zilog
Embedded in Life
An IXYS Company

eZ80® Family of Microprocessors

Zilog TCP/IP Stack API

Reference Manual

RM004016-1012

www.zilog.com

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

eZ80 and eZ80Acclaim! are registered trademarks of Zilog Inc. All other product or service names are the property of their respective owners.

Revision History

Each instance in the Revision History table below reflects a change to this document from its previous version. For more details, click the appropriate links in the table.

Date	Revision Level	Description	Page
Oct 2012	16	Corrected erroneous document control number from RM0041 to RM0040.	All
Dec 2011	15	Globally updated for the ZTP v2.4.0 release.	All
Aug 2010	14	Globally updated for the ZTP v2.3.0 release.	All
Nov 2008	13	Updated for the ZTP v2.2.0 release; updated recv, send, recvfrom, sendto, SNMP Functions, ztpSnmpV1Init, ztpSnmpV2Init, ztpSnmpV3Init, snmpGenerateTrap, SNMP Data Types, SNMP Data Structures, HTTP Data Structures, OID Structure, SNMPObjValue Structure, SNMPObjLs Structure, Table 15, Table 16, Table 17, Table 18. Added SNMP Enumerations, PPP Functions, SNMPObj Structure, and updated xc_ascdate sections. Removed udp_socket structure and tcp_socket structure sections.	14, 16, 20, 22, 45, 46, 47, 49, 50, 82, 86, 87, 89, 90, 91, 95, 96, 98
Jul 2007	12	Globally updated for branding.	All
Jul 2007	11	Globally updated for the ZTP v2.1.0 release.	All

Date	Revision Level	Description	Page
Jun 2007	10	Updated for style; updated ioctlsocket, ftp_connect, do_programatic_login, do_a_ftp_command, Http_Request Structure, accept, listen, hgleave, name2ip, xc_ascdate, Table 16, Table 17. Removed Kernel APIs, Process Manipulation Functions, Semaphore Functions, Mailbox Messaging Functions, Message Port Functions, Miscellaneous Operating System Functions, Kernel Macros, Sample usage in.C and.asm Files sections. Removed appendices.	All
Jul 2006	09	Globally updated for the ZTP v2.0.0 release.	All

Table of Contents

List of Tables

Introduction

This reference manual describes the APIs associated with Zilog's TCP/IP (ZTP) Stack for Zilog's eZ80 CPU-based microprocessors and microcontrollers. This ZTP release supports the eZ80Acclaim!™ family of devices, which includes the eZ80F91, eZ80F92 and eZ80F93 microcontrollers, and the eZ80® family of devices, which includes the eZ80L92 microprocessor.

About This Manual

Zilog has developed this manual to be used as a reference guide for ZTP APIs. Zilog recommends that you read and understand everything in this manual before developing with the Zilog TCP/IP stack.

Intended Audience

This document is written for Zilog customers who are familiar with real-time operating systems and are experienced with microprocessors, writing assembly code, or writing higher-level languages such as C.

Manual Organization

This reference manual presents a functional reference to the following APIs.

ZTP Networking APIs

HTTP Functions

HTTPS Functions

Related Documents

Table 1 lists a number of documents that support the eZ80 and eZ80Acclaim! families. To use ZTP efficiently, Zilog recommends that you become familiar with them.

Table 1. Related Documents

Document Title	Document Number
eZ80L92 Product Specification	PS0130
eZ80F91 MCU Product Specification	PS0192
eZ80F92/eZ80F93 Flash MCU Product Specification	PS0153
eZ80F92/eZ80F93 Ethernet Module Product Specification	PS0186
eZ80F92/eZ80F93 Flash Module Product Specification	PS0189
eZ80 CPU User Manual	UM0077
Zilog Real-Time Kernel Reference Manual	RM0006

Manual Conventions

The following convention is adopted to provide clarity and ease of use:

Courier New Typeface

Code lines and fragments, functions, and executable items are distinguished from general text by appearing in the Courier New typeface. For example, `#include <socket.h>`.

Software Release Versions

Software release versions in this manual are represented as `<version>`, which denotes the current release of the ZTP software available on www.zilog.com. Version numbers are expressed as `X.Y.Z`, in which `X` is the major release number; `Y` is the minor release number, and `Z` is the revision number.

Safeguards

When you use ZTP with one of Zilog's eZ80 development platforms, always use a grounding strap to prevent damage resulting from electrostatic discharge (ESD) to avoid permanent damage to the development platform.

It is important that you understand the following safety terms.

⚠ **Caution:** A procedure or file can be corrupted if you do not follow directions.

⚠ **Warning:** A procedure can cause injury or death if you do not follow directions.

Online Information

Visit Zilog's eZ80 and eZ80Acclaim! web pages for:

- Product information for eZ80 and eZ80Acclaim! devices
- Downloadable documentation describing the eZ80 and eZ80Acclaim! devices
- Source license information

ZTP API Reference

The Zilog TCP/IP Stack (ZTP) consists of a rich set of APIs for accessing the TCP/IP protocol stack. This section provides a description of each ZTP API, including inputs and outputs. Each API is classified according to the protocol or command that it is associated with.

Table 2 provides a quick reference to these ZTP APIs based on their protocols.

Table 2. ZTP API Quick Reference

ZTP Networking APIs
HTTP Functions
HTTPS Functions
SNMP Functions
SMTP Function
Telnet Functions
TimeP Protocol Function
DNS Functions
RARP Function
IGMP Functions
TFTP Functions
FTP Functions
Ping Function
SNTP Functions
PPP Functions

ZTP Networking APIs

This section describes the user interfaces to the ZTP stack. All of the APIs listed in this section return a negative value if an error occurs. Positive values are considered to be the expected output.

Table 3 provides a quick reference to ZTP Networking APIs.

Table 3. ZTP Networking APIs Quick Reference

socket	recvfrom
bind	sendto
accept	ioctlsocket
listen	getsockname
connect	getpeername
recv	inet_addr
send	inet_ntoa
close_s	

SOCKET

Include

```
#include <socket.h>
```

Prototype

```
INT16 socket (
 INT16 af,
 INT16 type,
 INT16 protocol
);
```

Description

The socket function creates a socket that is bound to a specific service provider.

Argument(s)

af	An address family specification. ZTP supports only the AF_INET Internet address family.

`type`	A type specification for the new socket. ZTP supports the following two types of sockets: SOCK_STREAM: Provides sequenced, reliable, two-way, connection-based byte streams with an out-of-band data transmission mechanism. Uses TCP for the Internet address family. SOCK_DGRAM: Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses UDP for the Internet address family. Socket type definitions appear in the socket.h header file.
`protocol`	The protocol function is a particular protocol to be used with sockets that are specific to an indicated address family. As this parameter is not used, the value passed must be zero across all versions of ZTP.

The `socket` function causes a socket descriptor and any related resources to be allocated and bound to a specific transport service provider.

Return Value(s)

If successful, the `socket` function returns the socket descriptor, the value of which must be greater than or equal to 0.

If the returned value is less than 0, then one of the following errors is returned.

EPROTONOSUPPORT	Protocol not supported.
ENOBUFS	Buffer not available.

BIND

Include

```
#include <socket.h>
```

Prototype

```
INT16 bind (
 INT16 s,
 struct sockaddr * name,
 INT16 namelen
);
```

Description

The sockets’ `bind` function associates a local address with a socket.

Argument(s)

`s`	A descriptor identifying an unbound socket.
`name`	The address to assigned to the socket from the sockaddr structure.
`namelen`	The length of the `name` parameter.

Note: The `bind` function is used on an unconnected socket before subsequent calls to the connect and listen functions. It is used to bind either connection-oriented (stream) or connectionless (datagram) sockets. Use the `bind` function to establish a local association of the socket by assigning a local name to an unnamed socket.

ReturnValue(s)

If successful, the `bind` function returns `ZTP_SOCK_OK`.

If less than 0, one of the following errors is returned.

EFAULT	Address family not supported.
EINVAL	Invalid socket descriptor (descriptor already in use).
EBADF	Invalid socket descriptor (not allocated).

See Also

The sockaddr data structure is used in conjunction with networking APIs.

ACCEPT

Include

```
#include <socket.h>
```

Prototype

```
INT16 accept
(
INT16 s,
struct sockaddr *peername,
INT16 *peernamelen
);
```

Description

The sockets’ `accept` function accepts an incoming connection attempt on a socket.

Argument(s)

`s`	A descriptor identifying a socket that has been placed in a listening state with the listen function. The connection is made with the socket that is returned by accept.
`peername`	An optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the peername parameter is determined by the address family established when the socket connection was created.
`peernamelen`	An optional pointer to an integer that contains the length of the `peernamelen`.

Notes:

1. The `accept` function extracts the first connection on the queue of pending connections on socket `s`. It then creates a new socket and returns a handle to the new socket. This newly-created socket is the socket that handles the actual connection. The accept function can block the caller until a connection is present if no pending connections are present in the queue, and the socket is marked as blocking. After successful completion, `accept` returns a new socket handle. The original socket remains open and listens for new connection requests.
2. The `addr` parameter is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. `addrlen` is a value-result parameter that should initially contain the amount of space pointed to by `addr`; upon return, it contains the actual length (in bytes) of the returned address.
3. The `accept` function is used with connection-oriented socket types such as SOCK_STREAM.

Return Value(s)

Success	If no error occurs, `accept` returns a value of type INT16 that is a descriptor for the new socket. The integer referred to by `addrlen` initially contains the amount of space pointed to by `addr`. Upon return, it contains the actual length, in bytes, of the address returned.

Failure One of the following error codes is returned:

`EOPNOTSUPP`: Socket type not supported.

`EBADF`: Invalid socket descriptor.

`EINVL`: Invalid socket descriptor.

`ENOCON`: Connection not arrived.

`EFAULT`: Error accepting new socket.

See Also

The sockaddr data structure is used in conjunction with networking APIs.

LISTEN

Include

```
#include <socket.h>
```

Prototype

```
INT16 listen (
INT16 s,
INT16 backlog
);
```

Description

The sockets’ `listen` function places a socket into a state within which it listens for an incoming connection.

Argument(s)

`s`	A descriptor identifying a bound, unconnected socket.
`backlog`	The maximum length of the queue of pending connections. If this value is `MAXSOCKS`, then the underlying service provider responsible for socket `s` sets the backlog to a maximum reasonable value.

Notes:

1. The socket s is placed into passive mode in which incoming connection requests are acknowledged and queued pending acceptance by the process.
2. Servers that can facilitate more than one connection request at a time use the `listen` function.

Return Value(s)

Success If no error occurs, `listen` returns a 0.

Failure One of the following values is returned:

`EINVAL`: Invalid socket descriptor.

`EBADF`: Invalid socket descriptor (not allocated).

`EOPNOTSUPP`: Socket type not supported.

`EFAULT`: backlog exceeding `MAXSOCKS`.

CONNECT

Include

```
#include <socket.h>
```

Prototype

```
INT16 connect
(
 INT16 s,
 struct sockaddr *peername,
 INT16 peernamelen
);
```

Description

The sockets' `connect` function establishes a connection to a specified socket.

Argument(s)

`s`	A descriptor identifying an unconnected socket.
`peername`	A pointer to the socket structure specifying the host to connect to.
`peernamelen`	The size of the `peername` parameter structure.

Notes: 1. The `connect` function is used to create a connection to a specified destination. If the socket s is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

2. By default, `connect` is a blocking call and is not returned unless a connection is established or is refused.

ReturnValue(s)

Success	If no error occurs, `connect` returns `ZTP_SOCK_OK`.
Failure	One of the following errors is returned:
	`EAFNOSUPPORT`: Address family not supported.
	`EINVAL`: Invalid descriptor.
	`ECONNREFUSED`: Connection refused by peer.

See Also

The sockaddr data structure is used in conjunction with networking APIs.

RECV

Include

```
#include <socket.h>
```

Prototype

```
INT16 recv
(
 INT16 s,
 INT8 * buf,
 INT16 nbyte,
 INT16 flags
);
```

Description

The sockets' `recv` function receives data from a connected socket.

Argument(s)

`s`	A descriptor identifying a connected socket.
`buf`	A pointer to a buffer for the incoming data.
`nbyte`	The length of `buf`.
`flags`	Reserved for future use.

Notes: 1. The `recv` function reads incoming data on connection-oriented sockets. The sockets must be connected before calling `recv`. For a connected socket, the `recv` function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are silently discarded.

2. For connection-oriented sockets (type `SOCK_STREAM` for example), calling `recv` returns as much information as is currently available (up to the size of the buffer supplied).
3. Zilog recommends not using `recv()` with datagram sockets.

ReturnValue(s)

Success	If no error occurs, `recv()` returns the number of bytes received. If the connection has been gracefully closed, the return value is `EFAULT`.
Failure	One of the following error codes is returned: `EDEADSOCK`: Socket is closed. `EBADF`: Invalid descriptor. `EPIPE`: Invalid socket type. `ZTP_ALREADY_BLOCKED` (–18): One thread is already blocked.
ZTP_SOCK_ERR	Indicates a system time out while receiving data.

SEND

Include

```
#include <socket.h>
```

Prototype

```
INT16 send
(
 INT16 s,
 INT8 *buf,
 INT16 nbyte,
 INT16 flags
);
```

Description

This sockets' `send` function sends data on a connected socket.

Argument(s)

`s`	A descriptor identifying a connected socket.
`buf`	A buffer containing the data to be transmitted.
`nbyte`	The length of the data in `buf`.
`flags`	An indicator specifying the method in which a call is made. If used, `tcp_FlagPUSH` – the appropriate outbound TCP segment – contains a `PSH` flagset in code bits.

Notes: 1. The send function is used to write outgoing data on a connected socket. The successful completion of a send does not indicate that the data was successfully delivered.

2. If no buffer space is available within the transport system to contain the data to be transmitted, `send` blocks unless the socket is placed in a nonblocking mode.
3. On nonblocking stream-oriented sockets, the number of bytes written is between one and the requested length, depending on buffer availability on both client and server.

Return Value(s)

Success	If no error occurs, `send` returns the total number of bytes sent, which can be less than the number indicated by `len` for nonblocking sockets.
Failure	One of the following errors is returned: `EDEADSOCK`: The socket is closed. `EBADF`: Invalid descriptor. `EPIPE`: Invalid socket type. `ZTP_ALREADY_BLOCKED` (–18): One thread is already blocked.
ZTP_SOCK_ERR	Indicates a system time out while sending data.

See Also

ZTP Core Macros

CLOSE_S

Include

```
#include <socket.h>
```

Prototype

```
INT16 close_s (INT16 s);
```

Description

The sockets' `close_s` function closes an existing socket.

Argument(s)

`s`	A descriptor identifying a socket to close.

Notes:

1. The `close_s` function closes an active socket. This function is used to release the socket descriptor `s` so that further references to `s` fail. Any pending asynchronous or blocking calls issued by any thread in this process are cancelled without displaying any notification messages. To return any socket resources to the system, an application must contain a matching call to `close_s` for each successful call to the socket.
2. If `close_s` is issued on a master socket (a socket used in a TCP server application and passed to the accept call as a parameter), all listening sockets on the same port are closed to accept those sockets that are already in the established state.

Return Value(s)

Success	`ZTP_SOCK_OK.`
Failure	`EBADF`: Invalid socket descriptor (not allocated).

RECVFROM

Include

```
#include <socket.h>
```

Prototype

```
INT16 recvfrom
(
 INT16 s,
 INT8 *buf,
 INT16 len,
 INT16 flags,
 struct sockaddr * from,
 INT16 * fromlen
);
```

Description

The sockets' `recvfrom` function receives a datagram and stores the source address.

Argument(s)

`s`	A descriptor identifying a bound socket.
`buf`	A buffer for incoming data.
`len`	The length of `buf`.
`flags`	An indicator specifying the way in which the call is made. As this parameter is not used, the value passed must be zero across all versions of ZTP.
`from`	An optional pointer to a buffer that will hold the source address upon return.
`fromlen`	An optional pointer to the size of the `from` buffer.

Note: The `recvfrom` function reads incoming data on unconnected sockets and captures the address from which the data is sent; the local address of the socket must be known. For server applications, this determination is usually made explicitly via the `bind` function. Explicit binding is discouraged for client applications. `recvfrom` must be used only with datagram sockets.

Return Value(s)

Success	If no error occurs, `recvfrom` returns the number of bytes received.
Failure	If an error occurs, one of the following error codes is returned: `EBADF`: Invalid descriptor. `EPIPE`: Invalid socket type. `ENOCON`: Connection refused. `EFAULT`: Another thread is already blocked on the socket.
ZTP_SOCK_ERR	Indicates a system time out while receiving data.

See Also

The sockaddr data structure is used in conjunction with networking APIs.

SENDTO

Include

```
#include <socket.h>
```

Prototype

```
INT16 sendto
(
 INT16 s,
 INT8 *buf,
 INT16 len,
 INT16 flags,
 struct sockaddr *to,
 INT16 tolen,
);
```

Description

The sockets’ `sendto` function sends data to a specific destination.

Argument(s)

`s`	A descriptor identifying a datagram socket.
`buf`	A buffer containing the data to be transmitted.
`len`	The length of the data in `buf`.
`flags`	An indicator specifying the way in which the call is made. As this parameter is not used, the value passed must be zero.
`to`	An optional pointer to the address of the target socket.
`tolen`	The size of the address specified in `to`.

Notes: 1. The `sendto` function is used to write outgoing data on a socket. For message-oriented sockets, the `to` parameter can be any valid address in the socket’s address family, including a broadcast address or any multicast address.

2. If the socket is unbound, unique values are assigned to the local association by the system, and the socket is then marked as bound.
3. The successful completion of a `sendto` does not indicate that the data was successfully delivered. `sendto` must be used only with connectionless datagram sockets.

Return Value(s)

Success	If no error occurs, `sendto` returns the total number of bytes sent, which can be less than the number indicated by `len`.
Failure	If an error occurs, one of the following error codes is returned: `EBADF`: Invalid descriptor. `EPIPE`: Invalid socket type. `ENOCON`: Connection refused.
ZTP_SOCK_ERR	Indicates a system time out while sending data.

See Also

The sockaddr data structure is used in conjunction with networking APIs.

IOCTLSOCKET

Include

```
#include <socket.h>
```

Prototype

```
INT16 ioctlsocket
(
 INT16 s,
 INT32 cmd,
 UINT32 *argp
);
```

Description

The sockets' `ioctlsocket` function controls the I/O mode of a socket.

Argument(s)

`s`	A descriptor identifying a socket.
`cmd`	One of the following supported commands to perform on socket `s`.

`UDPTIMEOUT`: Sets up finite time blocking for a UDP socket. The `argp` parameter specifies the value of `timeout` in seconds.

`TCPTIMEOUT`: Sets up finite time blocking for a TCP socket. The `argp` parameter specifies the value of `timeout` in seconds.

`FIONBIO`: Use with a NULL `argp` parameter to enable the nonblocking mode of socket `s`. The `argp` parameter points to a `UINT32` value. When a socket is created, it operates in blocking mode by default (nonblocking mode is disabled). This operation is consistent with BSD sockets.

`cmd` (cont'd)

`FCNCLBIO`: This command resumes any thread blocked on the socket for `recv()`/`send()`/`connect()`/`accept()`. The `argp` parameter points to a UINT32 value. If the thread is to be unblocked from `recv()`, `*argp` must be 1; otherwise, it must be 6 if the thread must be unblocked from `send()`, `connect()` or `accept()` calls.

`FUDPCKSUM`: This command disables UDP checksum calculation, which is enabled by default.

`FDISNAGLE`: This command disables the nagle algorithm which is enabled by default (used only for TCP sockets).

`FENANAGLE`: This command enables the nagle algorithm if disabled using `FDISNAGLE` (used only for TCP sockets).

`FIONREAD`: This command determines the amount of data pending in the network's input buffer that can be read from socket s (used for TCP/UDP sockets).

`FIONWRITE`: This command determines the amount of data pending in the network's output buffer that is yet to be sent out by the network stack (used only for TCP sockets).

`TCPKEEPALIVE_ON`: This command enables the Keep Alive feature of the TCP protocol. The argp parameter specifies the value of Keep Alive timeout in seconds.

`TCPKEEPALIVE_OFF`: This command disables the Keep Alive feature of the TCP protocol.

`argp` A pointer to a parameter for `cmd`.

Notes: 1. The `ioctlsocket` function can be used on any socket in any state. It is used to set or retrieve operating argument(s) associated with the socket.

2. Compatibility: The ioctlsocket function performs only a subset of functions on a socket when compared to the ioctl function found in Berkeley sockets.

Return Value(s)

Success	Returns 0 if successful. If cmd is FIONREAD, the number of bytes of data present in the socket buffer to be read is returned. If cmd is FIONWRITE, the number of bytes of data present in the socket buffer to be sent is returned.
Failure	One of the following error codes is returned: EFETNOSUPPORT: If requested command is not implemented. EBADF: Invalid descriptor. If cmd is FIONREAD/FIONWRITE, the return value is the amount of data pending in the network's input/output buffer that can be read/sent from socket s.

GETSOCKNAME

Include

```
#include <socket.h>
```

Prototype

```
INT getsockname
(
 INT16 s,
 struct sockaddr * name,
 INT * namelen
);
```

Description

The sockets’ `getsockname` function retrieves the local name for a socket.

Argument(s)

`s`	A descriptor identifying a bound socket.
`name`	Receives the address (name) of the socket.
`namelen`	The size of the `name` buffer.

Notes: 1. The `getsockname` function retrieves the current name for the socket descriptor specified by `s`. It is used on the bound or connected socket specified by the `s` parameter. The local association is returned. This call is especially useful when a `connect` call has been made without performing a bind first; the `getsockname` function determines the local association.

2. The `getsockname` function always does not return information about the host address when the socket has been bound to an unspeci-

fied address, unless the socket has been connected with connect or accept (for example, using ADDR_ANY).

Return Value(s)

If no error occurs, getsockname returns 0; otherwise, it returns –1.

When called, the namelen argument contains the size of the name buffer, in bytes. Upon return, the namelen parameter contains the actual size (in bytes) of the name parameter.

See Also

The sockaddr data structure is used in conjunction with networking APIs.

GETPEERNAME

Include

```
#include <socket.h>
```

Prototype

```
int getpeername
(
 short s,
 struct sockaddr * name,
 int * namelen
);
```

Description

The sockets’ `getpeername` function retrieves the name of the peer to which a socket is connected.

Argument(s)

`s`	A descriptor identifying a connected socket.
`name`	The structure that receives the name of the peer.
`namelen`	A pointer to the size of the `name` structure.

Note: The `getpeername` function retrieves the name of the peer connected to the socket `s` and stores it in the `sockaddr` structure identified by `name`. The `getpeername` function can be used only on a connected socket. For datagram sockets, only the name of a peer specified in a previous `connect` call is returned; any name specified by a previous `sendto` call is returned by `getpeername`.

Return Value(s)

If no error occurs, `getpeername` returns 0; otherwise, it returns -1.

When called, the `namelen` argument contains the size of the `name` buffer, in bytes. Upon return, the `namelen` parameter contains the actual size in bytes of the `name` returned.

See Also

The sockaddr data structure is used in conjunction with networking APIs.

INET_ADDR

Include

```
#include <ZTPtcp.h>
```

Prototype

```
UINT32 inet_addr
(
 INT8 *charp
);
```

Description

The sockets' `inet_addr` function converts a string containing an Internet Protocol (IPv4) dotted address into a UINT32 value.

Argument(s)

`charp`	A null-terminated character string representing a number expressed in the Internet standard dotted (.) notation.

Note: The `inet_addr` function interprets the character string specified by the `charp` parameter. This string represents a numeric Internet address expressed in the Internet standard dotted notation. The value returned is a number used as an Internet address. All Internet addresses are returned in the host byte order (*little endian*, in the case of eZ80 devices).

Return Value(s)

If no error occurs, `inet_addr` returns a UINT32 value containing a suitable binary representation of the Internet address given; otherwise, it returns 0.

INET_NTOA

Include

```
#include <ZTPtcp.h>
```

Prototype

```
INT8 *inet_ntoa(INT8 *s, UITN32 x)
```

Description

The `inet_ntoa` function converts an IPv4 network address into a string in Internet standard dotted format.

Argument(s)

`s`	A pointer to a memory buffer to hold a dotted-notation (`a.b.c.d`) IP address.
`x`	Unsigned long representation of an IP address.

Note: The `inet_ntoa` function takes a UINT32 parameter as an IP address and returns an ASCII string representing the address in dotted (.) notation, as in `a.b.c.d`.

Return Value(s)

If no error occurs, `inet_ntoa` returns a `INT8` pointer to a static buffer containing the text address in standard dotted (.) notation. Otherwise, it returns `NULL`.

HTTP Functions

The Zilog TCP/IP Stack supports the following three HTTP functions:

- http_init
- httpBasicAuth_init
- httpDigestAuth_init

HTTP_INIT

Include

```
#include <http.h>
```

Prototype

```
INT16 http_init (const Http_Method*
http_defmethods,const struct header_rec *
httpdefheaders, Webpage *website, UINT16 portnum);
```

Description

The `http_init` function initializes a webserver (or website), makes a TCP connection on a specified port, and waits for a client request. Upon receiving a request from the client, the webserver provides a response according to the webserver configuration.

Argument(s)

`http_defmethods`	A pointer to the supported methods structure.
`httpdef headers`	A pointer to the suppoted header structures.
`website`	A pointer to the website for which the server processes requests.
`portnum`	Port number on which the HTTP server listens.

Return Value(s)

If no error occurs, it returns the `http server` port number. Otherwise, it returns `SYSERR`.

HTTPBASICAUTH_INIT

Include

```
#include <http.h>
```

Prototype

```
INT16 httpBasicAuth_init
(const Http_Method * http_defmethods,
const struct header_rec * httpdefheaders,
Webpage *website, UINT16 portnum);
```

Description

The `httpBasicAuth_init` function initializes a webserver (or website) with Basic Authentication support, opens a TCP connection on a specified port, and waits for a client request. Upon receiving a request from the client, the webserver requests for authentication by asking for user name and password, which will be verified against the configured values. If the user name and password are correct then it responds according to the webserver configuration.

Argument(s)

`http_defmethods`	A pointer to the supported methods structure.
`httpdef headers`	A pointer to the suppoted header structures.
`website`	A pointer to the website for which the server processes requests.
`portnum`	Port number on which the HTTP server listens.

Return Value(s)

If no error occurs, it returns the `http server` port number. Otherwise, it returns SYSERR.

HTTPDIGESTAUTH_INIT

Include

```
#include <http.h>
```

Prototype

```
INT16 httpDigestAuth _init (const Http_Method*
httpAuth_defmethods,const struct header_rec *
httpdefheaders, Webpage *website, UINT16 portnum);
```

Description

The `httpDigestAuth _init` function initializes a webserver (or website) with MD5 Digest Authentication support, opens a TCP connection on a specified port, and waits for a client request. Upon receiving a request from the client, the webserver requests for authentication by asking for user name and password, which will be verified against the configured values. If the user name and password are verified correct then it provides a response according to the webserver configuration.

Argument(s)

`http_defmethods`	A pointer to the supported methods structure.
`httpdef headers`	A pointer to the suppoted header structures.
`website`	A pointer to the website for which the server processes requests.
`portnum`	Port number on which the HTTP server listens.

Return Value(s)

If no error occurs, it returns the `http server` port number. On failure, it returns SYSERR.

HTTP Supported Method: http_defmethods

```
const Http_Method http_defmethods[] = {
{ HTTP_GET, "GET", http_get },
{ HTTP_HEAD, "HEAD", http_get },
{ HTTP_POST, "POST", http_post },
{ HTTP_SUBSCRIBE, "SUBSCRIBE", http_post },
{ HTTP_UNSUBSCRIBE, "UNSUBSCRIBE", http_post },
{ 0, NULL, NULL },
};
const Http_Method httpAuth_defmethods[] = {
{ HTTP_GET, "GET", httpAuth_get },
{ HTTP_HEAD, "HEAD", httpAuth_get },
{ HTTP_POST, "POST", http_post },
{ HTTP_SUBSCRIBE, "SUBSCRIBE", http_post },
{ HTTP_UNSUBSCRIBE, "UNSUBSCRIBE", http_post },
{ 0, NULL, NULL },
};
```

The HTTP server calls the corresponding get function, based on which HTTP is initialized whenever it encounters an HTTP_GET request. The default method handlers can be overridden by replacing these defaults with another declaration of this structure.

Notes:

1. The default handlers provided with ZTP are sufficient to handle these HTTP methods; it is not necessary to override them. Do not override the default methods unless you are familiar with the HTTP protocol.
2. The http_defmethods array is extensible. Additional methods can be added to the list of standard HTTP methods by modifying the http_defmethods[] structure. These methods can be optional HTTP 1.1 methods such as Put, Delete, Trace, or custom methods such as My_Method.
3. When implementing a nonstandard method, it is unlikely that a standard web browser can invoke a custom method. Describing the operation of the HTTP protocol is beyond the scope of this manual.

4. All method handlers follow the same function prototype, as defined in the `http.h` file.

Example

The method handler simply parses the `http_request` and performs the appropriate action(s), as shown in the following example.

```
void method_handler( Http_Request * )
{
//Program coded by you
}
```

HTTP Supported Header: httpdefheaders

This array of `header_rec` structures constitutes the list of HTTP headers recognized by the webserver. The default list of recognized headers is shown in the following code:

```
const struct header_rec httpdefheaders[] = {
{ "Accept",               HTTP_HDR_ACCEPT },
{ "Cache-Control",        HTTP_HDR_CACHE_CONTROL },
{ "Callback",             HTTP_HDR_CALLBACK },
{ "Connection",           HTTP_HDR_CONNECTION },
{ "Content-Length",       HTTP_HDR_CONTENT_LENGTH },
{ "Content-Type",         HTTP_HDR_CONTENT_TYPE },
{ "Transfer-Encoding",    HTTP_HDR_TRANSFER_ENCODING },
{ "Date",                 HTTP_HDR_DATE },
{ "Location",             HTTP_HDR_LOCATION },
{ "Host",                 HTTP_HDR_HOST },
{ "Server",               HTTP_HDR_SERVER },
{ "Authorization",        HTTP_HDR_SEND_CLIENT_AUTH },
{ "WWW-Authenticate",     HTTP_HDR_ASK_CLIENT_AUTH},
{"Authentication-Info",   HTTP_HDR_SEND_SERVER_AUTH},
{ NULL,0 },
} ;
```

Before calling a method handler, the HTTP server parses incoming HTTP requests into an `http_request` structure, and passes this structure as a parameter to the handler. This http_request structure is listed in Appendix A. Definitions and Codes on page 85.

The HTTP server creates an entry in the rqstheaders field of the http_request structure for known headers from the httpdefheaders structure. Therefore, if the application requires additional headers that are not in the default httpdefheaders structure, you must provide the httpdefheaders structure before calling http_init.

HTTP Supported Header: website

A pointer to the website for which the server processes requests. The website parameter can contain both static web pages and dynamic web pages. Each element of the website array corresponds to a single static or dynamic web page. Two sample web page declarations for a static web-page and the dynamic page are described below:

```
Webpage website[] = {
{HTTP_PAGE_STATIC, "/", "text/html",
&my_static_page_htm},
{HTTP_PAGE_DYNAMIC, "/dynamic.htm", "text/html"},
```

Static Web Pages

If a website consists of only static webpages, the default HTTP library contains all of the necessary routines to process Get and Head requests without providing any additional code. The HTTP server calls its internal http_get method-handling function when a Get or Head request is received for any static webpage within the website array. The ZTP internal http_get method then returns the appropriate object in an HTTP response. However, if the website contains dynamic web pages, you must provide the code to complete the processing of the HTTP request.

Dynamic Web Pages

When the ZTP HTTP server encounters a request for a dynamic page, it parses the incoming request into an `http_request` structure, then calls a helper function to complete the request. For example, see the dynamic page entry in the *website* definition provided above. When processing a Get request on the `dynamic.htm` page, the HTTP server's `http_get` function calls the `MY_DYNAMIC_CGI` helper function to generate the HTTP response for return to the client. A pointer to the `http_request` structure is passed to the helper function, `my_dynamic_cgi`.

Additional HTTP APIs

The following function adds the specified `{header, value}` pair to the list of response headers that is sent back to the HTTP request.

```
void http_add_header (Http_Request *request, UINT16
header, INT8 *value)
```

The following routine searches through the list of argument(s) associated with the given HTTP request.

```
INT8 *http_find_argument (Http_request *request, UINT8
*key)
```

The following function adds the specified {header, value} pair to the list of response headers that is sent back to the HTTP request.

```
void http_add_header (Http_Request *request, UINT16
header, INT8 *value)
```

The following routine searches through the list of argument(s) associated with the given HTTP request for a parameter, the name of which matches the passed key. If such a parameter is found within the parsed request structure, a pointer to its value is returned.

```
INT8 *http_find_argument (Http_request *request, UINT8
*key)
```

The following routine searches through the list of `rqstheaders` in the `http_request` structure for a header, the name of which matches the specified key. If successful, a pointer to the value of the header is returned.

```
INT8 *http_find_header (Http_Request *rqst, UINT8 key)
```

The following routine parses the given HTTP parameter structure for a parameter, the name of which matches the specified key. If such a parameter is found within the passed list, the function returns a pointer to the parameter's value.

```
INT8 *http_find_param (Http_Params *params, UINT8
*key)
```

The following routine outputs the text representation of all of the instances of `httpdefheader` contained in the `resp-headers` array, along with its corresponding values.

```
void http_output_headers (Http_Request *request);
```

For more information, refer to the `website` demo provided with the standard projects, which is available in the following path:

```
<ZTPInstall>ZTP\SamplePrograms\ZTPDemo
```

Example 1

If the CGI routine calls the function `add_header(request, HTTP_HDR_LOCATION, Jupiter)` then calls `output_headers(request)`, the following text is added to the HTTP response:

```
Location: Jupiter\r\n INT16 http_output_reply
(Http_request *request, UINT16 reply)
```

The following function transmits the HTTP status line and response headers contained in the associated HTTP request structure. The status line is constructed from the passed reply code.

Example 2

A reply code of `HTTP_200_OK` results in the following status line being transmitted back to the requesting client:

```
HTTP/1.1 200 OK<CRLF>
```

Notes: 1. All pages returned by the HTTP server are marked as *no-cache* to indicate that proxies must revalidate the request before returning a cached copy of the appropriate resource. HTTP has been interfaced with file system with which web pages can be uploaded to the eZ80 CPU at run time using either TFTP or FTP.

2. All of the web files should be uploaded to a directory specified by: `INT8 httppath[] = "/"` in the `ZTPConfig.c` file. HTTP searches for the requested web page both in the static web page array and also searches in the directory specified by the `INT8 httppath[]` variable. The order of the search is determined by the variable `UINT8 g_DefaultSearchFS=FALSE`;. If this variable is `FALSE`, the first static web page array is searched; if no results are found, the specified directory is searched. If this variable is `TRUE`, the search occurs in reverse order.

See Also

Http_Request Structure

webpage Structure

Http_Method Structure

header_rec Structure

HTTPS Functions

The Zilog TCP/IP Stack supports the https_init secure HTTP function.

HTTPS_INIT

Include

```
#include "ssl2_server.h"
```

Prototype

```
int https_init
(
const Http_Method *methods, const struct header_rec
*headers, Webpage *webpages, int port
);
```

Description

A secure webserver is initialized by calling the `https_init` API, which takes the same number and type of argument(s) as the standard HTTP server API. It is possible to have both secure and nonsecure webservers running at the same time; however, these two webservers must operate on different ports. The port number typically used for nonsecure HTTP servers is 80; for secure HTTP servers (HTTP over SSL or HTTPS) the port number typically used is 443.

Argument(s)

`http_methods`	A pointer to the supported methods structure.
`httpdef headers`	A pointer to the suppoted header structures.
`webpage`	A pointer to the webpage for which the server processes requests.
`port`	Port on which the HTTPS server listens.

Return Value(s)

The `https_init` function returns the port number on which SSL is listening upon successfully opening the SSL device.

See Also

Http_Request Structure

webpage Structure

Http_Method Structure

header_rec Structure

SNMP Functions

The Zilog TCP/IP Stack supports four SNMP functions. Table 4 provides a quick reference to each of these functions.

Table 4. SNMP Functions Quick Reference

ztpSnmpV1Init
ztpSnmpV2Init
ztpSnmpV3Init
snmpGenerateTrap

ZTPSNMPV1INIT

Include

```
#include "snmpv1.h"
```

Prototype

```
INT16 ztpSnmpV1Init (SN_TRAP_NOTIFY snTrapNotifyFunc);
```

Description

The ztpSnmpV1Init API is called from the main() routine to enable the SNMP agent. This protocol can be used to read or write values in the MIB by using the Get, GetNext or Set operations. Requests originate from the SNMP management entity are sent to the SNMP agent. After the SNMP agent processes the request, it returns relevant information to the management entity. This management entity can obtain information about objects in the MIB using the Get or GetNext requests; or, it can modify the value of an object in the MIB using the Set request. The snTrapNotifyFunc parameter is used to inform the application whenever a trap is generated by an SNMP agent.

Argument(s)

snTrapNotifyFunc	Function pointer provided by the application to SNMP agent which is used to inform the application whenever a trap is generated by an SNMP agent.

Return Value(s)

ZTPSNMP_ERR_DAEMON_CREATE	In the event of an error when creating the SNMP thread.
ZTPSNMP_ERR_SUCCESS	Success.

ZTPSNMPV2INIT

Include

```
#include "snmpv1.h"
```

Prototype

```
INT16 ztpSnmpV2Init (SN_TRAP_NOTIFY snTrapNotifyFunc);
```

Description

The `ztpSnmpV2Init` API is called from the `main()` routine to enable the SNMPv2 agent. This protocol can be used to read or write values in the MIB by using the `Get`, `GetNext` or `Set` operations which are supported in SNMPv1. SNMPv2 also defines `GetBulk`, which is used to efficiently retrieve large blocks of data. Requests from the SNMP management entity are sent to the SNMP agent. After the SNMP agent processes the request, it returns relevant information to the management entity. The management entity can obtain information about objects in the MIB using the `Get`, `GetNext` or `GetBulk` requests; or, it can modify the value of an object in the MIB using the `Set` request. The `snTrapNotifyFunc` parameter is used to inform the application whenever a trap is generated by SNMP agent. This function creates a separate thread for an SNMPv2 entity and waits for incoming requests from the SNMP manager.

Argument(s)

`snTrapNotifyFunc`	Function pointer provided by the application to SNMP agent which is used to inform the application whenever a trap is generated by SNMP agent.

Return Value(s)

`ZTPSNMP_ERR_DAEMON_CREATE`	In the event of an error when creating the SNMP thread.
`ZTPSNMP_ERR_SUCCESS`	Success.

ZTPSNMPV3INIT

Include

```
#include "snmpv3.h
```

Prototype

```
INT16 ztpSnmpV3Init (SN_TRAP_NOTIFY snTrapNotifyFunc);
```

Description

`ztpSnmpV3Init` performs the same functions as `ztpSnmpV2Init`, with the additional functionalities of authentication and encryption (if enabled). The `ztpSnmpV3Init` function supports the user security model.

Argument(s)

`snTrapNotifyFunc`	Function pointer provided by the application to an SNMP agent, which is used to inform the application whenever a trap is generated by an SNMP agent.

Return Value(s)

`ZTPSNMP_ERR_DAEMON_CREATE`	In the event of an error when creating the SNMP thread.
`ZTPSNMP_ERR_SUCCESS`	Success.

Note: SNMPv2 performs the same functions as SNMPv1, with the added functionality provided by the `GetBulk` function, and SNMPv3 performs the same functions as SNMPv2, with the added functionalities of authentication and encryption (if enabled).

SNMPGENERATETRAP

Include

```
#include "snmpV1.h"
```

Prototype

```
INT16
snmpGenerateTrap
(
INT8 *userName,
UINT8 *pMgrAddress,
UINT8 Type,
UINT32 Code,
UINT16 NumObjects,
SNMPObj *pObjList
);
```

Description

The `snmpGenerateTrap` function sends a trap to inform the SNMP manager that an event has occurred on the agent. The SNMP module in ZTP is capable of generating the following SNMPv1/SNMPv2/SNMPv3 traps:

- Cold Start trap
- Link Up trap
- Link Down trap
- Enterprise-Specific trap

A Cold Start trap is generated when the system boots, regardless of whether the system is warm-booted (for example, executing the reboot command from the shell) or cold-booted (disconnecting and reconnecting the power supply). The system generates a Link Up trap whenever a network interface is (re)activated.

For example, during system initialization, the Ethernet interface becomes active and a Link Up Trap is generated.

Conversely, when a network interface changes state from active to inactive, a Link Down trap is generated. For example, a Link Down trap is generated when the PPP link is disconnected.

Argument(s)

`userName` Indicates the name of the user from whom the trap is sent. Select one of the user names provided in the `Snmp_Usrs` global variable, which is available in the `snmpv3_conf.c` file. This field is applicable only for SNMPV3. In the case of non-SNMPV3 applications, the value passed must be NULL.

`pMgrAddress` Indicates the IP address of the SNMP manager to which the trap is sent. In the event of a NULL value, the value provided in the `g_snmpTrapTargetIP` variable (located in the `snmp_conf.c` file) is considered to be the default.

`Type` One of the following values must be used:

`SN_TRAP_COLD_START`: Cold Start trap.

`SN_TRAP_LINK_DOWN`: Link Down trap.

`SN_TRAP_LINK_UP`: Link Up trap.

`SN_TRAP_AUTH_FAILURE`: Authentication failure.

`SN_TRAP_ENTERPRISE_SPECIFIC`: User-defined trap.

`Code` A 32-bit value unique to the application that identifies the particular trap message being generated.

`NumObjects`	This parameter specifies the number of `SNMPObj` structures that are to be included in the body of the trap message. If the application-specific trap does not require any objects to be included in the trap message, set this parameter to 0.
`pObjList`	This parameter is a pointer to an array of `NumObjects` `SNMPObj` structures that identify the SNMP objects to be included in the body of the trap message. If the application-specific trap does not require any objects to be included in the trap message, set this parameter to NULL.

Return Value(s)

`<0`	If any error occurs.
`0`	Success.

See Also

SNMP Macros

SNMPObj Structure

SMTP Function

The Zilog TCP/IP Stack supports one Simple Mail Transport Protocol (SMTP) function, `mail`, which is described below.

MAIL

Include

```
#include "smtp.h"
```

Prototype

```
INT16 mail(INT8 *Addr,
UINT16 port,
INT8 *subject,
INT8 *to,
INT8 *from,
INT8 *usrname ,
INT8 *passwd,
INT8 *data,
INT8 *error,
UINT16 errorlen)
```

Description

To allow you to send email messages using the SMTP, ZTP provides the `mail` function. The `mail` function sends an SMTP mail message to a specified SMTP server or port. The function establishes a TCP connection for the mail transfer. The same API can be used for both sending SMTP mail with CRAM-MD5 algorithm authentication.

Argument(s)

`Addr`	A pointer to a character string containing the name or IP address (in decimal/dotted notation) of the SMTP server.
`port`	The SMTP port to use (normally 25).

`subject` A pointer to a character string containing the *Subject:* text in the mail message.

`to` A pointer to a character string containing the email address of the recipient.

`from` A pointer to a character string containing the email address of the sender.

`usrname` A pointer to a character string containing the user name for authentication (valid only if SMTP CRAM MD5 authentication is enabled; otherwise, it is ignored).

`passwd` A pointer to a character string containing the user password for authentication (valid only if SMTP CRAM MD5 authentication is enabled; otherwise, it is ignored).

`data` A pointer to a character string containing the body of the email, along with any additional headers.

`error` A pointer to a buffer in which ZTP can place a text string describing the reason why the mail function failed to send the message.

`errorlen` The maximum size (in bytes) of the buffer referenced by the error parameter.

Return Value(s)

If no error occurs, it returns `OK`; upon failure, it returns `SYSERR`.

Telnet Functions

Table 5 provides a quick reference to the Telnet functions supported by the Zilog TCP/IP Stack. For more information about Telnet definitions and Enumerations, see Appendix A. Definitions and Codes on page 85.

Table 5. Telnet Functions Quick Reference

telnet_init
TelnetOpenConnection
TelnetCloseConnection
TelnetSendData

TELNET_INIT

Include

```
#include "telnet_api.h"
```

Prototype

```
void telnet_init (void)
```

Description

The `telnet_init` function initializes a Telnet server. A Telnet server thread, created as a result of this function, is used to handle requests from Telnet clients.

Argument(s)

None.

Return Value(s)

None.

TELNETOPENCONNECTION

Include

```
#include "telnet_api.h"
```

Prototype

```
TELNET_RET TelnetOpenConnection
(
 IP_ADDRESS ipAddr,
 TELNET_HANDLE *telnetAppHandle,
 TELNETREAD telnetReadCallback
)
```

Description

ZTP provides the `TelnetOpenConnection` function to establish a TCP connection with a specified server. This function also sends the `ECHO` and `SUPPRESSGOAHEAD` options to the server.

Argument(s)

`ipAddr`	A `uint32` value which contains the IP address (in decimal/dotted notation) of the Telnet server.
`telnetAppHandle`	A pointer to a handle furnished by the Telnet client to the application after a connection is established successfully.
`telnetReadCallback`	A function pointer furnished by the application which is used by the Telnet client to notify the application when data is received from the transmitting end.

Return Value(s)

Upon execution, the `TelnetOpenConnection` returns the following values:

`TELNET_ALREADY_CONNECTED`	Indicates that the Telnet connection already exists.
`TELNET_INVALID_ARG`	Indicates that one or more arguments are invalid.
`TELNET_LOWER_LAYER_FAILURE`	Indicates that the TCP connect failure occurred.
`TELNET_CONNECT_FAILURE`	Indicates that an unknown error occurred.
`TELNET_SUCCESS`	A Telnet connection has been established successfully.

See Also

Telnet Data Type Definitions

Telnet Enumerations

TELNETCLOSECONNECTION

Include

```
#include "telnet_api.h"
```

Prototype

```
TELNET_RET TelnetCloseConnection ( TELNET_HANDLE
telnetAppHandle );
```

Description

To terminate a Telnet session with the server, ZTP provides the `TelnetCloseConnection` function. It terminates the TCP connection with the specified server and cleans up connection-related information for this application.

Argument(s)

`telnetAppHandle`	Handle furnished by the Telnet client during the establishment of a successful connection.

Return Value(s)

`TELNET_NO_CONNECTION`	Indicates that the Telnet connection is not yet established.
`TELNET_INVALID_ARG`	Indicates that one or more arguments are invalid.
`TELNET_FAILURE`	Indicates that an unknown error occurred.
`TELNET_SUCCESS`	The Telnet connection has been terminated successfully.

See Also

Telnet Enumerations

Telnet Data Type Definitions

TELNETSENDDATA

Include

```
#include "telnet_api.h"
```

Prototype

```
TELNET_RET TelnetSendData
(
TELNET_HANDLE telnetAppHandle,
 TELNET_DATA *telnetData,
 TELNET_DATA_SIZE telnetDataSize
)
```

Description

To send required data to the server (executing server-side commands), ZTP provides the `TelnetSendData` function, which sends each character entered to the server. The character is displayed on the console when the server echoes back the character.

Argument(s)

`telnetAppHandle`	Handle furnished by the Telnet client to the application during the establishment of a successful connection.
`telnetData`	Actual data that must be sent to the server.
`telnetDataSize`	Size of the data to be sent.

Return Value(s)

The following values are returned when the function is executed.

`TELNET_NO_CONNECTION`	Indicates that the Telnet connection is not yet established.
`TELNET_INVALID_ARG`	Indicates that one or more arguments are invalid.
`TELNET_LOWER_LAYER_FAILURE`	Indicates failure at lower layers.
`TELNET_SUCCESS`	Data has been sent successfully.

See Also

Telnet Data Type Definitions

Telnet Enumerations

TimeP Protocol Function

The Zilog TCP/IP Stack supports a TimeP protocol function, `time_rqest`, which is described below.

TIME_RQEST

Include

```
#include "date.h"
```

Prototype

```
INT16 time_rqest(void);
```

Description

The `time_rqest()` function sends a time request to the time server, the IP address of which is specified in the `struct commonServers csTbl[ ]`, which is present in the `ZTPConfig.c` file. When the time request is received from the sever, the time is updated to the real-time clock (RTC). If the time server is not present or did not reply to the request, then the RTC will not be updated. The time server should be RFC 738-compliant.

Argument(s)

None.

Return Value(s)

If successful, the `time_rqest` function returns `OK`. If this function fails, it returns either `TIMEOUT` or `SYSERR`.

DNS Functions

The Zilog TCP/IP Stack supports two DNS functions. Table 6 provides a quick reference to each of these functions.

Table 6. DNS Functions Quick Reference

name2ip
ip2name

NAME2IP

include

```
"domain.h"
```

Prototype

```
UINT32 name2ip(INT8 *nam)
```

Description

The `name2ip` function resolves a host name to IP addresses. This function sends a DNS formatted in UDP datagram with the DNS IP acquired from the `cstbl` structure.

Argument(s)

`nam`	A pointer to a character string containing the host name or URL.

Return Value(s)

The `name2ip` function returns the IP addresses of the host or URL when successful. If this function fails, it returns `SYSERR`.

IP2NAME

include

`"domain.h"`

Prototype

```
INT8 * ip2name(UINT32 ip, INT8 *nam)
```

Description

The `ip2name` function returns the DNS name for a host when furnished its IP address. This function sends a DNS formatted in UDP datagram with the DNS IP acquired from the `cstbl` structure.

Argument(s)

`ip`	IP addresses for which name resolution is required.
`nam`	A pointer to a character buffer to hold the resolved name.

Return Value(s)

The `ip2name` function returns the pointer to the character buffer holding the resolved name when successful. If this function fails, it returns `SYS-ERR`.

RARP Function

Zilog TCP/IP Stack supports a Reverse Address Resolution Protocol (RARP) function, rarpsend, which is described below.

RARPSEND

Include

```
#include "rarp.h"
```

Prototype

```
INT16 rarpsend(UINT8 ifn)
```

Description

The Reverse Address Resolution Protocol provides a mechanism for a host to obtain an IP address at startup. The host obtains a RARP response with an IP address from a network server by sending the server a RARP request using the network broadcast address and its own physical address as identification. The server is required to maintain a map of hardware addresses to IP addresses.

Argument(s)

`ifn`	Number of the Ethernet interface for which IP addresses are required.

Return Value(s)

The `rarp` function returns `OK` when successful, and `SYSERR` upon failure.

IGMP Functions

Zilog TCP/IP Stack supports two IGMP functions. Table 7 provides a quick reference to these IGMP functions.

Table 7. IGMP Functions Quick Reference

hgjoin
hgleave

HGJOIN

Include

```
#include "igmp.h"
```

Prototype

```
INT16 hgjoin
(
UINT8 ifnum,
 UINT32 ipa,
 UINT8 ttl
);
```

Description

The `hgjoin` function joins the eZ80 CPU to a specified multicast group and sends a membership report for that particular group. If the eZ80 CPU is already a member of the group, the membership report for the group will not be sent.

Argument(s)

`ifnum`	A value that should be set to the interface number of the primary Ethernet interface.

`ipa` IP addresses of the multicast group to join.

`ttl` The `ttl` parameter is the *time to live* value, which is a routing parameter used to restrict the number of gateways/multicast routers through which a multicast packet can pass.

Return Value(s)

The `hgjoin` function returns `OK` when successful and `SYSERR` upon failure.

HGLEAVE

Include

```
#include "igmp.h"
```

Prototype

```
INT16 hgleave( UINT8 ifnum, UINT32 ipa )
```

Description

The `hgleave` function removes the eZ80 CPU from the membership of the joined multicast group.

Argument(s)

`ifnum`	A value that should be set to the interface number of the primary Ethernet interface.
`ipa`	IP addresses of the multicast group to leave.

Return Value(s)

The `hgleave` function returns `OK` when successful and `SYSERR` upon failure.

TFTP Functions

The Zilog TCP/IP Stack supports two TFTP functions. Table 8 provides a quick reference to these TFTP functions.

Table 8. TFTP Functions Quick Reference

tftp_get
tftp_put

TFTP_GET

Include

```
#include "tftp.h"
```

Prototype

```
INT32 tftp_get(INT8 *Addr, INT8 *filename)
```

Description

The `tftp_get` function is used to download files from the TFTP server. This file is then stored in the thread's current working directory (CWD). If the CWD contains a file with the same name as the file that is downloaded from the server, the original file will be overwritten with the new file.

Argument(s)

`Addr`	A pointer to a character string containing the IP address of the TFTP server.
`filename`	A pointer to the name of the file to be downloaded.

Return Value(s)

Upon success, the `tftp_get` function returns the number of bytes that are loaded into the file system; it returns 0 upon failure.

TFTP_PUT

Include

```
#include "tftp.h"
```

Prototype

```
INT16 tftp_put(INT8 *Addr, INT8 *filename)
```

Description

The `tftp_put` function is used to upload files from the eZ80 CPU to the TFTP server. The file to be uploaded must be present in the thread's current working directory (CWD).

Argument(s)

`Addr`	A pointer to a character string containing the IP address of TFTP server.
`filename`	A pointer to the name of the file to be uploaded.

Return Value(s)

The `tftp_put` function returns the number of bytes sent when successful, and 0 upon failure.

FTP Functions

Zilog TCP/IP Stack supports four FTP functions. Table 9 provides a quick reference to these FTP functions.

Table 9. FTP Functions Quick Reference

ftpdinit
ftp_connect
do_programatic_login
do_a_ftp_command

FTPDINIT

Include

No header files needed. Declare function as `extern` before calling it.

Prototype

```
void ftpdinit(void);
```

Description

The `ftpdinit` API starts an FTP service on the ZTP Stack.

Argument(s)

None.

Return Value(s)

None.

FTP_CONNECT

Include

```
#include "ftpclient_api.h"
```

Prototype

```
int ftp_connect
(
INT8 * server_name,
 int server_port,
 RZK_DEVICE_CB_t * stdout
);
```

Description

The `ftp_connect` function is used to connect to a selected FTP server running on the `FTP_PORT`.

Argument(s)

`server_name`	A pointer to the IP address of the FTP server running on the remote machine (in dotted notation).
`server_port`	A number that identifies the TCP/IP port to use on the server.
`stdout`	A pointer to an integer value specifying the device to write to.

Return Value(s)

The `ftp_connect` function returns a 0 when successful; otherwise, it returns a negative value.

DO_PROGRAMATIC_LOGIN

Include

```
#include "ftpclient_api.h"
```

Prototype

```
int do_programatic_login
(
RZK_DEVICE_CB_t * stdin,
RZK_DEVICE_CB_t * stdout,
 INT8 *username,
 INT8 *passwd
);
```

Description

The `do_programatic_login` function allows the eZ80 FTP client to log into the FTP server with the specified user name and the password.

Argument(s)

`stdin`	A pointer to a console device
`stdout`	A pointer to a console device
`username`	A pointer to a username
`passwd`	A pointer to a password

Return Value(s)

The `do_programatic_login` function returns a 1 when successful and a 0 upon failure.

DO_A_FTP_COMMAND

Include

```
#include "ftpclient_api.h"
```

Prototype

```
INT16 do_a_ftp_command
(
RZK_DEVICE_CB_t * device,
 UINT16 nargs,
 INT8 *args[]
);
```

Description

ZTP provides `do_a_ftp_command` function to issue FTP commands. The command name and the arguments to the command should be provided as an array of strings.

Argument(s)

`device`	A pointer to a console device.
`nargs`	The number of arguments the command expects.
`args`	A pointer to a command name and its arguments.

Return Value(s)

The `do_a_ftp_command` function returns a 0 when successful; otherwise, it returns a negative value.

Note: The Zilog TCP/IP Stack supports a number of FTP commands. The third parameter of this API, `args[]`, can contain any FTP command, and its respective arguments, which are listed in Table 10.

Table 10. do_a_ftp_command Commands and Arguments

Command Name	Arguments	Description
ascii	None	Sets the file transfer type to the network ASCII (default).
bin	None	Sets the file transfer type to support binary image transfer.
bye	None	Terminate the FTP session with the Remote Server and exit FTP. An end of file will also terminate the session and exit.
cd	None	Remote-directory Change the working directory on the remote machine to the remote directory.
close	None	Terminate the FTP session with the Remote Server, and returns to the command interpreter. Any defined macros are erased.
delete	remote-file	Delete the file remote-file on the remote machine.
dir	[remote-directory]	Print a listing of the directory contents in the directory remote directory. If no directory is specified, the current working directory on the remote machine is used.
get	remote-file [local-file]	Retrieve the remote-file and store it on the local machine. If the local file name is not specified, it receives the same name it has on the remote machine. The current settings for type, form, mode, and structure are used while transferring the file.
hash	None	Toggle hash-sign (#) printing for each data block transferred. The size of a data block is 512 bytes.
help	[command]	Print an informative message about the meaning of a command. If no argument is supplied, FTP prints a list of the known commands.

Table 10. do_a_ftp_command Commands and Arguments (Continued)

Command Name	Arguments	Description
lcd	[directory]	Change the working directory on the local machine. If no directory is specified, the user's home directory is used.
ls	[remote-directory]	Print a listing of the contents of a directory on the remote machine. The listing includes any system-dependent information that the server chooses to include; for example, most Unix systems will produce output from the command 'ls' -l (also see nlst in this table). If remote-directory remains unspecified, the current working directory is used.
list	[remote-directory]	Synonym for `ls`.
mkdir	directory-name	Create a directory on the remote machine.
nlst	[remote-directory]	Print a list of the files in a directory on the remote machine. If remote-directory remains unspecified, the current working directory is used.
put	local-file [remote-file]	Store a local file on the remote machine. If remote-file remains unspecified, the local file name is used to name the remote file. File transfer uses the current settings for type, format, mode, and structure.
pwd	None	Print the name of the current working directory on the remote machine.
quit	None	A synonym for `bye`.
recv	remote-file [local-file]	A synonym for `get`.
rename	[from] [to]	On the remote machine, rename the [from] file to [to].
rmdir	directory-name	Delete a directory on the remote machine.
system		Show the type of operating system running on the remote machine.

Ping Function

The Zilog TCP/IP Stack supports the `ping` function, which is described below.

PING

Include

```
#include <ztptcp.h>
```

Prototype

```
UINT8 ping(UINT32 dst, UINT32 count);
```

Description

An application can use the `ping` API to determine if a remote device is using a specific IP address. The `dst` parameter specifies the IP address of the device to which an ICMP Echo Request packet is sent. The `ping` packets is sent `count` number of times.

Argument(s)

`dst`	The target of the `ping` packet.
`count`	Specifies the number of times the `ping` packet is sent.

Return Value(s)

The API waits for a response from the target device. If a response is received, then `TRUE` is returned. If this API fails to receive a response, then `FALSE` is returned.

ICMP Functionality

ZTP supports the following ICMP error returns:

- Port Unreachable
- Redirection

Port Unreachable

One rule of UDP is that if it receives a UDP datagram and the destination port does not correspond to a port that is in use, UDP responds with `ICMP port unreachable`. For example, if any host sends a UDP packet with a port number on which no application is running on the eZ80 CPU, then this error is returned.

Redirection

Based on the ICMP redirection message that the eZ80 CPU receives, it will be redirected to the next available router. Four redirection errors are supported (redirect for *network*, *host*, *TOS and network*, and *TOS and host*).

SNTP Functions

The Zilog TCP/IP Stack supports the Simple Network Transfer Protocol client protocol function, ztpSNTPClient(), which is described below.

ZTPSNTPCLIENT()

Include

```
#include <SNTPClient.h>
```

Prototype

```
INT16 ztpSNTPClient
(
INT8 *targetIPAddress,
INT16 portNum
);
```

Description

To update the system time, ZTP provides the `ztpSNTPClient` function. The function sends the time request message to the specified `targetIPAddress` and the `portNum`. The function receives the time (in seconds) from the `targetIPAddress`, converts this time into the `day, date month year hours:minutes:seconds` format and updates the system time.

Argument(s)

`targetIPAddress`	A pointer to an IP Address of the time server.
`portNum`	Port number through which the client communicates with the time server.

Return Value(s)

`SNTP_SOCKET_ERROR`	Indicates that the socket connection could not be established.
`SNTP_IOCTLSOCKET_FAIL`	Indicates that the requested command is not implemented.
`SNTP_RZK_DEV_OPEN_ERROR`	Indicates that the RZK device could not be opened.
`SNTP_SEND_TO_ERROR`	Indicates that an error occurred due to invalid descriptor or invalid socket type or the connection was refused.
`SNTP_RECIEVE_FROM_ERROR`	Indicates that an error occurred due to invalid descriptor or invalid socket type or the connection was refused or other thread is already blocked on this socket.
`SNTP_VERSION_NUMBER_ERROR`	Indicates that the version number of the client and the server mismatches.
`SNTP_MODE_ERROR`	Indicates that the mode is not of a server.
`SNTP_MEM_ALLOC_FAILURE`	Indicates that an error occurred while allocating memory.

See Also

SNTP Client Enumerations

PPP Functions

Zilog TCP/IP Stack supports two PPP functions. Table 11 provides a quick reference to each of these functions.

Table 11. PPP Functions Quick Reference

ztpPPPInit
ztpPPPStop

ZTPPPPINIT

Include

```
#include "zppp.h"
```

Prototype

```
INT16 ztpPPPInit(void)
```

Description

The `ztpPPPInit` API initializes and starts the PPP negotiations. If PPP is configured as a PPP server, then it initializes the modem and waits for the connection from the client. If the PPP is configured as client, then it initializes the link layer – high-level data link control (HDLC) or PPPoE – and if initialization is successful, it starts the Link Control Protocol (LCP) negotiations. If the `ztpPPPInit` API is called after the PPP connection is established successfully, then it returns `PPP_ALREADY_INITIALIZED` error.

Argument(s)

None.

Return Value(s)

PPP_SUCCESS	If the PPP thread is created.
PPP_FAILURE	If the PPP thread is not created.
PPP_ALREADY_INITIALIZED	If this API is called after the PPP connection is established successfully.

ZTPPPPSTOP

Include

```
#include "zppp.h"
```

Prototype

```
INT16 ztpPPPStop(void)
```

Description

The `ztpPPPStop` API forces the PPP layer to disconnect from the remote peer. If this API is called without initializing the PPP, it returns 0. If the PPP is configured as a server and the `g_PppServerAutoInitialize` Flag is set to `TRUE`, then `ztpPPPInit( )` is called internally after the connection is disconnected.

Argument(s)

None.

Return Value(s)

`1`	If PPP layer is disconnected from peer.
`0`	If PPP is not connected to any peer.

Appendix A. Definitions and Codes

This appendix describes the Zilog TCP/IP Stack's data types, structures, enumerators, constants, macros and error codes.

Data Type Definitions

This section defines a number of data types used with ZTP, including enumerators for ZTP and data types for Telnet, SSL and SNMP.

ZTP Data Types

Table 12 lists the number of ZTP data types and their definitions.

Table 12. ZTP Data Types

Data Type	Definition
UINT32	unsigned int 32-bit
INT32	signed int 32-bit
UINT24	unsigned int 24-bit
INT24	signed int 24-bit
UINT	unsigned int
INT	signed int
UINT16	unsigned short
INT16	signed short
INT8	signed char
UINT8	unsigned char
WORD	UINT16
DWORD	UINT32

Telnet Data Types

Table 13 lists definitions of the Telnet data types.

Table 13. Telnet Data Type Definitions

Data Type	Definition
TELNET_HANDLE	Unsigned char
TELNET_DATA_SIZE	Unsigned short
TELNET_DATA	Unsigned char
IP_ADDRESS	Unsigned long
TELNETREAD	typedef void (*TELNETREAD)(TELNET_HANDLE, UINT8 *, UINT16);

SNMP Data Types

Table 14 lists the Simple Network Management Protocol data type and its definition.

Table 14. SNMP Data Types

Data Type	Definition
ZTPSNMP_TRAP_NOTIFY	typedef void (*ZTPSNMP_TRAP_NOTIFY) (INT8)

ZTP Error Codes

This section lists the error codes defined by ZTP.

ZTP Core Error Codes

Table 15 lists a number of error codes returned by the networking APIs.

Table 15. ZTP Core Error Codes

Error	Code
ZTP_SOCK_OK	(INT16)0
ZTP_SOCK_ERR	(INT16)–1
EAFNOSUPPORT	(INT16)–2
EOPNOTSUPP	(INT16)–3
EFAULT	(INT16)–4
EISCONN	(INT16)–5
ECONNREFUSED	(INT16)–6
EPROTONOSUPPORT	(INT16)–7
ENOBUFS	(INT16)–8
EINVAL	(INT16)–9
EBADF	(INT16)–10
ENOCON	(INT16)–11
EMFILE	(INT16)–12
EINVALBKLOG	(INT16)–13
EPIPE	(INT16)–14
EFETNOSUPPORT	(INT16)–15
EDEADSOCK	(INT16)–16
EIOBINPRGRSS	(INT16)–17
OK	1
SYSERR	(–1UL)

Telnet Enumerations

The following enumerator governs Telnet errors.

```
TELNET_RET

typedef enum{
 TELNET_SUCCESS,
 TELNET_BEGIN_ERROR_CODE = 0x400,
 TELNET_INVALID_ARG = TELNET_BEGIN_ERROR_CODE,
 TELNET_CONNECT_FAILURE,
 TELNET_CLOSE_FAILURE,
 TELNET_NO_CONNECTION,
 TELNET_ALREADY_CONNECTED,
 TELNET_OVER_SIZED_DATA,
 TELNET_ALREADY_INITIALIZED,
 TELNET_LOWER_LAYER_FAILURE,
 TELNET_FAILURE
}TELNET_RET;
```

SNTP Client Enumerations

The following enumerator governs the error values returned by the `ztpSNTPClient()` API.

```
typedef enum{
 SNTP_SUCCESS = 0,
 SNTP_SERVER_RETURN_SUCCESS=0,
SNTP_SOCKET_CREATION_ERROR,
 SNTP_SEND_TO_ERROR,
 SNTP_RECIEVE_FROM_ERROR,
 SNTP_IOCTLSOCKET_FAIL,
 SNTP_RZK_DEV_OPEN_ERROR,
 SNTP_MEM_ALLOC_FAILURE
 SNTP_VERSION_NUMBER_ERROR,
 SNTP_MODE_ERROR
}SNTP_ERRORS;
```

SNMP Enumerations

```
typedef enum ztpsnmpErr
{
 ZTPSNMP_ERR_SUCCESS ,
 ZTPSNMP_ERR_DAEMON_CREATE = -100,
 ZTPSNMP_ERR_SOCK_CREATE ,
 ZTPSNMP_ERR_SOCK_BIND ,
 ZTPSNMP_ERR_WRONG_ASN1_CODING,
 ZTPSNMP_ERR_INVALID_COMMUNITY_NAME,
 ZTPSNMP_ERR_INVALID_REQ_TYPE,
 ZTPSNMP_ERR_INVALID_VERSION,
 ZTPSNMP_ERR_MEM_ALLOC_FAILURE,
 ZTPSNMP_ERR_BUILD_VAR_BIND,
 ZTPSNMP_ERR_FRAME_REPLY,
 ZTPSNMP_ERR_INVALID_REQ_OP,
 ZTPSNMP_ERR_SOCK_SEND,
 ZTPSNMP_ERR_NO_SUCH_MIB,
 ZTPSNMP_ERR_INVALID_MIB_OPERATION,
 ZTPSNMP_ERR_MATCH_NOT_FOUND
}ZTP_SNMP_ERR_CODES ;
```

ZTP Macros

This section lists the number of macros defined by ZTP, including macros for the ZTP core and the `ioctlsocket` API.

ZTP Core Macros

Table 16 lists the macro codes returned by the networking APIs.

Table 16. ZTP Core Macros

Macro	Code
SOCK_STREAM	0
SOCK_DGRAM	1
AF_INET	1
tcp_FlagPUSH	0x0008

ioctlsocket Macros

Table 17 lists the macro codes used by the ioctlsocket API.

Table 17. ioctlsocket Macros

Macro	Code
FIONBIO	1
FIONREAD	2
SIOCATMARK	3
FCNCLBIO	4
FUDPCKSUM	5
UDPTIMEOUT	6
FDISNAGLE	7

Table 17. ioctlsocket Macros (Continued)

Macro	Code
FENANAGLE	8
FIONWRITE	9
TCPKEEPALIVE_ON	13
TCPKEEPALIVE_OFF	14

SNMP Macros

Table 18 lists the macro codes used by the SNMP.

Table 18. SNMP Macros

Macro	Code
SN_TRAP_COLD_START	0
SN_TRAP_WARM_START	1
SN_TRAP_LINK_DOWN	2
SN_TRAP_LINK_UP	3
SN_TRAP_AUTH_FAILURE	4
SN_TRAP_EGP_NWIGHBOR_LOSS	5
SN_TRAP_ENTERPRISE_SPECIFIC	6

ZTP Data Structures

This section lists a number of data structures defined by ZTP, including structures for the ZTP core, the ioctlsocket API, and the Secure Sockets Layer.

ZTP Core Data Structures

The sockaddr data structure is used in conjunction with networking APIs.

```
struct sockaddr
{
 INT16 sa_family;
 INT8 sa_data[14];
};
```

HTTP Data Structures

The following data structures are used in conjunction with HTTP APIs.

header_rec Structure

Http_Hdr Structure

http_params Structure

Http_Request Structure

Http_Method Structure

staticpage Structure

webpage Structure

header_rec Structure

```
struct header_rec
{
 INT8 *name;
 UINT16 val;
};
```

Http_Hdr Structure

```
typedef struct http_hdr
{
 UINT8              key;
 INT8*              value;
} Http_Hdr;
```

http_params Structure

```
struct http_params
{
/** The key, typically an http header. */
 UINT8 *key;
/** The value associated with that key. */
 INT8 *value;
};
```

Http_Request Structure

```
typedef struct http_request {
 UINT8              method;
 UINT16             reply;
 UINT8              numheaders;
 UINT8              numparams;
 UINT8              numrespheaders;
 INT16              fd;
 const struct http_method * methods;
 const struct webpage * website;
 const struct header_rec * headers;
 INT8 * bufstart; /* first free space */
 UINT8 * extraheader;
 Http_Hdr rqstheaders[HTTP_MAX_HEADERS];
 Http_Hdr               respheaders[HTTP_MAX_HEADERS];
 Http_Params            params[HTTP_MAX_PARAMS];
 Http_Auth              *AuthParams;
 INT8                   buffer[HTTP_REQUEST_BUF];
 INT8                   keepalive;
} Http_Request;
```

Http_Method Structure

```
typedef struct http_method
{
 UINT8                  key;
 INT8                   *name;
 void                   (*method)(Http_Request *);
} Http_Method;
```

staticpage Structure

```
struct staticpage {
/** A pointer to the actual contents of the page. This
/* could be the actual string representing the entire
/* page, or an array of bytes (e.g. the array produced
/* by the mkwebpage program). */
 UINT8 *contents;
/* The size of the above array, since it is not null
/* terminated. If this is actually a string, it would
/* be equal to strlen(array). **/
 INT32   size;
};
```

webpage Structure

```
struct webpage
{
 UINT8 type;
 INT8 *path;
 INT8 *mimetype;
/* Either a structure defining the static page, or the
/* 'cgi' function which will generate this page. **/
 union
 {
  const struct staticpage *spage;
  INT16 (*cgi)(struct http_request *);
 } content;
};
```

SNMP Data Structures

The following data structures are used in conjunction with SNMP APIs.

OID Structure

```
typedef struct objId
{
 UINT16             objId[32];
 UINT16             oidLen ;
}OID ;
```

SNMPDisplayStr Structure

```
typedef struct sn_descr_s
{
  void              *pData;
  UINT16            Length;
} SNMPDisplayStr;
```

SNMPObjValue Structure

```
typedef union snmpObjVal
{
 void                   *pData;
 OID *pOid;             // Object Identifier
 SNMPDisplayStr *pDescr;// Octet String, big
                        //integer, Display String
 INT8                   *pInt8;
 INT16                  *pInt16;
 INT32                  *pInt24;
 INT32                  *pInt32;
 UINT8                  *pUint8;
 UINT16                 *pUint16;
 UINT32                 *pUint24;
 UINT32                      *pUint32;
 UINT8                       *pPhys;
 UINT32                      *pIP;
 UINT32                      *pCounter;
 UINT32                      *pGauge;
 UINT32                      *pTimeTicks;
} SNMPObjValue;
```

SNMPObjLs Structure

```
typedef struct snmpObjList
{
 OID                    objId ;
 SNMPObjValue           objVal;
 UINT8                  objValType;
 UINT8                  *objMem
}SNMPObjLs;
```

SNMPObj Structure

```
typedef struct snmpObj
{
  UINT16                *objId;
  UINT8                 objIdLen;
  UINT8                 objValType;
  SNMPObjValue          objVal;
}SNMPObj ;
```

ZTP C Run-Time Library Functions

ZTP includes its own set of C run-time library functions, in addition to those available in the ZDS II C Compiler's run-time library. ZTP's C run-time routines are named differently so as to differentiate with the ZDS II C Compiler's run-time library routines.

For more information about the ZDS II C Compiler's run-time library, refer to the Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144).

Table 19 provides a brief description about library routines.

Table 19. Library Routines

Library Routine	Description
xc_ascdate	Convert time to ASCII.
xc_fprintf	Print formatted text to a specified device.
xc_printf	Print formatted text onto a console.
xc_sprintf	Print formatted text into a specified buffer.
xc_strcasecmp	Case-insensitive string comparison.
xc_index	Find character in a string.

XC_ASCDATE

Include

```
#include "xc_lib.h"
```

Prototype

```
INT16 xc_ascdate (UINT32 time, INT8 *str)
```

Description

Convert time to ASCII. The `xc_ascdate` function takes its first argument as the number of seconds since midnight, January 1, 1970, and produces an ASCII string for the date and time corresponding to that time. The ASCII string is copied into the second argument, which must point to a buffer large enough to contain it (twenty characters including the terminating NULL).

Argument(s)

`time`	The time, in seconds, since midnight, January 1st, 1970.
`str`	A pointer to a user-supplied buffer to contain an output string.

Return Value(s)

This function always returns `OK`.

XC_FPRINTF

Include

```
#include "xc_lib.h"
```

Prototype

```
INT16 xc_fprintf (RZK_DEVICE_CB_t * descriptor, INT8
*format,...)
```

Description

Print formatted text to the device specified in the `descriptor` parameter. The `xc_fprintf` function interprets its second argument as an ASCII format to use in printing its remaining arguments to a device identified by first argument. The format contains simple text and special format codes that are identified by a preceding percent (%) character.

`b` Print an `int` as a binary number.

`c` Print a single character.

`d` Print an `int` as a decimal number.

`o` Print an `int` as an octal number.

`s` Print a string.

`u` Print an unsigned `int` as a decimal number.

`x` Print an `int` as a hexadecimal number.

`%` Print a % character.

In addition, the following elements can be inserted between the percentage symbol (%) and the format code to modify the output:

- An integer specifying the minimum field width
- A minus sign, indicating left justification

- The letter l, indicating a long data type
- An asterisk indicating the field width to be taken from the next unprocessed argument
- A period followed by an integer, indicating the maximum field width for a string

xc_fprintf uses the same conversion specifiers as kprintf.

Argument(s)

descriptor	A pointer to an integer value specifying the device to print to.
format	A pointer to a string defining what to print.
...	Arguments corresponding to the format codes, if any.

Return Value(s)

When successful, the xc_fprintf function returns OK.

XC_PRINTF

Include

```
#include "xc_lib.h"
```

Prototype

```
INT16 xc_printf (INT8 *format,...)
```

Description

Print Formatted Text. The `xc_printf` function prints formatted text onto a console. It is equivalent to calling `xc_printf` with a first argument of `CONSOLE`.

Argument(s)

`format`	A pointer to a string defining what to print.
…	Arguments corresponding to the format codes, if any.

Return Value(s)

When successful, this function returns `OK`.

See Also

xc_fprintf

XC_SPRINTF

Include

```
#include "xc_lib.h"
```

Prototype

```
INT8 * xc_sprintf (INT8 *buffer, INT8 *format,...)
```

Description

Print Formatted Text. The `xc_sprintf` function prints formatted text into a specified buffer. Except for the output medium, it is identical to the `xc_fprintf` function.

Argument(s)

See the xc_fprintf function.

Return Value(s)

When successful, the `xc_sprintf` function returns `OK`.

See Also

xc_fprintf

XC_STRCASECMP

Include

```
#include "xc_lib.h"
```

Prototype

```
INT16 xc_strcasecmp (INT8 *str1, INT8 *str2)
```

Description

Case-insensitive String Comparison. The `xc_strcasecmp` function performs a byte-by-byte comparison of two strings, in which it looks for the first character that differs other than by its case. If the first character in the first string that does not match is less than its corresponding character in the second string – or if the first string is shorter than the second string – a negative value is returned. If the character is larger than its corresponding character in the second string – or if the second string is shorter than the first – a positive nonzero value returns. If the two strings are the same (except, possibly, in their case), a zero is returned.

Argument(s)

`str1`	A pointer to first of the two strings in the comparison.
`str2`	A pointer to second of the two strings in the comparison.

Return Value(s)

The `xc_strcasecmp` function returns an integer that describes whether the first string is less than, equal to, or greater than the second string.

XC_INDEX

Include

```
#include "xc_lib.h"
```

Prototype

```
INT8 *xc_index (INT8 *str, INT8 c)
```

Description

Find a Character in a String. The `xc_index` function searches a string for the first occurrence of the specified character, and returns a pointer to the character.

Argument(s)

`str`	A pointer to the string to be searched.
`c`	The character to search for.

Return Value(s)

If the character is found, a pointer to its location in the string is returned. If no match is found, a `NULL` pointer is returned.

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base or consider participating in the Zilog Forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.