EFI Application Toolkit Socket Protocol Interface External Product Specification

Revision 0.9

August 28, 2000

ESG Server Software Technologies (SST)

Revision History

Date	Revision	Modifications
10/11/99	0.1	Initial Outline
11/4/99	0.3	Internal release to EFI Developers Forum.
12/3/99	0.7	Released for customer review
01/24/00	0.8	Released as part of the 0.7 Application Developers Toolkit
08/28/00	0.9	Added PPP specifics

Disclaimers

Table Of Contents

Table Of Figures

Table Of Tables

This page intentionally left blank

1 Introduction

Network connectivity has become a fundamental capability for any computer system. Ordinarily, we think of the production OS providing this service. However, it today's managed PC environment, this capability is required from power off (through platform management controllers mounted on the baseboard) through pre-boot operations for remote system configuration and diagnostic purposes.

In the pre-boot space, the Extensible Firmware Interface (EFI) boot time services domain is becoming the standard operating environment. To enable EFI application development in this space, a standard network interface needs to be defined. This document specifies a BSD socket compatible interface for this purpose. The socket interface was chosen because it is network protocol independent and the most commonly used interface for applications. As a reference implementation for the EFI Application Toolkit, this socket interface is incorporated into a port of the FreeBSD TCP/IPv4 protocol stack, which in turn utilizes the EFI Simple Network Interface (SNI) protocol.

The EFI Socket Protocol Interface provides one of the core components of the EFI Application Toolkit. In addition to defining a standard application network interface, it describes the architecture of a complete TCP/IPv4 reference implementation.

1.1 Scope

This specification defines the EFI Socket protocol interface and describes how it was integrated into a port of the FreeBSD TCP/IP stack. In addition, it describes how the TCP/IP stack interfaces to the EFI SNI protocol and a socket library interface that facilitates EFI application portability. The end result is a design document that describes a complete network stack within an EFI boot services environment.

1.2 Target Audience

This document targets individuals who wish to understand the product functionality provided and the implementation details of a socket protocol implementation under EFI. It forms the basis or a user manual for the EFI Socket Protocol interface.

1.3 Reference Documents

The following documents were useful in preparing this specification:

- *Extensible Firmware Interface Specification*. Version 0.91, July 30, 1999.
- *EFI Developer's Guide*. Version 0.2, July 14, 1999.
- *EFI Application Toolkit Product Requirements Document*. Revision 0.97, Sept. 27, 1999.
- *EFI Application Toolkit C Library (libc) Port Specification*. Revision 0.01 Sept. 30, 1999
- *Preboot Execution Environment (PXE) Specification*. Version 2.0 (32/64 bit)
- *The FreeBSD General Commands, System Calls, and Library Functions Manuals,* 4th Berkeley Distribution, April 19, 1994.

1.4 Document Conventions

This document uses typographic and illustrative conventions described below.

1.4.1 Data Structure Descriptions

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. All implementations designed to conform to EFI Socket Protocol Interface will use “little endian” operation.

1.4.2 Typographic Conventions

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

`Prototype`	This typeface is used to indicate prototype code.
Argument	This typeface is used to indicate arguments and function name references.
`Name`	This typeface is used to indicate actual code or a programming construct.

2 Socket Reference Implementation Architecture

2.1 Overview

The EFI Socket Protocol is based on BSD socket interface paradigm. This paradigm is useful on several levels. First, the socket interface is network transport independent. Although most common implementations interface to a TCP/IPv4 network stack, it is just as applicable to a TCP/IPv6, IPX, NetBEUI, or Appltalk, network stack. Second, the socket interface is the de facto standard networking interface for applications.

The reference implementation described here outlines a complete implementation that includes a socket library, TCP/IPv4 network stack implemented as an EFI protocol, and a point-to-point serial protocol (PPP) module implemented as an EFI protocol. From an EFI perspective, the most important aspect of the architecture is the Socket Protocol Interface specification implemented by the TCP/IPv4 stack. By implementing to the EFI Socket Protocol Interface, new and/or different network stack implementations can be run on a platform without requiring a recompile of the applications that use them. Graphically, the architecture of the stack is as follows:

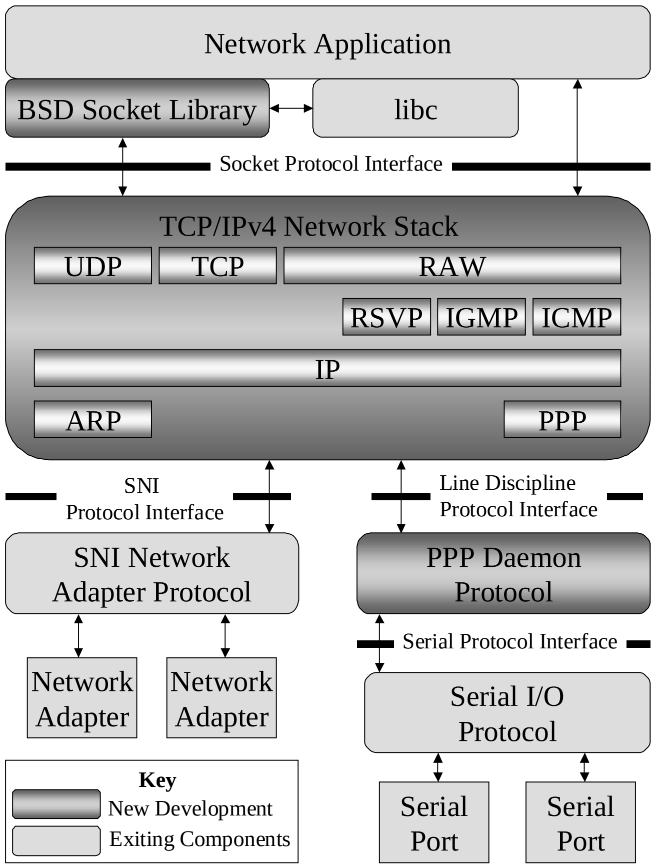

Figure 2-1 Network Stack Architecture

2.2 BSD Socket Library

Although an application is free to use the EFI Socket Protocol Interface directly, it will generally be more convenient to use the BSD compatible socket library *libsocket.lib*. In addition to supplying the core system calls such as *socket(), bind(), accept(),* etc., it also supplies the name resolution and utility routines such as *gethostbyname(), gethostbyaddr(), gethostent(), inet_addr(),* etc.

The bulk of the implementation is derived from a port of FreeBSD source. Network applications will use the include files normally associated with a FreeBSD Unix distribution (/usr/include) which are distributed with the implementation. **When using the socket library interface, the structure definitions and parameter values specified in the FreeBSD include files supersede those defined in this specification. The socket library will make the appropriate conversions and translations where needed.**

2.2.1 Library Initialization

Before the socket library can be generally used, a call to **EfiSocketInit()** must be made. This function takes two parameters: a pointer to the handle of the EFI image including the library and a pointer to the EFI system table. The initialization sequence will include calling the libc initialization routine and mapping the **`EFI_SOCKET_PROTOCOL`** GUID and "**socket:**" device to the libc file I/O subsystem.

Alternatively, the application may make an explicit call to the libc initialization routine *InitializeLibC()* or an implicit call through a libc runtime startup function (e.g. *_start_shellapp_a()* without making an explicit call to **EfiSocketInit()**. In this case, socket library initialization will take place on the first call to *socket()* which will obtain the image handle and system table pointer from the C library.

Both the C and socket library initialization calls are idempotent. If called, they may be called in any order and multiple times without ill effect.

2.2.2 Libc File I/O Integration

One of the advantages of using the socket library as the primary interface to the network stack is the integration of socket descriptors into the file I/O subsystem of the EFI Toolkit Libc implementation. This allows an application to use *read(), write(),* and *ioctl()* calls on a socket just as they would under a typical Unix environment. This facility is not available when using the native EFI Socket Protocol Interface. The socket library will hook into the I/O facility of libc by registering the device name "**socket:**" and associate file descriptor translation routines (*open, close, read, write, ioctl, lseek,* and *fstat*). When an application calls *socket()*, the socket library will make the following libc call:

```
fd = open("socket:", O_RDWR);
```

returning **`fd`** as the socket descriptor. The system calls *lseek()* and *fstat()* will return **`EOPNOTSUPP`**.

2.2.3 Library Routines

The following table details all routines that can operate on socket descriptors. Those noted with an asterisk are made available through *libc.lib*. All others are made available in *libsocket.lib*. The reader is referred to the man pages supplied with the reference implementation for interface details. Calls with a double asterisk note non-standard (from a libc perspective) system calls.

accept()	getsockopt()	recv()	setsockopt()
bind()	ioctl()*	recvfrom()	shutdown()
close()*	listen()	read()	socket()
connect()	open()*	send()	write()*
EfiSocketInit()**	pollsocket()**	sendto()	

Table 2-1 System Calls

The remaining tables detail the utility routines associated with the network stack. All routines are made available through *libsocket.lib*.

endhostent()	gethostent()	hstrerror()
gethostbyaddr()	int h_errno	sethostent()
gethostbyname()	herror()	

Table 2-2 Host Entry Routines

inet_addr()	inet_lnaof()	inet_netof()	inet_ntoa()
inet_aton()	inet_makeaddr()	inet_network()	

Table 2-3 Internet Address Manipulation Routines

dn_comp()	res_init()	res_query()	res_send()
dn_expand()	res_mkquery()	res_search()	

Table 2-4 Domain Name Server Resolver Routines

htonl()	htons()	ntohl()	ntohs()

Table 2-5 Byte Order Routines

2.3 TCP/IPv4 Network Stack

The reference TCP/IPv4 network stack is a port of the FreeBSD implementation. This includes implementations of IPv4, TCP, UDP, ARP, and ICMP. The stack also includes support for the Internet Group Management Protocol (IGMP) and the Resource ReSerVation Protocol (RSVP).

The stack is implemented as an EFI protocol that is available for concurrent use by an application and one or more EFI protocols. Access to the network stack is through the EFI Socket Protocol Interface. A complete description of the protocol interface can be found in section 3.

2.3.1 Porting Methodology

The vast majority of the FreeBSD implementation was used without modification. The sections of source that did not apply (such as signals and privilege checking) are conditionally compiled out using the following construct:

```
#ifdef _ORG_FREEBSD_
```

Using this approach, rather than deleting the non-applicable code, provides a means for documenting the port. This should facilitate ports of subsequent releases of the FreeBSD networking code.

The significant aspects of the port were in porting FreeBSD kernel services. These were primarily in memory management (*kmem_alloc, kmem_malloc, kmem_suballoc, malloc*, and *free*), setting priority levels (*spl, splx, splnet, splhigh,* and *splimp*), preemption (*tsleep* and *wakeup*), and timer support (*hardclock* and *softclock*). The file **efi_kern_support.c** contains all emulated kernal calls.

The other porting issues involved interfacing the FreeBSD network adapter interface to the EFI SNI protocol. At TCP/IPv4 protocol initialization, the EFI boot service *RegisterProtocolNotify()* is called to receive notifications of SNI protocol installations. When the notification is received, a `softc_t` type is allocated with the corresponding `struct ifnet` structure initialized to point to SNI specific routines for *if_init, if_start, if_poll_recv, if_poll_xmit, if_ioctl, if_watchdog,* and *if_poll_intern*. Finally, the FreeBSD routine *if_attach()* is called to make the interface available to the network stack. The file **efi_netiface.c** contains all SNI routines.

2.3.2 Network Stack Runtime Initialization

As with a FreeBSD system, all network stack configuration is performed at runtime. The design envisions a startup script (such as **startup.nsh** for the EFI shell) that will perform the same configuration operations that FreeBSD “rc” files do. This includes setting the IP address and network masks for all network interfaces and possibly setting routing information.

This approach requires configuration utilities similar to the FreeBSD ifconfig(8) and route(8) utilities. These utilities may need to take into account that the platform may have gone through a multi-stage platform management bring-up. In this case, the utilities would want to configure the network stack to use the same IP address and network mask that was used in a prior stage of platform bring-up. The high level design of the network configuration and diagnostic utilities are beyond the scope of this document.

3 EFI Socket Protocol Interface

This section defines the EFI Socket Protocol Interface. The intention of this interface is to provide a consistent programmatic interface to an EFI based network stack. The implementation and architecture of the network stack is not defined by this interface. Refer to Section 2.3 for an overview of a TCP/IPv4 implementation of this protocol interface.

3.1 EFI_SOCKET_INTERFACE Protocol

The **EFI_SOCKET_INTERFACE** protocol may be used to interface to one or more network protocol stacks.

```
#define EFI_SOCKET_PROTOCOL \
     { 198de03a-69fb-11d3-a714-00a0c972370d }
#define EFI_SOCKET_INTERFACE_REVISION   0x10000

#define EFI_SOCKERR(val)               EFIERR_OEM(0x10000 | val)

//
//  In addition to standard EFI status codes, this specification
//  defines additional return values which are compatible with
//  the EFI_ERROR() macro
//
```

Define	Value	Meaning
EFI_SOCKERR_FAILURE	EFI_SOCKERR(1)	Implementation specific error
EFI_SOCKERR_ADDRINUSE	EFI_SOCKERR(2)	Requested address is in use
EFI_SOCKERR_ADDRNOTAVAIL	EFI_SOCKERR(3)	Can not assign requested address
EFI_SOCKERR_AFNOSUPPORT	EFI_SOCKERR(4)	Address family not supported
EFI_SOCKERR_CONNABORTED	EFI_SOCKERR(5)	Software caused connection abort
EFI_SOCKERR_CONNREFUSED	EFI_SOCKERR(6)	Remote end refused connection
EFI_SOCKERR_CONNRESET	EFI_SOCKERR(7)	Connection closed by remote end
EFI_SOCKERR_HOSTUNREACH	EFI_SOCKERR(8)	Unable to determine route to host
EFI_SOCKERR_INPROGRESS	EFI_SOCKERR(9)	A blocking call is in progress, or the interface is still processing a callback function
EFI_SOCKERR_ISCONN	EFI_SOCKERR(10)	Socket is already connected
EFI_SOCKERR_MSGSIZE	EFI_SOCKERR(11)	Message too long for protocol
EFI_SOCKERR_NETDOWN	EFI_SOCKERR(12)	Network interface associated with socket is not available
EFI_SOCKERR_NETUNREACH	EFI_SOCKERR(13)	Network unreachable from this host
EFI_SOCKERR_NOTCONN	EFI_SOCKERR(14)	Socket is not connected
EFI_SOCKERR_WOULDBLOCK	EFI_SOCKERR(15)	Returned by non-blocking socket when

Define	Value	Meaning
		operation would normally block

```
typedef UINT32  SOCKET;
typedef UINT32  SOCK_EVENTS;

typedef struct {
    UINT8  AddressFamily;
    UINT8  AddressData[14];
} EFI_SOCKETADDR;

typedef struct {
    UINT32  Domain;
    UINT32  Type;
    UINT32  Protocol;
 } EFI_SOCKET_PROTO;

//
//  The following structures are used in GetSockOpt()
//  and SetSockOpt() calls.
//
typedef struct {
    UINT32 Seconds;
    UINT32 Microseconds;
} EFI_SOCK_TIMEOUT;

typedef struct {
    UINT32 OnOff;
    UINT32 Seconds;
} EFI_SOCK_LINGER;

typedef struct _EFI_SOCKET {
    UINT64              Revision;
    EFI_GETVENDORGUID   GetVendorGuid;
    EFI_GETPROTOCOLS    GetProtocols;
    EFI_SOCKET          Socket;
    EFI_BIND            Bind;
    EFI_LISTEN          Listen;
    EFI_ACCEPT          Accept;
    EFI_CONNECT         Connect;
    EFI_SEND            Send;
    EFI_RECEIVE         Receive;
    EFI_POLLSOCKET      PollSocket;
    EFI_GETSOCKOPT      GetSockOpt;
    EFI_SETSOCKOPT      SetSockOpt;
    EFI_SHUTDOWN        Shutdown;
    EFI_SOCKETIOCTL     SocketIoctl;
    EFI_GETPEERADDR     GetPeerAddr;
```

```
    EFI_GETSOCKADDR       GetSockAddr;
    EFI_CLOSESOCKET       CloseSocket;
    EFI_GETLASTERROR      GetLastError;
} EFI_SOCKET_INTERFACE;
```

Parameters

Revision	Defines the revision of the **EFI_SOCKET_INTERFACE** structure and interface semantics. All future revisions will be backward compatible to the current revision
GetProtocols	Returns a list of the network protocols supported by this interface.
Socket	Create an endpoint for communication.
Bind	Bind assigns an address to an unnamed socket.
Listen	Listen for connections on a socket.
Accept	Accept a connection on a socket.
Connect	Initiate a connection on a socket.
Send	Send data on a socket.
Receive	Receive data from a socket.
PollSocket	Poll for completed operations on non-blocking mode sockets.
GetSockOpt	Get current socket options.
SetSockOpt	Set socket options.
Shutdown	Shut down socket send and/or receive operations
SocketIoctl	Perform implementation specific control operations
CloseSocket	Close communication endpoint
GetLastError	Retrieve the last implementation specific error associated with a socket.

Members of the **EFI_SOCKADDR** type are defined as follows:

AddressFamily	Defines the format of the address data.
AddressData	Variable address data. Although defined at 14 bytes for memory allocation convenience, the actual length is address family specific and may be shorter or longer.

Member of the **EFI_SOCKET_PROTO** type are defined as follows:

Domain	Defines the communications domain which typically defines the semantics and format of the associated network address.
Type	Defines the communications semantics such as connection-oriented or datagram.
Protocol	Defines the network protocols supported.

Description

Although not a 1-to-1 mapping to a traditional socket interface, the **`SOCKET`** protocol provides a familiar programmatic paradigm to a network stack. The socket interface abstraction is network protocol independent and an implementation of the EFI **`SOCKET`** protocol may support one or more network protocols. It is also possible to have multiple **`SOCKET`** protocol implementations loaded and available at one time, presumably supplying an interface to non-overlapping network protocol implementations.

3.1.1 SOCKET.GetVendorGuid Function

The **GetVendorGuid()** function returns the vendor GUID of the network protocol implementation.

```
EFI_STATUS
(EFIAPI *EFI_GETVENDORGUID) {
     IN  EFI_SOCKET_INTERFACE    *This,
     OUT EFI_GUID                *VendorGuid
     );
```

Parameters

This A pointer to the **EFI_SOCKET_INTERFACE** instance.

VendorGuid On output, the buffer will contain the vendor GUID.

Description

The **GetVendorGuid()** function is used to determine the vendor of the network protocol implementation. This is particularly useful for network applications or higher level interface libraries that that assume implementation specific features that fall outside the scope of the EFI Socket Protocol Interface specification.

Status Codes Returned

EFI_SUCCESS	The vendor GUID was successfully returned.

3.1.2 SOCKET.GetProtocols Function

The **`GetProtocols()`** function returns the network protocols supported by the underlying implementation.

```
EFI_STATUS
(EFIAPI *EFI_GETPROTOCOLS) {
     IN  EFI_SOCKET_INTERFACE    *This,
     IN OUT UINTN               *ArraySize,
     OUT EFI_SOCKET_PROTO        *ProtoArray
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
ArraySize	On input, the size (in elements) of the *ProtoArray* buffer. On output, the number of elements actually returned.
ProtoArray	On output, an array of **`EFI_SOCKET_PROTO`** elements that represent the network protocols supported by this instance of the interface.

Description

The **`GetProtocols()`** function is used to determine the supported network protocols and semantics for this instance of the **`SOCKET`** interface. It returns an array of **`EFI_SOCKET_PROTO`** elements that specify all valid network domains, communication semantics, and network protocols that can be used in the **`Socket()`** function. If the *ArraySize* is too small to fit the results, the function returns **`EFI_BUFFER_TO_SMALL`** and updates *ArraySize* to indicate the size of the array needed.

In combination with the EFI boot service function **`LocateHandle()`**, an application can use this call to find the **`SOCKET`** interface that supports the desired network protocol(s).

The values returned are ultimately implementation dependent. However, in order to promote application compatibility, this specification defines certain values for the Internet address domain as follows:

Domain	Value
Internet	2

Communication Semantics	**Value**
Connection-oriented Stream	1
Datagram	2
Raw	3
Reliable Deliver Message	4
Sequenced Packet Stream	5

The following table of network protocols is a representative list. A complete list can be found in RFC1700. Note that the IP and RAW protocols use values that are undefined or reserved by RFC1700.

Network Protocol	**Value**
IP	0
ICMP	1
TCP	6
UDP	17
RAW	255

Status Codes Returned

EFI_SUCCESS	The result array of **EFI_SOCKET_PROTO** was returned
EFI_BUFFER_TOO_SMALL	The *ArraySize* is too small for the result. *ArraySize* has been updated with the size needed to complete the request.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.3 SOCKET.Socket Function

The **`Socket()`** function returns a socket descriptor representing an endpoint for communication.

```
EFI_STATUS
(EFIAPI *EFI_SOCKET) (
     IN  EFI_SOCKET_INTERFACE   *This,
     IN SOCKET                  *Socket,
     IN  UINT32                 Domain,
     IN  UINT32                 Type,
     IN  UINT32                 Protocol
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	A pointer to storage of returned socket descriptor.
Domain	The communications domain of the socket.
Type	The type of communications semantic for the socket.
Protocol	The specific protocol to be used with the socket

Description

The **Socket()** function creates an unnamed socket in the specified communications domain. The descriptor returned in *Socket* is used as an identifier for subsequent function calls that operate on a socket. The values used for the *Domain*, *Type*, and *Protocol* are defined by the underlying network protocols associated with **`EFI_SOCKET_INTERFACE`**. A list of supported domains, types, and protocols for the underlying network stack can be generated through the **GetProtocols()** interface function.

The *Domain* parameter specifies the communications domain of the socket. This is typically associated with a particular address family that defines the contents and semantics of an **`EFI_SOCKADDR.`** The *Type* parameter specifies the communications semantics of the socket. Example semantics include connection-oriented services, datagram services, reliable delivery, and raw network access. The *Protocol* parameter specifies the network protocol to use for this socket. Many network stack implementations will use a default protocol to meet the communications semantics specified in the *type* parameter if this value is zero.

Status Codes Returned

EFI_SUCCESS	A socket was successfully created
EFI_OUT_OF_RESOURCES	Insufficient resources are available to support the socket
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.4 SOCKET.Bind Function

The **Bind()** function assigns an address to an unnamed socket.

```
EFI_STATUS
(EFIAPI *EFI_BIND) (
      IN   EFI_SOCKET_INTERFACE   *This,
      IN   SOCKET                 Socket,
      IN   EFI_SOCKADDR           *Addr,
      IN   UINT32                 AddrLen
      );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
Addr	Pointer to **EFI_SOCKADDR** that names the socket.
AddrLen	Length, in bytes, of relevant data in **EFI_SOCKADDR**.

Description

The **Bind()** function assigns an address to an unnamed socket making it addressable by another entity on the network. The semantics of the address are defined by the communications domain specified when the socket was created.

Status Codes Returned

EFI_SUCCESS	The socket was successfully named.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_SOCKERR_ADDRINUSE	Requested address is already in use.
EFI_SOCKERR_ADDRNOTAVAIL	The requested address can not be assigned to the socket.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.5 SOCKET.Listen Function

The **`Listen()`** function places the socket in a listen state and limits the number of incoming connections that will be queue.

```
EFI_STATUS
(EFIAPI *EFI_LISTEN) (
     IN  EFI_SOCKET_INTERFACE    *This,
     IN  SOCKET                  Socket,
     IN  UINT32                  Backlog
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	Socket descriptor associated with this request.
Backlog	Limits the number of pending connections.

Description

Sockets created with a connection-oriented communications semantic may require the socket to be placed in a listening state. The **Listen()** function allows for that state transition and specifies an upper bound on the number of pending connections that will be queued through the *Backlog* parameter. Once the queue limit has been reached, the underlying protocol is free to reject further connection attempts.

Status Codes Returned

EFI_SUCCESS	The socket was successfully placed into the listen state.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_UNSUPPORTED	The socket is not of a type that supports this operation.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.6 SOCKET.Accept Function

The **`Accept()`** function accepts a new connection on a socket.

```
EFI_STATUS
(EFIAPI *EFI_ACCEPT) (
     IN  EFI_SOCKET_INTERFACE   *This,
     IN  SOCKET                 Socket,
     OUT SOCKET                 *NewSocket,
     OUT EFI_SOCKADDR           *Addr,
     IN OUT UINT32              *AddrLen
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	Socket descriptor associated with initial *Listen()* request.
NewSocket	New socket associated with accepted connection.
Addr	Pointer to **`EFI_SOCKADDR`** that receives the address of the remote endpoint.
AddrLen	On input specifies the size, in bytes, of the **`EFI_SOCKADDR`** buffer. On return, specifies the length of relevant data in **`EFI_SOCKADDR`**.

Description

The **Accept()** function extracts the first connection request on the queue of pending connections, creates a new socket with the same properties as *Socket* and allocates a new socket descriptor which is returned in *NewSocket*.

The address of the remote end of the connection is placed in *Addr* and *AddrLen* is updated to indicate the size of the relevant **`EFI_SOCKADDR`** data.

Underlying network implementations may support non-blocking I/O. If the socket does not support, or has not been configured for, non-blocking I/O and no pending connections are present on the queue, this call will block until a connection request is received. If the socket has been configured for non-blocking I/O and no pending connections are present on the queue, **Accept()** returns **`EFI_SOCKERR_WOULDBLOCK`**. In this case, subsequent calls to **PollSocket()** can determine when a connect request has been received at which time **Accpet()** would be called to remove the request from the queue.

Status Codes Returned

EFI_SUCCESS	A socket was successfully created for new connection.
EFI_INVALID_PARAMETER	Socket specified an invalid socket descriptor.
EFI_OUT_OF_RESOURCES	Insufficient resources available to support the new socket
EFI_UNSUPPORTED	The **Listen()** function has not be called on this socket.
EFI_SOCKERR_WOULDBLOCK	Operation can not complete without blocking.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.7 SOCKET.Connect Function

The **Connect()** function names the remote end of a socket.

```
EFI_STATUS
(EFIAPI *EFI_CONNECT) (
     IN  EFI_SOCKET_INTERFACE   *This,
     IN  SOCKET                 Socket,
     IN  EFI_SOCKADDR           *Addr,
     IN  UINT32                 AddrLen
     );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
Addr	Pointer to **EFI_SOCKADDR** that names the remote end of the socket.
AddrLen	Length, in bytes, of relevant data in **EFI_SOCKADDR**.

Description

The **Connect()** function names the remote end of a socket by specifying the address appropriate for the communications domain.

If the socket was created with a datagram communications semantic, this specifies the remote system with which the socket is to be associated for datagrams sent through the **Send()** function, and the only address from which datagrams are to be received through the **Receive()** function.

If the socket was created with a connection-oriented communications semantic, this call attempts to make a connection to another socket. The other socket is specified by name, which is an address in the communications domain of the socket. Each communications domain sets the semantics and format for the *Addr* parameter.

Unless the socket is configured for non-blocking I/O, this may cause the caller to block until the connection is established or the attempt is abandoned by the underlying network implementation. For non-blocking sockets, **EFI_SOCKERR_WOULDBLOCK** may be returned. In this case, subsequent calls to **PollSocket()** can determine when the request has completed.

Generally, connection-oriented sockets may successfully call **Connect()** only once; datagram sockets may use **Connect()** multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

Status Codes Returned

EFI_SUCCESS	The socket was successfully connected.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_TIMEOUT	Remote system did not respond to connection request.
EFI_SOCKERR_ADDRINUSE	The requested address is already in use.
EFI_SOCKERR_ADDRNOTAVAIL	The requested address not available on this machine.
EFI_SOCKERR_AFNOTSUPPORT	Specified address family can’t be used with this socket.
EFI_SOCKERR_CONNREFUSED	Connection refused by remote system.
EFI_SOCKERR_ISCONN	Socket is already connected.
EFI_SOCKERR_HOSTUNREACH	Interface was unable to determine a route to the host.
EFI_SOCKERR_NETUNREACH	A required network was unreachable from this system.
EFI_SOCKERR_WOULDBLOCK	Operation can not complete without blocking.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.8 SOCKET.Send Function

The **Send()** function sends data to the endpoint of the socket.

```
EFI_STATUS
(EFIAPI *EFI_SEND) (
      IN  EFI_SOCKET_INTERFACE   *This,
      IN  SOCKET                 Socket,
      IN  VOID                   *Buffer,
      IN  UINT32                 BufferSize,
      IN  UINT32                 Flags,
      IN  EFI_SOCKADDR           *Addr OPTIONAL,
      IN  UINT32                 AddrLen OPTIONAL,
      OUT UINT32                 *BytesSent
      );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
Buffer	Pointer to data to be sent.
BufferSize	Number of bytes to send.
Flags	Flags that affect the send operation.
Addr	Optional pointer to **EFI_SOCKADDR** containing the address of the receiver.
AddrLen	Optional length, in bytes, of relevant data in **EFI_SOCKADDR.**
BytesSent	The number of bytes actually sent.

Description

The **Send()** function sends data to the remote endpoint of a socket. If the remote endpoint of the socket has been named through the **Connect()** function, the *Addr* and *AddrLen* parameters are not required and may be NULL and 0 respectively.

The data associated with *Buffer* is under the control of the underlying network implementation and must not be altered until the call completes. The communications semantic and protocol for the socket may limit the amount of data that can be sent in a single call.

If the underlying network does not have the resources to complete the call, it may block the caller until it does. For network implementations that support non-blocking I/O, this condition will return **EFI_SOCKERR_WOULDBLOCK**. In this case, subsequent calls to **PollSocket()** can determine when the request has completed and the data associated with *Buffer* is no longer owned by the underlying network implementation.

The *Flags* parameter is a network implementation dependent value that can affect the way the send operation is handled. For application compatibility, this specification defines the following flag values.

Flag	Value	Description
Urgent Data	0x00000001	Send data as urgent if supported by underlying protocol.
Peek	0x00000002	**Not a valid send operation.**
Don't Route	0x00000004	Bypass routing
End of Record	0x00000008	Data complete record.
Wait For All Data	0x00000010	**Not a valid send operation.**
Implementation Dependent	0xffff0000	Bit fields available for implementation dependent flags.

Upon successful completion of this call, the *BytesSent* parameter is updated to indicate the number of bytes actually sent.

Status Codes Returned

EFI_SUCCESS	The send operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_OUT_OF_RESOURCES	Insufficient resources available for operation to succeed.
EFI_UNSUPPORTED	Sock does not support an option specified in *Flags*.
EFI_SOCKERR_AFNOTSUPPORT	Specified address family can't be used with this socket.
EFI_SOCKERR_CONNRESET	The connection was forcibly closed by the remote system.
EFI_SOCKERR_HOSTUNREACH	A route to the host is no longer available.
EFI_SOCKERR_MSGSIZE	Buffer size could not be supported by communications semantics of the socket.
EFI_SOCKERR_NOTCONN	The socket is not connected and *Addr* is NULL
EFI_SOCKERR_WOULDBLOCK	Operation can not complete without blocking.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.9 SOCKET.Receive Function

The **Receive()** function receives data from a socket.

```
EFI_STATUS
(EFIAPI *EFI_RECEIVE) (
      IN  EFI_SOCKET_INTERFACE   *This,
      IN  SOCKET                 Socket,
      OUT VOID                   *Buffer,
      IN  UINT32                 BufferSize,
      IN  UINT32                 Flags,
      OUT EFI_SOCKADDR           *Addr OPTIONAL,
      IN OUT UINT32              *AddrLen OPTIONAL,
      OUT UINT32                 *BytesReceived
      );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
Buffer	Pointer to receive buffer.
BufferSize	Size of receive buffer in bytes.
Flags	Flags that affect the receive operation.
Addr	Optional pointer to **EFI_SOCKADDR** that receives the address of the sender.
AddrLen	Optional length, in bytes, of relevant data in **EFI_SOCKADDR.**
BytesReceived	The number of bytes actually received.

Description

The **Receive()** function is used to receive data from the remote endpoint of a socket. If the remote endpoint of the socket has been named through the **Connect()** function, the *Addr* and *AddrLen* parameters are not required and may be NULL and 0 respectively.

Unless otherwise configured through **SetSockOpt()**, the underlying network implementation is free to return only the data that is available at the time of the call and not the amount requested. If no data is available, this call will block. For network implementations that support non-blocking I/O, this condition will return **EFI_SOCKERR_WOULDBLOCK**. In this case, subsequent calls to **Poll-Socket()** can determine then receive data becomes available and retrieved through additional calls to **Receive()**.

The *Flags* parameter is a network implementation dependent value that can affect the way receive operations are handled. For application compatibility, this specification defines the following flag values.

Flag	Value	Description
Urgent Data	0x00000001	Receive available urgent data if supported by underlying protocol.
Peek	0x00000002	Receive available data but do not remote it from the receive queue. Subsequent receive operations will return the same data.
Don't Route	0x00000004	**Not a valid receive operation.**
End of Record	0x00000008	**Not a valid receive operation.**
Wait For All Data	0x00000010	Wait until *BufferSize* bytes are received. The function may return a smaller amount of data if the connection is terminated or an error is pending for the socket.
Implementation Dependent	0xffff0000	Bit fields available for implementation dependent flags.

Upon successful completion of this call, the *BytesReceived* parameter is updated to indicate the number of bytes actually received.

Status Codes Returned

EFI_SUCCESS	The receive operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_OUT_OF_RESOURCES	Insufficient resources available for operation to succeed.
EFI_UNSUPPORTED	Sock does not support an option specified in *Flags*.
EFI_SOCKERR_CONNRESET	The connection was forcibly closed by the remote system.
EFI_SOCKERR_NOTCONN	The socket is associated with a connection-oriented communications semantic and has not been connected.
EFI_SOCKERR_WOULDBLOCK	Operation can not complete without blocking.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.10 SOCKET.PollSocket Function

The **PollSocket()** function checks for completed events on a non-blocking socket.

```
EFI_STATUS
(EFIAPI *EFI_POLLSOCKET) (
      IN  EFI_SOCKET_INTERFACE   *This,
      IN  SOCKET                 Socket,
      IN  SOCKET_EVENTS          EventMask,
      OUT SOCKET_EVENTS          *EventResults,
      );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
EventMask	Bit mask of events to check.
EventResutls	Pointer to storage for event results.

Description

The **PollSocket()** function provides a means of checking the state of a non-blocking socket. The *EventMask* specifies the events of interest and the value returned in *EventResult* indicates the completed events. The following event values are defined:

Definition	**Bits**
Incoming connect request pending	0x00000001
Outgoing connect request completed successfully	0x00000002
Connection reset or aborted	0x00000004
Normal receive data pending	0x00000008
Special receive data pending	0x00000010
May send normal data	0x00000020
May send special data	0x00000040
Bits available for implementation dependent definition.	0xffff0000

Status Codes Returned

EFI_SUCCESS	The poll operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.

EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.11 SOCKET.GetSockOpt Function

The **GetSockOpt()** function returns options on a socket.

```
EFI_STATUS
(EFIAPI *EFI_ GETSOCKOPT) (
     IN  EFI_SOCKET_INTERFACE   *This,
     IN  SOCKET                 Socket,
     IN  UINT32                 Level,
     IN  UINT32                 Option,
     OUT VOID                   *Buffer,
     IN OUT  UINT32             *BufferSize
     );
```

Parameters

This — A pointer to the **EFI_SOCKET_INTERFACE** instance.

Socket — Socket descriptor associated with this request.

Level — Protocol level at which the option applies.

Option — Identifies the option to retrieve.

Buffer — Pointer to storage for the option value.

BufferSize — On input, size in bytes of *Buffer*. On output, the size actually returned.

Description

The **GetSockOpt()** function retrieves the option values for the specified option. Options may exist at multiple protocol levels that can be specified with the *Level* parameter. All implementations must support the "Socket Level" which has a value of 0xffffffff. The options and return values are communications domain and network implementation dependent, however, this following option identifier values are defined to support application compatibility. Unless otherwise indicated, the size of the return values is 4 bytes (UINT32) and is treated as a Boolean values. The caller always supplies storage for returned values.

Option	**Value**	**Description**
Debug	1	Reports the state of underlying network debugging.
Accept Connection	2	Reports if socket is in listening state.
Reuse Address	3	Reports if **Bind()** allows reuse of local addresses.
Keep Alive	4	Reports if periodic messages are sent on connected socket.
Don't Route	5	Reports if outgoing messages bypass standard routing.

Option	**Value**	**Description**
Broadcast	6	Reports if broadcast datagrams can be sent on the socket.
Linger	7	Reports the state and timeout value for handling unsent messages when **CloseSocket()** is called. It returns an **`EFI_SOCK_LINGER`** structure.
Urgent Data Inline	8	Reports if urgent data is delivered from the standard receive queue.
Send Buffer Size	9	Report the size of the send buffer for the underlying network implementation.
Receive Buffer Size	10	Report the size of the receive buffer for the underlying network implementation.
Send Low-water Mark	11	Report the minimum amount for send operations. Non-blocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed.
Receive Low-water Mark	12	Report the minimum amount for receive operations. Blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount.
Send Timeout	13	Report the timeout value for send operations. It returns a **`EFI_SOCK_TIMEOUT`** structure. If a send operation has blocked for this much time, it returns with a partial count or with the error **`EFI_SOCKERR_WOULDBLOCK`** if no data were sent.
Receive Timeout	14	Report the timeout value for receive operations. It returns an **`EFI_SOCK_TIMEOUT`** structure. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error **`EFI_SOCKERR_WOULDBLOCK`** if no data were received.
Error Status	15	Reports any pending error on the socket and clear the error status.
Protocol Type	16	Reports the *Type* value that was set when the socket was created with **Socket()**.
Non-blocking I/O	17	Report if socket is configured for non-blocking I/O
Reserved	18-255	These values are reserved. All other values are implementation specific.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_BUFFER_TO_SMALL	The *BufferSize* value is too small to complete operation.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_UNSUPPORTED	Option specified in *Option* is not supported for *Level*.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.12 SOCKET.SetSockOpt Function

The **SetSockOpt()** function sets options on a socket.

```
EFI_STATUS
(EFIAPI *EFI_SETSOCKOPT) (
     IN  EFI_SOCKET_INTERFACE   *This,
     IN  SOCKET                 Socket,
     IN  UINT32                 Level,
     IN  UINT32                 Option,
     IN  VOID                   *Buffer,
     IN  UINT32                 BufferSize
     );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
Level	Protocol level at which the option applies.
Option	Identifies the option to set.
Buffer	Pointer to option value.
BufferSize	Size, in bytes, of *Buffer*.

Description

The **SetSockOpt()** function sets the option values for the specified option. Options may exist at multiple protocol levels that can be specified with the *Level* parameter. All implementations must support the “Socket Level” which has a value of 0xffffffff. The options and their values are communications domain and network implementation dependent, however, this following option identifier values are defined to support application compatibility. Unless otherwise indicated, the size of an option value is 4 bytes (UINT32) and is treated as a Boolean value.

Option	**Value**	**Description**
Debug	1	Enable/disable underlying network debugging.
Accept Connection	2	**This is read-only option.**
Reuse Address	3	Enable/disable reuse of local addresses.
Keep Alive	4	Enable/disable periodic messages to be sent on connected socket.
Don’t Route	5	Enable/disable if outgoing messages bypass standard routing.

Option	Value	Description
Broadcast	6	Enable/disable broadcast datagrams on the socket.
Linger	7	Set the state and timeout value for handling unsent messages when **CloseSocket()** is called. The input value is an **EFI_SOCK_LINGER** structure.
Urgent Data Inline	8	Enable/disable urgent data being delivered from the standard receive queue.
Send Buffer Size	9	Set the size of the send buffer for the underlying network implementation.
Receive Buffer Size	10	Set the size of the receive buffer for the underlying network implementation.
Send Low-water Mark	11	Set the minimum amount for send operations. Non-blocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed.
Receive Low-water Mark	12	Set the minimum amount for receive operations. Blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount.
Send Timeout	13	Set the timeout value for send operations. This takes a pointer to an **EFI_SOCK_TIMEOUT** structure. If a send operation has blocked for this much time, it returns with a partial count or with the error **EFI_SOCKERR_WOULDBLOCK** if no data were sent.
Receive Timeout	14	Set the timeout value for receive operations. This takes a pointer to an **EFI_SOCK_TIMEOUT** structure. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error **EFI_SOCKERR_WOULDBLOCK** if no data were received.
Error Status	15	**This is read-only option.**
Protocol Type	16	**This is read-only option.**
Non-blocking I/O	17	Enable/disable socket for non-blocking I/O
Reserved	18-255	These values are reserved. All other values are implementation specific.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_UNSUPPORTED	Option specified in *Option* is unsupported for *Level*.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.13 SOCKET.Shutdown Function

The **Shutdown()** function disables send and/or receive operations on a socket.

```
EFI_STATUS
(EFIAPI *EFI_SHUTDOWN) (
      IN   EFI_SOCKET_INTERFACE   *This,
      IN   SOCKET                 Socket,
      IN   UINT32                 How
      );
```

Parameters

This	A pointer to the **EFI_SOCKET_INTERFACE** instance.
Socket	Socket descriptor associated with this request.
How	Type of shutdown.

Description

The **Shutdown()** function causes all or part of a full-duplex connection on a socket to be shut down. It disables subsequent send and/or receive operations depending on the value of the *How* parameter. The *How* parameter can have the following values:

Shutdown send operations	0
Shutdown receive operations	1
Shutdown both send and receive operations	2

Status Codes Returned

EFI_SUCCESS	The shutdown operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_SOCKERR_NOTCONN	The socket is not connected.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.14 SOCKET.SocketIoctl Function

The **`SocketIoctl()`** manipulates the configuration and/or state of the underlying network implementation.

```
EFI_STATUS
(EFIAPI *EFI_ SOCKETIOCTL) (
     IN  EFI_SOCKET_INTERFACE    *This,
     IN  SOCKET                  Socket,
     IN  UINT32                  Cmd,
     IN OUT  VOID                *Argp
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	Socket descriptor associated with this request.
Cmd	I/O control command.
Argp	Pointer to arguments for control command.

Description

The **SocketIoctl()** function provides the means to affect the network implementation as opposed to an individual socket which is typically done through [**Get**|**Set**]**SockOpt()**. The semantics of this call are network implementation specific.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_UNSUPPORTED	Operation specified in *Cmd* is unsupported.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.15 SOCKET.GetPeerAddr Function

The **`GetPeerAddr()`** function returns address of a connected peer.

```
EFI_STATUS
(EFIAPI *EFI_GETPEERADDR) (
     IN   EFI_SOCKET_INTERFACE   *This,
     IN   SOCKET                 Socket,
     OUT EFI_SOCKADDR            *Addr,
     IN OUT UINT32               *AddrLen
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	Socket descriptor associated with this request.
Addr	Pointer to **`EFI_SOCKADDR`** that receives the address of the remote endpoint.
AddrLen	On input specifies the size, in bytes, of the **`EFI_SOCKADDR`** buffer. On return, specifies the length of relevant data in **`EFI_SOCKADDR`**.

Description

The **GetPeerAddr()** function returns the **EFI_SOCKADDR** of the peer connected to the socket specified by *Socket*.

Status Codes Returned

EFI_SUCCESS	The socket was successfully named.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_SOCKERR_NOTCONN	The specified socket is not connected.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.16 SOCKET.GetSockAddr Function

The **GetSockAddr()** function returns the local address of a socket.

```
EFI_STATUS
(EFIAPI *EFI_GETSOCKADDR) (
     IN   EFI_SOCKET_INTERFACE   *This,
     IN   SOCKET                 Socket,
     OUT EFI_SOCKADDR            *Addr,
     IN OUT UINT32               *AddrLen
     );
```

Parameters

This A pointer to the **EFI_SOCKET_INTERFACE** instance.

Socket Socket descriptor associated with this request.

Addr Pointer to **EFI_SOCKADDR** that receives the local address.

AddrLen On input specifies the size, in bytes, of the **EFI_SOCKADDR** buffer. On return, specifies the length of relevant data in **EFI_SOCKADDR**.

Description

The **GetSockAddr()** function returns the **EFI_SOCKADDR** for the socket specified by *Socket*.

Status Codes Returned

EFI_SUCCESS	The socket was successfully named.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.17 SOCKET.CloseSocket Function

The **`CloseSocket()`** function closes a socket.

```
EFI_STATUS
(EFIAPI *EFI_ CLOSESOCKET) (
      IN  EFI_SOCKET_INTERFACE    *This,
      IN  SOCKET                  Socket,
      );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	Socket descriptor associated with this request.

Description

The **CloseSocket()** function destroys a socket created through the **Socket()** function. All resources associated with the socket are freed. No further socket operations may be performed using the *Socket* parameter.

Status Codes Returned

EFI_SUCCESS	The socket was successfully closed.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

3.1.18 SOCKET.GetLastError Function

The **`GetLastError()`** function retrieves the last implementation dependent error.

```
EFI_STATUS
(EFIAPI *EFI_ GETLASTERROR) (
     IN  EFI_SOCKET_INTERFACE   *This,
     IN  SOCKET                 Socket,
     OUT UINT32                 *LastError,
     );
```

Parameters

This	A pointer to the **`EFI_SOCKET_INTERFACE`** instance.
Socket	Socket descriptor associated with this request.

Description

The **GetLastError()** function returns the last network implementation specific error. This function should be called when **`EFI_SOCKERR_FAILURE`** is returned to determine the exact nature of the error.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	An invalid socket descriptor was specified.
EFI_SOCKERR_FAILURE	An implementation specific error has occurred.

4 TCP/IPv4 Implementation Capabilities

This section outlines the user interface capabilities of the TCP/IPv4 implementation as it relates to the EFI Socket Protocol Interface. It defines the domains, communications semantics, and network protocols supported. It also defines the socket options and send/receive flags supported by the implementation. Finally, it defines the implementation specific behavior primarily through the **SocketIoctl()** function.

4.1 Supported Protocols

The TCP/IPv4 implementation supports the address family in the Internet domain (2). The matrix below defines which communications semantics are available for each supported protocol.

Protocol **Type**	IP 0	ICMP 1	IGMP 2	TCP 6	UDP 17	RSVP 46	RAW 255
1 - Connection-oriented Stream				X			
2 - Datagram					X		
3 - Raw	X	X	X			X	X

Figure 4-1 Internet Domain Socket Type/Protocol Matrix

4.2 Supported Options and Flags

The TCP/IPv4 implementation supports all **[Get|Set]SockOpt()** options and send/receive flags specified in the EFI Socket Protocol Interface. This section details the additional options that are supported. When not defined by the EFI Socket Protocol Interface, all structures and #define values are derived from the appropriate files in FreeBSD /usr/include.

4.2.1 IP/UDP Level Options

The following socket options work on protocol levels **`IPPROTO_IP`** and **`IPPROTO_UDP`**. These operations should be performed on sockets that were created with **`SOCK_DGRAM`** communications semantics.

Option	**Value Type**	**Description**
IP_OPTIONS	**`options buffer*`**	Set/get IP options
IP_TOS	**`int*`**	IP type of service
IP_TTL	**`int*`**	IP time to live
IP_RECVDSTADDR	**`int*`**	Receive IP destination address with datagram
IP_MULTICAST_IF	**`struct in_addr*`**	Set/get IP multicast interface

Option	**Value Type**	**Description**
IP_MULTICAST_TTL	`u_char*`	Set/get IP multicast time to live
IP_MULTICAST_LOOP	`u_char*`	Set/get IP multicast loopback flag
IP_ADD_MEMBERSHIP	`struct ip_mreq *`	Set/get a multicast group membership
IP_DROP_MEMBERSHIP	`struct ip_mreq *`	Drop multicast group membership
IP_PORTRANGE	`int*`	Set/Get range to choose for unspecified port. Value types have the following effect: **`IP_PORTRANGE_HIGH:`** **`IPPORT_HIFIRSTAUTO`** through **`IPPORT_HILASTAUTO`** **`IP_PORTRANGE_LOW:`** **`IPPORT_RESERVED - 1`** through **`IPPORT_RESERVEDSTART`** **`IP_PORTRANGE_DEFAULT:`** **`IPPORT_RESERVED`** through **`IPPORT_USERRESERVED`**
IP_RECVIF	`int*`	Boolean: Receive reception interface with data-gram

4.2.2 TCP Level Options

The following socket options work on protocol level **`IPPROTO_TCP`**. These operations should be performed on sockets that were created with **`SOCK_STREAM`** communications semantics.

Option	**Value Type**	**Description**
TCP_NODELAY	`int*`	Boolean: Don't delay send to coalesce packets
TCP_MAXSEG	`int*`	Set/get maximum segment size
TCP_NOPUSH	`int*`	Boolean: Don't push last block of write
TCP_NOOPT	`int*`	Boolean: Don't use TCP options

4.2.3 Raw Level Options

The following socket options work on protocol level **`IPPROTO_RAW`**. These operations should be performed on sockets that were created with **`SOCK_RAW`** communications semantics.

Option	Value Type	Description
IP_HDRINCL	**`int*`**	Boolean: Users includes IP header data
IP_RSVP_ON	**`void`**	Enable RSVP
IP_RSVP_OFF	**`void`**	Disable RSVP

4.3 SocketIoctl() Operations

This section describes the **SocketIoctl()** operation supported by the TCP/IPv4 implementation. When not defined by the EFI Socket Protocol Interface, all structures and #define values are derived from the appropriate files in FreeBSD /usr/include. Unless otherwise stated, the following commands can be made on any socket.

4.3.1 Socket I/O Control Operations

Cmd	Argp	Description
FIOASYNC	**`EFI_EVENT*`**	This command configures the socket for asynchronous operation. The **`EFI_EVENT`** is signaled whenever the operational state of the socket has changed. The **PollSocket()** function should be used to determine what aspect of socket state has changed. This command is typically used in conjunction with non-block I/O. Passing a NULL *Argp* value will disable asynchronous notification.
FIONBIO	**`UINT32*`**	This command set the non-block I/O configuration. A non-zero value for UINT32 enables non-blocking I/O while a zero value disables non-block I/O.
FIONREAD	**`UINT32*`**	This command will return the number of bytes waiting on the receive queue.
SIOCATMARK	**`UINT32*`**	This command returns Boolean for if the socket has out-of-band data available to read.
SIOCUPCALL	**`struct upcall_req*`**	This command sets a callback address this is called when operational state of the socket has changed. This would happen at the same moment in time that an asynchronous event notification would be made. **The PollSocket()** function should be used to determine what aspect of socket state has changed. This command is typically used in conjunction with non-block I/O. Setting **`upcall`** to NULL will disable callback notifications.

4.3.2 IP Layer I/O Control Operations

Cmd	Argp	Description
SIOCAIFADDR SIOCDIFADDR	**struct ifreq***	These commands add and delete the IP address in **ifr_addr** for the network interface specified in **ifr_name**.
SIOCGIFADDR SIOCSIFADDR	**struct ifreq***	These commands return and set the IP address in **ifr_addr** for the network interface specified in **ifr_name**.
SIOCGIFBRDADDR SIOCSIFBRDADDR	**struct ifreq***	These commands return and set the IP broadcast address in **if_addr** for the network interface specified in **ifr_name**.
SIOCGIFDSTADDR SIOCSIFDSTADDR	**struct ifreq***	These commands return and set the destination IP address in **ifr_addr** for the point-to-point interface specified in **ifr_name**.
SIOCGIFNETMASK SIOCSIFNETMASK	**struct ifreq***	These commands return and set the IP net mask in **ifr_addr** for the network interface specified in **ifr_name**.

4.3.3 Network Interface I/O Control Operations

Cmd	Argp	Description
SIOCADDMULTI SIOCDELMULTI	**struct ifreq***	These commands add and delete a multicast address from the network interface specified in **ifr_name**. The network interface must support multicast address for this command to succeed. The **ifr_addr** field is assumed to be of type **struct sockaddr_dl**.
SIOCGIFCONF	**struct ifconf***	This command returns the address configuration for all network addresses and their associated IP addresses. This list is returned in the buffer pointed to by **ifc_buf** which represents and array of **struct ifreq** entries. Care must be taken to index the ray by taking into account the length of **sockaddr** associated address family for each entry.
SIOCGIFFLAGS SIOCSIFFLAGS	**struct ifreq***	These commands return and set the interface flags in **ifr_flags** for the network interface specified in **ifr_name**.

Cmd	Argp	Description
SIOCGIFMETRIC SIOCSIFMETRIC	**struct ifreq***	These commands return and set the routing metric in **ifr_metric** for the network interface specified in **ifr_name**.
SIOCGIFMTU SIOCSIFMTU	**struct ifreq***	These commands return and set the MTU in **ifr_mtu** for the network interface specified in **ifr_name**. The MTU can not be set to a value larger that can be supported by the network interface.

4.4 Routing Table and ARP Cache Manipulation

The TCP/IPv4 implementation provides an interface for manipulating the internal routing tables which includes the ARP cache. Route table manipulation is a rather pithy subject that is beyond the scope of this document. For that reason, this section provides only a broad overview of the subject. Typically, this interface would be used by a command line utility such as route(8) and arp(8). The reader is referred to the source code of these utilities for a more complete example of manipulating route configuration.

All routing table manipulation must be done on a socket that was created for the route communications domain using the raw socket communications semantic. For example:

```
s = socket(PF_ROUTE, SOCK_RAW, 0);
```

All configuration requests are formed from a **struct rt_msghdr.** The structure has the following format:

```
struct rt_msghdr
     u_short                rtm_msglen;
     u_char                 rtm_version;
     u_char                 rtm_type;
     u_short                rtm_index;
     int                    rtm_flags;
     int                    rtm_addrs;
     pid_t                  rtm_pid;
     int                    rtm_seq;
     int                    rtm_errno;
     int                    rtm_use;
     u_long                 rtm_inits;
     struct rt_metrics      rtm_rmx;
};
```

(This structure, as well as all route configuration structures, can be found in **<net/route.h>**)

The information following the route message header is dependent on the type of configuration operation being performed but are typically one or more socket address types (**sockaddr**, **sockaddr_in**, **sockaddr_dl**, etc.) aligned on 32 bit boundaries. The configuration command types supported are **RTM_ADD**, **RTM_DELETE**, **RTM_CHANGE**, **RTM_GET**, and **RTM_LOCK**.

Configuration information is written to the socket as a single contiguous buffer using **SOCKET.Send** function. The *Addr* and *AddrLen* parameters are ignored on the send request.

The socket may also be read using **SOCKET.Receive** to receive data in a **`rt_msghdr`** format. Receives will complete when the routing subsystem has change the routing configuration due to explicit configuration changes or as a result of messages received from network

5 PPP Implementation

This section describes the implementation details of the PPP network protocol that provides the TCP/IPv4 network stack with connectivity through one or more serial ports.

The logical approach to providing PPP support would be to implement the serial interface and PPP in an EFI protocol that supplied a Simple Network Protocol interface to the TCP/IPv4 network stack. However, the more natural fit into the FreeBSD TCP/IP implementation is to provide a FreeBSD tty line discipline interface to connect the network stack with the appropriate serial device after the PPP connections has been established. A background “daemon”, implemented as an EFI driver, is responsible for managing the serial interface using the EFI Serial Protocol, performs authentication, and negotiates PPP options with the remote end of the connection. Refer to Figure 2-1 Network Stack Architecture for an illustration of the relationship of these components.

5.1 PPP Line Discipline

The TCP/IPv4 EFI protocol contains the code for performing PPP frame encoding and decoding through a tty line switch discipline interface. The PPP line discipline is implemented in the file *ppp_tty.c*. The network stack provides access to the *linesw* structure through the EFI protocol defined as **PPP_PROTOCOL_GUID** in *atk_ppp.h*:

```
#define PPP_PROTOCOL_GUID  \
     { 2bff7bf9-5d04-11d4-87a3-0006292e8a3b }
```

The interface pointer to his protocol is a pointer to the following FreeBSD structure defined in *sys/conf.h*:

```
struct linesw {
     l_open_t   *l_open;
     l_close_t  *l_close;
     l_read_t   *l_read;
     l_write_t  *l_write;
     l_ioctl_t  *l_ioctl;
     l_rint_t   *l_rint;
     l_start_t  *l_start;
     l_modem_t  *l_modem;
     u_char     l_hotchar;
};
```

As a tty line discipline, the PPP code in the network stack was designed to interface to a kernel level tty driver. Therefore, all calls to line switch functions take a pointer to a *tty* structure defined in *sys/tty.h*. The user of the PPP EFI interface (usually the PPP daemon) is responsible for allocating the tty structure and initializing the following fields:

t_ispeed Set to input baud rate. **OPTIONAL**

t_ospeed Set to output baud rate. **OPTIONAL**

t_oproc Set with a pointer to a function responsible for sending output to the serial device. **MANDITORY**

t_pgrp Set (cast) to an EFI event. If non-zero, the PPP frame decoder will signal the specified EFI event when it has placed a decoded PPP frame on the input queue. **OPTIONAL**

t_line Must be set to **PPPDISC**. **MANDITORY**

t_state Must have the **TS_CONNECTED** bit set in order to have the TCP/IP stack process PPP frames. **MANDITORY**

The function prototype for the **t_oproc** field is as follows:

```
void (*t_oproc) (struct tty *);
```

All data to be sent by the output procedure is contained in a FreeBSD *clist* structure placed on the **t_outq** of the *tty* structure. The *clist* structure is defined in *sys/tty.h*. In a FreeBSD implementation, a *clist* entry is processed by a kernel level *getc()* function. Because the TCP/IP stack contains the implementation of the *getc()* and related functions, it is important to have the output procedure use the same routine. This is accomplished by having the PPP line discipline *l_open()* code set a pointer to its internal *getc()* function in the **t_param** field of the *tty* structure. The following code from the PPP Daemon provides an example of how an output function can be written to use the clist/getc interface.

```
/*
 * SerialPortWrite - does the actual write to the serial port
 */
void
SerialPortWrite( struct tty *ptty )
{
    struct clist  *cl = &ptty->t_outq;
    int i, len, chr, count;
    char *p;

    count = cl->c_cc;
    p = malloc(count);

    if (p == NULL)
        return;
    for (i = 0; i < count; i++) {
        /*
         *  Hack to call getc() in tcpip stack.
         */
        chr = ptty->t_param((struct tty*)cl, NULL);
        if ( chr < 0 )
            break;
        p[i] = (unsigned char) chr;
    }

    if (i)
        len = (int)write(ttyfd, p, i);
    free(p);
}
```

Finally, the receive interface into the TCP/IP stack is accomplished through the *l_rint()* line discipline entry. As characters are received, they are passed to this routine along with the associated *tty* structure. When a receive error is detected on the serial device, the special error character **TTY_OE** should be passed through *l_rint()*. The following code from the PPP Daemon provides an example of using the *l_rint()* interface.

```
ReadIntr( void )
{
    int len;
    u_char p = ReadBuf; /* implemented here as a global */

    if (ttyfd == -1)
        return;

    if ((len = (int)read(ttyfd, p, sizeof(ReadBuf))) < 0) {
        if (errno != EWOULDBLOCK && errno != EINTR) {
            pppdisc->l_rint(TTY_OE, tp);
        }
    } else {
        /*
         *  Feed serial data to ppp stack for packetization
         */
        while(len--) {
            (void)pppdisc->l_rint(*p++, tp);
        }
    }
}
```

5.2 PPP Daemon (pppd)

The PPP Daemon (pppd) is a port of the FreeBSD implementation. It is responsible for setting up the serial port, initializing and answering the a modem, negating PPP options with the remote end, authenticating the remote end, configuring the network interface, and interfacing the serial port to the TCP/IPv4 network stack. The PPP code in the TCP/IPv4 stack is responsible for frame encode/decode and passing IP frames on through the stack and socket interface.

The pppd receives configuration options (serial port, line speed, IP address, etc.) through the LoadOptions field of its Image Protocol and/or text files. It is implemented as an EFI driver to allow it to run in the background to send and receive serial data on behalf of the TCP/IPv4 network stack. It does not export a EFI protocol interface.

The supported authentication protocols are none, PAP and CHAP. The export restricted libraries libmd.lib and libcyrpt.lib from the EFI Application Toolkit provide support for the PAP and CHAP authentication.

Pppd does not provide a true modem driver. It does allow one or more user specified modem strings to be sent to the modem and will send the ATA answer sequence when the Ring Indicate modem control line is asserted. It also supports direct connect communications.

The user can specify whether pppd should return to the parent program as soon as it is initialized or whether it should wait until a connection attempt has completed; returning a status value that indicates success or the exact nature of the connection failure.

Refer to the manual page for pppd for complete configuration option description.