialization functions in the EFI Driver Library are used to initialize the execution environment so that other EFI Driver Library function may be used. Table 2-1 lists the initialization support functions that are described in the following sections.

Table 2-1 Initialization Functions

Name	Description
EfiInitializeDriverLib	Initializes the EFI Driver Library.
EfiLibInstallDriverBinding	Initializes the EFI Driver Library and installs a Driver Binding Protocol instance.
EfiLibInstallAllDriverProtocols	Initializes the EFI Driver Library and install the Driver Binding Protocol, the Component Name Protocol, the Driver Configuration Protocol, and the Driver Diagnostics Protocol instances.

2.1.1 EfiInitializeDriverLib Function

The **EfiInitializeDriverLib()** function initializes the EFI Driver Library.

```
EFI_STATUS
EfiInitializeDriverLib (
  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 Driver Library functions. Additional calls to this function are ignored. This function initializes all the global variables required by the EFI Driver Library functions. In addition, it verifies the CRCs for all the EFI system tables.

Status Codes Returned

EFI_SUCCESS	The EFI Driver Library was initialized.

2.1.2 EfiLibInstallDriverBinding Function

The **EfiLibInstallDriverBinding()** function initializes the EFI Driver Library and installs the Driver Binding Protocol.

```
EFI_STATUS
EfiLibInstallDriverBinding (
  IN EFI_HANDLE                   ImageHandle,
  IN EFI_SYSTEM_TABLE             *SystemTable,
  IN EFI_DRIVER_BINDING_PROTOCOL  *DriverBinding,
  IN EFI_HANDLE                   *DriverBindingHandle
  );
```

Parameters

ImageHandle	A handle for the image that is initializing the library.
SystemTable	A pointer to the EFI system table.
DriverBinding	A pointer to the EFI Driver Binding Protocol instance.
DriverBindingHandle	The handle on which to install the *DriverBinding* protocol. If this parameter is **NULL**, then *DriverBinding* is installed onto a newly created handle.

Description

This function initializes the EFI Driver Library by calling **EfiInitializeDriverLib()**. It then installs *DriverBinding* into the handle specified by *DriverBindingHandle*. If *DriverBindingHandle* is **NULL**, then *DriverBinding* is installed onto a newly created handle. *DriverBinding->ImageHandle* is initialized to *ImageHandle*, and *DriverBinding->DriverBindingHandle* is initialized to the handle on which *DriverBinding* was installed.

Status Codes Returned

EFI_SUCCESS	The EFI Driver Library was initialized and *DriverBinding* was installed.
Otherwise	*DriverBinding* could not be installed.

2.1.3 EfiLibInstallAllDriverProtocols Function

The **EfiLibInstallAllDriverProtocols()** function initializes the EFI Driver Library and installs the Driver Binding Protocol, the Component Name Protocol, the Driver Configuration Protocol, and the Driver Diagnostics Protocol.

```
EFI_STATUS
EfiLibInstallDriverBinding (
  IN EFI_HANDLE                       ImageHandle,
  IN EFI_SYSTEM_TABLE                 *SystemTable,
  IN EFI_DRIVER_BINDING_PROTOCOL      *DriverBinding,
  IN EFI_HANDLE                       *DriverBindingHandle,
  IN EFI_COMPONENT_NAME_PROTOCOL      *ComponentName,
  IN EFI_DRIVER_CONFIGURATION_PROTOCOL *DriverConfiguration,
  IN EFI_DRIVER_DIAGNOSTICS_PROTOCOL  *DriverDiagnostics
  );
```

Parameters

ImageHandle	A handle for the image that is initializing the library.
SystemTable	A pointer to the EFI system table.
DriverBinding	A pointer to the EFI Driver Binding Protocol instance.
DriverBindingHandle	The handle on which to install the *DriverBinding* protocol. If this parameter is **NULL**, then *DriverBinding* is installed onto a newly created handle.
ComponentName	A pointer to the EFI Component Name Protocol instance. This is an optional parameter that may be **NULL**.
DriverConfiguration	A pointer to the EFI Driver Configuration Protocol instance. This is an optional parameter that may be **NULL**.
DriverDiagnostics	A pointer to the EFI Driver Diagnostics Protocol instance. This is an optional parameter that may be **NULL**.

Description

This function initializes the EFI Driver Library and installs the Driver Binding Protocol by calling **EfiLibInstallDriverBinding()**. Then, it installs *ComponentName*, *DriverConfiguration*, and *DriverDiagnostics* onto *DriverBinding->DriverBindingHandle*. If *ComponnetName* is **NULL**, it is ignored. If *DriverConfiguration* is **NULL**, it is ignored. If *DriverDiagnostics* is **NULL**, then it is ignored. This function does all of the work required to initialize most EFI drivers that follow the EFI Driver Model.

Status Codes Returned

EFI_SUCCESS	The EFI Driver Library was initialized and *DriverBinding*, *ComponentName*, *DriverConfiguration*, and *DriverDiagnostics* were installed.
Otherwise	One of the protocols could not be installed.

2.2 Linked List Support Macros

The EFI Driver 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 **EFI_LIST_ENTRY** data structure. Each of the nodes in the linked list must also contain a **EFI_LIST_ENTRY** data structure. The **EFI_LIST_ENTRY** data structure simply contains a forward link and a backward link. The following is the definition of the **EFI_LIST_ENTRY** data structure.

```
typedef struct _EFI_LIST_ENTRY {
  struct _EFI_LIST_ENTRY  *Flink;
  struct _EFI_LIST_ENTRY  *Blink;
} EFI_LIST_ENTRY;
```

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

Table 2-2 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.
SwapListEntries	Swaps two nodes of a doubly linked list.

2.2.1 InitializeListHead Macro

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

```
VOID
InitializeListHead(
  EFI_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*.

2.2.2 IsListEmpty Macro

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

```
BOOLEAN
IsListEmpty(
  EFI_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

2.2.3 RemoveEntryList Macro

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

```
VOID
RemoveEntryList(
  EFI_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.

2.2.4 InsertTailList Macro

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

```
VOID
InsertTailList(
  EFI_LIST_ENTRY *ListHead,
  EFI_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*.

2.2.5 InsertHeadList Macro

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

```
VOID
InsertHeadList(
  EFI_LIST_ENTRY *ListHead,
  EFI_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*.

2.2.6 SwapListEntries Macro

The **SwapListEntries()** macro swaps two nodes in a doubly linked list.

```
VOID
SwapListEntries (
  EFI_EFI_LIST_ENTRY *Entry1,
  EFI_EFI_LIST_ENTRY *Entry2
  );
```

Parameters

Entry1	Element in the doubly linked list in front of Entry2.
Entry2	Element in the doubly linked list behind Entry1.

Description

Swap the location of the two elements of a doubly linked list. Entry2 is placed in front of Entry1. The list must have been initialized with **InitializeListHead()** before using this function.

2.3 Memory Support Functions

The EFI Driver Library provides a set of functions that operate on buffers in memory. Buffers can either be allocated on the stack, as global variables, or from the memory pool. To prevent memory leaks, it is the caller's responsibility to maintain buffers allocated from the memory pool. This means that the caller must free a buffer when that buffer is no longer needed. Table 2-3 contains the list of memory support functions that are described in the following sections.

Table 2-3 Memory Support Functions

Name	Description
EfiCopyMem	Copies one buffer to another buffer
EfiSetMem	Fills a buffer with a specified value
EfiZeroMem	Fills a buffer with zeros.
EfiCompareMem	Compares the contents of two buffers.
EfiLibAllocatePool	Allocates a buffer from the memory pool.
EfiLibAllocateZeroPool	Allocates a buffer from the memory pool and fills it with zeros.

2.3.1 EfiCopyMem Macro

The **EfiCopyMem()** macro copies one buffer to another buffer.

```
VOID
EfiCopyMem (
  IN VOID     *Destination,
  IN VOID     *Source,
  IN UINTN    Length
  );
```

Parameters

Destination	Pointer to the destination buffer.
Source	Pointer to the source buffer.
Length	The number of bytes to copy.

Description

This macro copies *Length* bytes from the buffer specified by *Source* to the buffer specified by *Destination*.

Status Codes Returned

None.

2.3.2 EfiSetMem Macro

The **EfiSetMem()** macro fills a buffer with a specified value.

```
VOID
EfiSetMem (
  IN VOID     *Destination,
  IN UINTN    Length,
  IN UINT8    Value
  );
```

Parameters

Destination	Pointer to the destination buffer.
Length	The number of bytes to fill.
Value	The byte value used to fill the buffer *Destination*.

Description

This macro fills *Length* bytes of the buffer specified by *Destination* with the value specified by *Value*.

Status Codes Returned

None.

2.3.3 EfiZeroMem Macro

The **EfiZeroMem()** macro fills a buffer with zeros.

```
VOID
EfiZeroMem (
  IN VOID     *Destination,
  IN UINTN    Length
  );
```

Parameters

Destination	Pointer to the destination buffer.
Length	The number of bytes to zero.

Description

This macro fills *Length* bytes of the buffer specified by *Destination* with zeros.

Status Codes Returned

None.

2.3.4 EfiCompareMem Function

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

```
INTN
EfiCompareMem (
  IN VOID     *MemOne,
  IN VOID     *MemTwo,
  IN UINTN    Len
  );
```

Parameters

MemOne	Pointer to the first buffer to compare.
MemTwo	Pointer to the second buffer to compare.
Len	Number of bytes to compare.

Description

This function compares *Len* bytes of *MemOne* to *Len* bytes of *MemTwo*. 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	MemOne is identical to MemTwo for Len bytes
≠ 0	MemOne is not identical to MemTwo for Len bytes

2.3.5 EfiLibAllocatePool Function

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

```
VOID *
EfiLibAllocatePool (
  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.

2.3.6 EfiLibAllocateZeroPool Function

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

```
VOID *
EfiLibAllocateZeroPool (
  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.

2.4 String Functions

The EFI Driver Library provides a set of functions. Table 2-3 contains the list of string functions that are described in the following sections.

Table 2-3 Memory Support Functions

Name	Description
EfiStrCpy	Copies one Unicode string to another Unicode string.
EfiStrLen	Determines the length of a Unicode string.
EfiStrSize	Determines the size of a Unicode string in bytes.
EfiStrCmp	Compares two Unicode strings.
EfiStrCat	Concatenates two Unicode strings.
EfiAsciiStrLen	Determines the length of an ASCII string
EfiAsciiStrCpy	Copies one ASCII string to another ASCII string
EfiAsciiStrnCmp	Compares two ASCII strings for a given length
EfiLibLookupUnicodeString	Looks up a Unicode string in a table of Unicode strings
EfiLibAddUnicodeString	Creates and/or adds a Unicode string to a table of Unicode strings
EfiLibFreeUnicodeStringTable	Frees the memory associated with a table of Unicode strings.

2.4.1 EfiStrCpy Function

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

```
VOID
EfiStrCpy (
  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*.

2.4.2 EfiStrLen Function

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

```
UINTN
EfiStrLen (
  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*.

2.4.3 EfiStrSize Function

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

```
UINTN
EfiStrSize (
  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.

2.4.4 EfiStrCmp Function

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

```
INTN
EfiStrCmp (
  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.

2.4.5 EfiStrCat Function

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

```
VOID
EfiStrCat (
  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*.

2.4.6 EfiAsciiStrLen Function

The **EfiAsciiStrLen()** function calculate the length of an ASCII string.

```
UINTN
EfiAsciiStrLen (
  IN CHAR8   *String
  );
```

Parameters

String	Pointer to a Null-terminated ASCII string.

Description

This function compute the length of an ASCII string.

2.4.7 EfiAsciiStrCpy Function

The **EfiAsciiStrCpy()** function copies one ASCII string to another ASCII string.

```
CHAR8 *
EfiAsciiStrCpy (
  IN CHAR8    *Destination,
  IN CHAR8    *Source
  );
```

Parameters

Destination	Pointer to a Null-terminated ASCII string.
Source	Pointer to a Null-terminated ASCII string.

Description

This function copies the contents of the ASCII string *Source* to the ASCII string *Destination*.

2.4.8 EfiAsciiStrnCmp Function

The **EfiAsciiStrnCmp()** function compares two ASCII strings for a length.

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

Parameters

s1	Pointer to a Null-terminated ASCII string.
s2	Pointer to a Null-terminated ASCII string.
len	Number of characters to compare.

Description

This function compares the ASCII string *s1* to the ASCII string *s2* for *len* characters. If the first *len* characters of *s1* is identical to the first *len* characters of *s2*, then 0 is returned. Otherwise, the position of the first mismatched ASCII character is returned.

Status Codes Returned

0	The substring of *s1* and *s2* is identical.
≠ 0	The substring of *s1* and *s2* is not identical.

2.4.9 EfiLibLookupUnicodeString Function

The **EfiLibLookupUnicodeString()** function returns a Unicode string of a specific language from a table of Unicode strings. This library function is used to implement the Component Name Protocol for EFI Drivers.

```
EFI_STATUS
EfiLibLookupUnicodeString (
  CHAR8                       *Language,
  CHAR8                       *SupportedLanguages,
  EFI_UNICODE_STRING_TABLE    *UnicodeStringTable,
  CHAR16                      **UnicodeString
  );
```

Parameters

Language	A pointer to the ISO 639-2 language code for the Unicode string to look up and return.
SupportedLanguages	A pointer to the set of ISO 639-2 language codes that the Unicode string table supports. *Language* must be a member of this set.
UnicodeStringTable	A pointer to the table of Unicode strings. Type **EFI_UNICODE_STRING_TABLE** is defined in "Related Definitions".
UnicodeString	A pointer to the Unicode string from *UnicodeStringTable* that matches the language specified by *Language*.

Related Definitions

```
typedef struct {
  CHAR8   *Language;
  CHAR16  *UnicodeString;
} EFI_UNICODE_STRING_TABLE;
```

Description

This function looks up a Unicode string in *UnicodeStringTable*. If *Language* is a member of *SupportedLanguages* and a Unicode string is found in *UnicodeStringTable* that matches the language code specified by *Language*, then it is returned in *UnicodeString*.

Status Codes Returned

EFI_SUCCESS	The Unicode string that matches the language specified by *Language* was found in the table of Unicoide strings *UnicodeStringTable*, and it was returned in *UnicodeString*.
EFI_INVALID_PARAMETER	*Language* is **NULL**.
EFI_INVALID_PARAMETER	*UnicodeString* is **NULL**.
EFI_UNSUPPORTED	*SupportedLanguages* is **NULL**.
EFI_UNSUPPORTED	*UnicodeStringTable* is **NULL**.
EFI_UNSUPPORTED	The language specified by *Language* is not a member of *SupportedLanguages*.
EFI_UNSUPPORTED	The language specified by *Language* is not supported by *UnicodeStringTable*.

2.4.10 EfiLibAddUnicodeString Function

The **EfiLibAddUnicodeString()** function adds a Unicode string of a specific language to a table of Unicode strings. This library function is used to implement the Component Name Protocol for EFI Drivers.

```
EFI_STATUS
EfiLibLookupUnicodeString (
  CHAR8                       *Language,
  CHAR8                       *SupportedLanguages,
  EFI_UNICODE_STRING_TABLE    **UnicodeStringTable,
  CHAR16                      *UnicodeString
  );
```

Parameters

Language	A pointer to the ISO 639-2 language code for the Unicode string to add.
SupportedLanguages	A pointer to the set of ISO 639-2 language codes that the Unicode string table supports. *Language* must be a member of this set.
UnicodeStringTable	A pointer to the table of Unicode strings. Type **EFI_UNICODE_STRING_TABLE** is defined in **EfiLibLookupUnicodeString()**.
UnicodeString	A pointer to the Unicode string to add.

Description

This function adds a Unicode string to *UnicodeStringTable*. If *Language* is a member of *SupportedLanguages* then *UnicodeString* is added to *UnicodeStringTable*. New buffers are allocated for both *Language* and *UnicodeString*. The contents of *Language* and *UnicodeString* are copied into these new buffers. These buffers are automatically freed when **EfiLibFreeUnicodeStringTable()** is called.

Status Codes Returned

EFI_SUCCESS	The Unicode string that matches the language specified by *Language* was found in the table of Unicode strings *UnicodeStringTable*, and it was returned in *UnicodeString*.
EFI_INVALID_PARAMETER	*Language* is **NULL**.
EFI_INVALID_PARAMETER	*UnicodeStringTable* is **NULL**.
EFI_INVALID_PARAMETER	*UnicodeString* is **NULL**.
EFI_INVALID_PARAMETER	*UnicodeString* is an empty string.
EFI_UNSUPPORTED	*SupportedLanguages* is **NULL**.
EFI_ALREADY_STARTED	A Unicode string with language *Language* is already present in *UnicodeStringTable*.
EFI_OUT_OF_RESOURCES	There is not enough memory to add another Unicode string to *UnicodeStringTable*.
EFI_UNSUPPORTED	The language specified by *Language* is not a member of *SupportedLanguages*.

2.4.11 EfiLibFreeUnicodeStringTable Function

The **EfiLibFreeUnicodeStringTable()** function frees a Unicode string table that was created with **EfiLibAddUnicodeString()**. This library function is used to implement the Component Name Protocol for EFI Drivers.

```
EFI_STATUS
EfiLibFreeUnicodeStringTable (
  EFI_UNICODE_STRING_TABLE  *UnicodeStringTable
  );
```

Parameters

UnicodeStringTable	A pointer to the table of Unicode strings. Type **EFI_UNICODE_STRING_TABLE** is defined in **EfiLibLookupUnicodeString()**.

Description

This function frees the table of Unicode strings in *UnicodeStringTable*. If *UnicodeStringTable* is **NULL**, then **EFI_SUCCESS** is returned. Otherwise, each language code, and each Unicode string in the Unicode string table are freed, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The Unicode string table was freed.

2.5 Text I/O Functions

The Text I/O functions in the EFI Driver 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. The format of argument descriptors is described below:

```
%[flags][width]type

flags:
  '-' - Left justify
  '+' - Prefix a sign
  ' ' - Prefix a blank
  ',' - Place commas in numberss
  '0' - Prefix for width with zeros
  'l' - UINT64
  'L' - UINT64

width:
  '*' - Get width from a UINTN argument from the argument list
  Decimal number that represents width of print

type:
  'X' - argument is a UINTN hex number, prefix '0'
  'x' - argument is a hex number
  'd' - argument is a decimal number
  's','a' - argument is an ascii string
  'S' - argument is an Unicode string
  'g' - argument is a pointer to an EFI_GUID
  't' - argument is a pointer to an EFI_TIME structure
  'c' - argument is an ascii character
  'r' - argument is EFI_STATUS
  '%' - Print a %
```

Table 2-5 contains the list of Text I/O functions and macros that are described in the following sections.

Table 2-5 Text I/O Functions

Name	Description
DEBUG	Sends a formatted ASCII string to the error console device.

2.5.1 DEBUG Macro

The **DEBUG()** macro sends a formatted ASCI string to the error console device.

```
DEBUG (
  IN UINTN    ErrorLevel,
  IN CHAR8    *Format,
  ...
  );
```

Parameters

ErrorLevel	If error level is set print the formatted ASCII string.
Format	A pointer to an ASCII string containing format information.
...	Variable length argument list.

Description

This function uses the format string *Format* and the variable argument list to build a formatted ASCII string. This formatted ASCII string is then sent to the error console device. The *Format* parameter may use the following to specify the format and types of parameters to display within the formatted ASCII string:

2.6 Math Functions

The EFI Driver Library provides a few math functions to operate on 64 bit operands. These include shift operations, multiplication and division. Table 2-6 lists the set of math functions that are described in the following sections.

Table 2-6 Math Functions

Name	Description
DriverLibLShiftU64	Shift a 64 bit integer left between 0 and 63 bits.
DriverLibRShiftU64	Shift a 64 bit integer right between 0 and 63 bits.
DriverLibMultU64x32	Multiply a 64 bit unsigned integer by a natural integer and generate a 64 bit unsigned result.
DriverLibDivU64x32	Divide a 64 bit unsigned integer by a natural unsigned integer and generate a 64 bit unsigned result with an optional natural remainder.

2.6.1 DriverLibLShiftU64 Function

The **DriverLibLShiftU64()** function shifts a 64 bit integer left between 0 and 63 bits.

```
UINT64
DriverLibLShiftU64 (
  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.

2.6.2 DriverLibRShiftU64 Function

The **DriverLibRShiftU64()** function shifts a 64 bit integer right between 0 and 63 bits.

```
UINT64
DriverLibRShiftU64 (
  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.

2.6.3 DriverLibMultU64x32 Function

The **DriverLibMultU64x32()** function multiples a 64 bit unsigned integer by an unsigned natural value and generates a 64 bit unsigned result.

```
UINT64
DriverLibMultU64x32 (
  IN UINT64   Multiplicand,
  IN UINTN    Multiplier
  );
```

Parameters

Multiplicand	A 64 bit unsigned value.
Multiplier	An unsigned natural multiplier.

Description

This function multiples the 64 bit unsigned value *Multiplcand* by the unsigned natural value *Multiplier* and generates a 64 bit unsigned result. This 64 bit unsigned result is returned.

2.6.4 DriverLibDivU64x32 Function

The **DriverLibDivU64x32()** function divides a 64 bit unsigned integer by an unsigned natural value and generates a 64 bit unsigned result and an unsigned natural remainder.

```
UINT64
DriverLibDivU64x32 (
  IN UINT64   Dividend,
  IN UINTN    Divisor,
  OUT UINTN   *Remainder OPTIONAL
  );
```

Parameters

Dividend	A 64 bit unsigned value.
Divisor	An unsigned natural value.
Remainder	A pointer to an unsigned natural value.

Description

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

2.7 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 2-7 lists the support functions for creating and maintaining spin locks. These functions are described in the following sections.

Table 2-7 Spin Lock Functions

Name	Description
EfiInitializeLock	Initialize a spin lock.
EfiAcquireLock	Acquire a spin lock.
EfiAcquireLockOrFail	Acquires a spin lock or returns an error if the spin lock is already locked.
EfiReleaseLock	Release a spin lock.

2.7.1 EfiInitializeLock Function

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

```
VOID
EfiInitializeLock (
  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.

2.7.2 EfiAcquireLock Function

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

```
VOID
EfiAcquireLock (
  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.

2.7.3 EfiAcquireLockOrFail Function

The **EfiAcquireLockOrFail()** function acquires ownership of a lock. If the lock is already owned , then an error is returned.

```
EFI_STATUS
EfiAcquireLockOrFail (
  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. If the lock is already owned, then **EFI_ACCESS_DENIED** is returned. Otherwise, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The lock was acquired.
EFI_ACCESS_DENIED	The lock could not be acquired because it is already owned.

2.7.4 EfiReleaseLock Function

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

```
VOID
EfiReleaseLock (
  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.

2.8 Device Path Support Functions

Table 2-8 lists the support functions for creating and maintaining device path data structures. These functions are described in the following sections.

Table 2-8 Device Path Support Functions

Name	Description
EfiDevicePathInstance	Retrieves the next device path instance from a device path.
EfiAppendDevicePath	Appends a device path to all the instances of another device path.
EfiAppendDevicePathNode	Appends a device path node to all the instances of a device path.
EfiAppendDevicePathInstance	Appends a device path instance to a device path.
EfiFileDevicePath	Appends a file path to a device path.
EfiDevicePathSize	Returns the size of a device path in bytes.
EfiDuplicateDevicePath	Creates a new copy of a device path.

2.8.1 EfiDevicePathInstance Function

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

```
EFI_DEVICE_PATH_PROTOCOL *
EfiDevicePathInstance (
  IN OUT EFI_DEVICE_PATH_PROTOCOL  **DevicePath,
  OUT UINTN                        *Size
  );
```

Parameters

DevicePath	A pointer to a device path data structure.
Size	A pointer to the size in bytes of a device path instance.

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**.

2.8.2 EfiAppendDevicePath Function

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

```
EFI_DEVICE_PATH_PROTOCOL *
EfiAppendDevicePath (
  IN EFI_DEVICE_PATH_PROTOCOL      *Src1,
  IN EFI_DEVICE_PATH_PROTOCOL      *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.

2.8.3 EfiAppendDevicePathNode Function

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

```
EFI_DEVICE_PATH_PROTOCOL *
EfiAppendDevicePathNode (
  IN EFI_DEVICE_PATH_PROTOCOL      *Src1,
  IN EFI_DEVICE_PATH_PROTOCOL      *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.

2.8.4 EfiAppendDevicePathInstance Function

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

```
EFI_DEVICE_PATH_PROTOCOL *
EfiAppendDevicePathInstance (
  IN EFI_DEVICE_PATH_PROTOCOL      *Src,
  IN EFI_DEVICE_PATH_PROTOCOL      *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.

2.8.5 EfiFileDevicePath Function

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

```
EFI_DEVICE_PATH_PROTOCOL *
EfiFileDevicePath (
  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.

2.8.6 EfiDevicePathSize Function

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

```
UINTN
EfiDevicePathSize (
  IN EFI_DEVICE_PATH_PROTOCOL      *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.

2.8.7 EfiDuplicateDevicePath Function

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

```
EFI_DEVICE_PATH_PROTOCOL *
EfiDuplicateDevicePath (
  IN EFI_DEVICE_PATH_PROTOCOL      *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.

2.9 Miscellaneous Functions and Macros

Table 2-9 lists some miscellaneous helper functions that are described in the following sections.

Table 2-9 Miscellaneous Functions and Macros

Name	Description
EfiCompareGuid	Compares two 128 bit GUIDs.
CR	Returns a pointer to an element's containing record.
EfiLibCreateProtocolNotifyEvent	Creates a notification event that fires every time a protocol instance is created.
EfiLibGetSystemConfigurationTable	Retrieves a system configuration table from the EFI System Table.

2.9.1 EfiCompareGuid Function

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

```
BOOLEAN
EfiCompareGuid(
  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 **TRUE** is returned. If there are any bit differences in the two GUIDs, then **FALSE** is returned.

Status Codes Returned

TRUE	The two GUIDs are identical.
FALSE	The two GUIDs are not identical

2.9.2 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.

2.9.3 EfiLibCreateProtocolNotifyEvent Function

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

```
EFI_EVENT
EfiLibCreateProtocolNotifyEvent(
  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.

2.9.4 EfiLibGetSystemConfigurationTable Function

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

```
EFI_STATUS
EfiLibGetSystemConfigurationTable (
  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.