eZ80® Family of Microprocessors

Zilog TCP/IP Software Suite Programmer's Guide

Reference Manual

RM004114-1211

Copyright ©2011 Zilog Inc. All rights reserved.

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

©2011 Zilog Inc. All rights reserved. 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concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded.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
Dec 2011	14	Globally updated for the ZTP v2.4.0 release.	All
Aug 2010	13	Globally updated for the ZTP v2.3.0 release; modified configwlan; added keyIndex and pass-phrase commands.	163, 165, 166
Nov 2008	12	Globally updated for the ZTP v2.2.0 release; added Configuring PPP, 2, scan, join, configwlan, setipparams sections. Updated Build Options for the ZDSII Environment, Common Libraries, Figure 1, 1, User Configuration Details, 3, 4, Configuring the SHELL, Configuring the Management Information Base, Configuring the Simple Network Management Protocol, Connecting to a Remote Host Across a Network, How to Use DNS, How to Use PPP, How to Use SNMP, SNMP Objects, Adding Objects to the MIB, Using SNMP to Manipulate Leaf Objects in the MIB, How to Add a Table to the MIB, The SNMP_GET_FUNC Support Routine, The SNMP_SET_FUNC Support Routine, Updating SNMP Values, 1 sections. Added Appendix B. Guidelines to Porting SNMP and PPP Applications.	3, 4, 7, 14, 15, 16, 24, 28, 29, 31, 34, 58, 66, 69, 77, 81, 84, 87, 89, 92, 94, 98, 103, 105,160, 161, 162,163
Jul 2007	11	Globally updated for branding.	All
Jul 2007	10	Globally updated for the ZTP v2.1.0 release.	All

Date	Revision Level	Description	Page
Jun 2007	09	Updated for style. Added User Configuration Details. Updated SNMP Objects, How to Use SMTP, 1, 8, Configuring PPP, How to Use SNMP, cd, gettime, settime, sleep, Stub Library sections. Removed ZTP Resource Usage, Operating system Overview, Protocol Overview, and ZTP HTTP Server Overview. Removed Build Operations for IAR Embedded Workbench Environment, Understanding SNMP, and Getting started with ZTP sections.	All
Jul 2006	08	Globally updated for the ZTP v2.0.0 release.	All

Table of Contents

Revision History iii

Introduction viii
- About This Manual viii
- Intended Audience viii
- Manual Organization ix
- Software Release Versions ix
- Safeguards x
- Online Information x

Product Overview 1
- System Features 1
- ZTP Software 2

ZTP Configuration. 5
- Network-Configurable Parameters 5
  - Datalink Layer Configuration 5
  - Configuring PPP 7
  - User Configuration Details 14
  - Network Configuration 16
- Build Options for the ZDS II Environment 31
  - Libraries 33

Using ZTP 36
- How to Use HTTP 36
  - Initializing HTTP 36
  - Building Web Pages 50
- How to Use TFTP 53
- How to Use SMTP 54
- How to Use the Telnet Server 57

How to Use the Telnet Client 57
Connecting to a Remote Host Across a Network 58
Closing a Connection to a Remote Host 59
Sending Data to a Remote Host 60
How to Use the FTP Server 61
How to Use the FTP Client 62
Connecting to an FTP Server 62
Log In With a Username and Password 63
Issuing FTP Commands 63
How to Use BOOTP 64
How to Use DHCP 64
How to Use DNS 66
How to Use IGMP 67
How to Use TIMEP 68
Requesting the Time 68
How to Use PPP 69
How to Use the HTTPS Server 71
How to Use the Shell 74
How to Use SNMP 77
Working with SNMPv3 101
How to Use the SNTP Client 102
ZTP Shell Command Reference 103
Appendix A. Creating ZTP Shell Commands 167
ping Command Example 167
Appendix B. Guidelines to Porting SNMP and PPP Applications 169
API Changes 169
Index 172
Customer Support 179

Introduction

This reference manual describes the architecture of the Zilog TCP/IP (ZTP) Software Suite, which features a set of TCP/IP software libraries, board support packages (BSPs), application protocols and a version of the Zilog Real-Time Kernel (RZK) for Zilog's eZ80 microprocessors and eZ80Acclaim! microcontrollers. The ZTP libraries require minimum memory and transform these devices into efficient embedded webservers.

Note: This document describes the ZTP Software Suite v2.3.0 and later. If you are using ZTP Software Suite v1.3 or a prior version, refer to the Zilog TCP/IP Software Suite v1.3.4 Programmer's Guide (RM0008), which is available free for download from the Zilog website.

About This Manual

Zilog recommends that you read and understand the complete manual before using this product to develop code. This manual describes how to develop software using the ZTP Software Suite. For additional information regarding the ZTP Software Suite, please refer to the Zilog TCP/IP Stack API Reference Manual (RM0040).

Intended Audience

This document is written for Zilog customers who have exposure to microprocessors and networking fundamentals.

Manual Organization

This reference manual is organized into the following chapters and appendices.

Product Overview

This chapter describes the product overview of ZTP.

ZTP Configuration

This chapter discusses the details about ZTP's configurable parameters, and the build options for ZDS II environment.

Using ZTP

This chapter describes how to use the various protocols available in the ZTP Software Suite.

ZTP Shell Command Reference

This chapter describes the ZTP shell commands.

Appendix A. Creating ZTP Shell Commands

This appendix provides an example of how to create your own shell commands.

Appendix B. Guidelines to Porting SNMP and PPP Applications

This appendix describes the differences between ZTP v2.1.0 (and earlier) and ZTP v2.2.0 and later releases.

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

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

Product Overview

The Zilog TCP/IP (ZTP) Software Suite includes a preemptive, multitasking real-time kernel, the Zilog Real-Time Kernel (RZK), which is an operating system developed by Zilog. ZTP contains a set of libraries that implement an embedded TCP/IP stack. In addition, ZTP also contains a number of application protocols.

System Features

The key features of ZTP include:

- Compact, preemptive, multitasking real-time kernel with interprocess communications (IPC) support and soft real-time attributes
- Complete TCP/IP stack
- Compatible with all members of the eZ80 family
- Implementation of the following standard network protocols:

ARP	DHCP	DNS	FTP	HTTP	SSL	ICMP
IGMP	IP	PPP	RARP	SMTP	TCP	SNMP
UDP	SNTP	Telnet	TFTP	TIMEP		

- Interoperable with all RFC-compliant TCP/IP and Network Protocol implementations to provide seamless connectivity
- A board support package (BSP) containing an Ethernet Media Access Controller (EMAC) driver for the CrystalScan 8900A, the eZ80F91 integrated EMAC, and a WLAN driver for the Realtek 8711 chipset
- A serial driver

- Final stack size is link/time-configurable and determined by the protocols included in the build
- Application demonstrations

ZTP Software

The ZTP software is comprised of two planes.

1. The first plane represents the Zilog's RTOS, RZK; it is referred to as the *OS plane*. The OS plane includes a scheduler, a memory manager, and IPC services.
2. The second plane represents the embedded TCP/IP protocol stack; it is referred to as the *stack plane*. Modules in the stack plane typically require the services of the OS plane to ensure that they can coexist with other applications that compete for the processor.

Figure 1 displays the architecture of the Zilog TCP/IP protocol stack, which corresponds to the Open Systems Interconnect (OSI) model. This figure also displays the locations in which the application can interface to ZTP; these locations are denoted by the color teal.

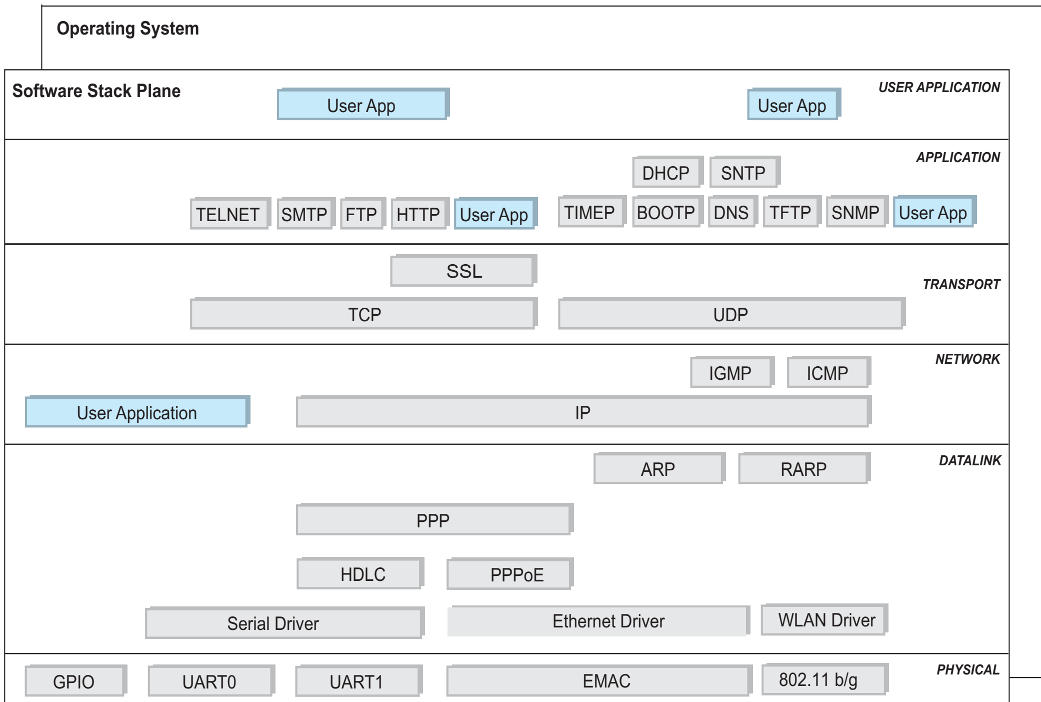

Figure 1. Architecture of the Zilog TCP/IP Protocol Stack

Many TCP/IP protocols are designed to operate on a client-server model. Therefore, Table 1 lists the complete name of each ZTP protocol and also indicates whether ZTP implements a client or a server for each of the application protocols displayed in Figure 1. Protocols that implement the Transport, Network, and Datalink layers typically operate in Peer-to-Peer mode, requiring both a client component and a server component to allow interoperability. These protocols are designated as Peer in Table 1.

Table 1. ZTP Protocol Layers

Protocol	Expansion	Client, Server, or Peer
ARP	Address Resolution Protocol	Peer
DHCP	Dynamic Host Configuration Protocol	Client
DNS	Domain Name Server	Client
FTP	File Transfer Protocol	Client and Server
HTTP	Hyper Text Transfer Protocol	Server
ICMP	Internet Control Message Protocol	Peer
IGMP	Internet Group Management Protocol	Peer
IP	Internet Protocol	Peer
PPP	Point-to-Point Protocol	Peer
RARP	Reverse Address Resolution Protocol	Peer
SMTP	Simple Mail Transfer Protocol	Client
SNMP	Simple Network Management Protocol	Server
SSL	Secure Socket Layer	Server
TCP	Transmission Control Protocol	Peer
Telnet	Telnet	Client and Server
TFTP	Trivial file Transfer Protocol	Client
TIMEP	Time Protocol	Client
UDP	User Datagram Protocol	Peer
SNTP	Simple Network Time Protocol	Client
PPPoE	Point-to-Point Protocol over Ethernet	Client

ZTP Configuration

ZTP is highly configurable and scalable. This chapter discusses the details about its configurable parameters. ZTP configuration is divided into the following three main modules:

1. RZK configuration
2. BSP configuration
3. Network configuration

For more information about RZK and BSP configurations, refer to the Zilog Real-Time Kernel User Manual (UM0075).

Network-Configurable Parameters

This section discusses the configurable parameters of the Datalink Layer, which includes the point-to-point protocol (PPP), the network stack, and the shell.

Datalink Layer Configuration

Table 1 lists a number of device configurations that are used by ZTP in the Datalink Layer. In the table, the default system values are identified by an asterisk. To modify these default values, you must include the corresponding file in the project workspace and modify it according to project requirements.

Table 1. Datalink Layer Configurable Parameters

Component Configuration	File To Modify	Variable/Macro To Modify	Valid Configuration Values
EMAC Driver	RZK\Conf\ emac_conf.c	f91_mac_addr	EMAC address (default values in hexadecimal): 0x00, 0x90, 0x23, 0x00, 0x04, 0x04
		F91_emac_config (valid only for the eZ80F91 platform)	Structure that contains the following values for initializing the EMAC device: txBufSize = 0–1368* mode = F91_10_HD; 10 Mbps Half Duplex mode = F91_10_FD; 10 Mbps Full Duplex mode = F91_100_HD; 100 Mbps Half Duplex mode = F91_100_FD; 100 Mbps Full Duplex mode = F91_AUTO; Autosense bufSize = 0-32*
UART Driver	RZK\Conf\ uart_conf.c	serparams	Structure that contains the following values for initializing the UART device: baud = 2400, 9600 or 19200, 38400, 57600*, or 115200 databits = 7 or 8* stopbits = 1* or 2 parity = PAREVEN, PARODD, or PARNONE*

Note: *Default value: for example, 1368 is default value for upper limit of txBufSize.

Table 1. Datalink Layer Configurable Parameters (Continued)

Component Configuration	File To Modify	Variable/Macro To Modify	Valid Configuration Values
UART Driver	RZK\Conf\uart_conf.c	serparams (cont'd)	Settings can also contain combinational values with logical OR (|)operation: SERSET_DTR_ON* (UART1): Assert data terminal ready (DTR) on open, reset it on close. SERSET_RTSCTS* (UART1): Use RTS/CTS hardware flow control. SERSET_DTRDSR: Use DTR/DSR hardware flow control. SERSET_XONXOFF: Use XON/XOFF SW flow control. SERSET_ONLCR* (UART0): Map NL to CR-NL on output. SERSET_SYNC* (UART0): Use Synchronous routines instead of interrupts. SERSET_IGNHUP* (UART0): Ignore Hangup (CD drop).

Note: *Default value: for example, 1368 is default value for upper limit of txBufSize.

Configuring PPP

The `PPP_CONF.c` file must be configured to enable communication over Point-to-Point (PPP) protocol. PPP supports HDLC and PPPoE as the lower layer interfaces. If the PPPoE macro is defined in the project workspace, then the project will be configured for PPPoE.

Changes to PPP parameters can be made by changing a parameter in the `PPP_CONF` structure in `PPP_CONF.c` file.

> **Note:** ZTP supports PPPoE only as a client.

The members of the `PPP_CONF` structure is explained below.

```
struct PppConf {
    INT8            *myuser;
    INT8            *mypassword;
    UINT16          auth;
    UINT16          MRU;
    UINT16          ConfigTimer;
    UINT16          MaxConfigRequest;
    UINT8           ppp_mode;
};
```

INT8 *myuser. A pointer to a user name string that can be used for authentication. If ZTP performs as a PPP client, then this user name is sent to the peer for authentication and if the ZTP performs as a PPP server, then the user name can be used to authenticate the connecting peer.

INT8 *mypassword. A pointer to a password string that can be used for authentication. If ZTP performs as a PPP client then this is the password that will be sent to the peer for authentication and if ZTP performs as a PPP server, then the password can be used to authenticate the connecting peer.

UINT16 auth. A value that specifies the authentication protocol to use. A value of 0 means that the peer is not authenticated. A value of `ZTP_PPP_PAP` requires the remote to authenticate using the PAP protocol or a value of `ZTP_PPP_CHAP` requires the remote to authenticate using `CHAP` protocol.

UINT16 MRU. The maximum receive unit (MRU) specifies the largest packet size that can be received from the peer.

UINT16 ConfigTimer. The time interval between two PPP configuration request packets.

UINT16 MaxConfigRequest. A value that specifies the maximum number of times a configuration request packet is sent and if either unanswered or rejected before the connection is terminated.

UINT8 ppp_mode. Mode of operation of the PPP layer (`PPP_CLIENT` or `PPP_SERVER`).

The members of the `PppNetworkConf` structure is explained below.

```
struct PppNetworkConf {
   INT8           *myaddress;
   INT8           *peeraddress;
   INT8           *PrimaryDns;
   INT8           *SecondaryDns;
   INT8           *PrimaryNbns;
   INT8           *SecondaryNbns;
};
```

INT8 *myaddress. A string that contains the four-octet IP address that is used for the local end of the connection. The value 0 indicates that the local IP is obtained by negotiation from the other end of the connection.

INT8 *peeraddress. A string that contains the four-octet IP address that is used for the remote end of the connection. The value 0 indicates that the remote IP is obtained by negotiation from the other end of the connection. If a value is specified, the connection is established only if the remote end negotiates the same address.

INT8 *PrimaryDns. A string that contains the four-octet primary DNS server IP address that is used for the remote end of the connection. The value 0 indicates that the DNS server IP is obtained through negotiation from the other end of the connection. If a value is specified, then the DNS server IP address is offered to the peer, else it is not offered.

INT8 *SecondaryDns. A string that contains the four-octet secondary DNS server IP address that is used for the remote end of the connection. If a value is specified, then the secondary DNS server IP address is offered

to the peer. Otherwise, if the value is 0, then the secondary DNS server IP address is not offered to the peer.

INT8 *PrimaryNbns. A string that contains the four-octet primary NBNS server IP address that is used for the remote end of the connection. If a value is specified, then the secondary DNS server IP address is offered to the peer. Otherwise, if the value is 0, then the secondary DNS server IP address is not offered to the peer.

INT8 *SecondaryNbns. A string that contains the four-octet secondary NBNS server IP address that is used for the remote end of the connection. If a value is specified, then the secondary DNS server IP address is offered to the peer. Otherwise, if the value is 0, then the secondary DNS server IP address is not offered to the peer.

The following are the other configuration parameters:

UINT8 g_EnablePppDebug. Setting this variable to TRUE enables the debug prints on the console. The debug prints on the console provides the summary of all of the LCP, PAP/CHAP, IPCP options during PPP negotiations. Setting this variable to FALSE disables all of the debug prints in PPP.

UINT8 g_PppServerAutoInitialize. When a PPP connection is terminated (either by ZTP or by the peer), the ZTP PPP protocol reinitializes and accepts new connection requests from clients, but only if the PPP connection is configured as a server for dial-up or as a direct cable connection, and its variable is set to TRUE. If the variable is set to FALSE, then the application calls the PPP initialization routine (`ztpPPPInit`). If configured as a client, then this variable is not affected.

Configuring PPP with HDLC

Structures of the type `chatscript_t` contain chat scripts (character strings) that are used in exchanges between the modem and the PPP software to perform tasks such as, answering an incoming call (PPP server) or dialing a specific phone number (PPP client). There are four default `chatscript_t` scripts in the `PPP_CONF.c` file that can be used as a

starting point in creating your projects. Table 2 on page 11 lists the default modemchat scripts.

```
typedef struct chatscript {
 INT8            *SendScript;
 INT8            *ReceiveScript;
 UINT16          TimeOutValue;
}chatscript_t;
```

INT8 *SendScript. A pointer to a string that is sent to the modem. NULL is used if the string is not sent.

INT8 *ReceiveScript. A pointer to a string that is expected from the modem; use NULL if no response is expected.

UINT16 TimeOutValue. The maximum number of seconds to wait for an expected string from the external device. After sending a string, the modem control software sets a timer and waits for the expected string. If the expected string arrives before the time-out period, the timer is stopped and the next `modemchat` in the script is executed. However, a time-out occurs before the expected string is received, the PPP layer closes the serial port and abandons this connection attempt. If the time-out is specified as 0, the time-out period is set to an infinite value.

Table 2. Modemchat Scripts and their Description

Modemchat Scripts	Description
`DialServer`	The `DialServer` is used when the ZTP performs as a PPP server answering incoming calls from an external modem.
`DialClient`	The `DialClient` is used when the ZTP performs as a PPP client dialing outgoing calls using an external modem.

Table 2. Modemchat Scripts and their Description

Modemchat Scripts	Description
`DccServer`	The `DccServer` is used when the ZTP performs as a PPP server using Direct Cable Connect (DCC, NULL modem) to a client PC with Windows.
`DccClient`	The `DccClient` is used when the ZTP performs as a PPP client using Direct Cable Connect (DCC, NULL modem) to a server PC with Windows.

```
typedef struct HdlcConfig {
  INT8 *SerDevName;
  struct chatscript *chat;
  UINT16 nchat;
}HdlcConfig_t;
```

INT8 *SerDevName. Name of the serial device to which the modem is connected (`SERIAL0` or `SERIAL1`).

struct chatscript *chat. Pointer to the chat script used (`DialServer/DialClient/DccServer/DccClient`).

UINT16 nchat. Number of entries in the chat script pointed by `chatscript *chat` structure.

Examples of the PPP settings for server and client appear in the code fragments are provided below.

PPP Server Settings

```
struct PppConf PPP_CONF = {
  "ez80",         /* User ID */
  "Zilog123",     /* Password */
  ZTP_PPP_PAP,    /* ZTP_PPP_CHAP,ZTP_PPP_CHAP
Authentication protocol*/
  1400,           /* MRU */
  3,              /* ConfigTimer-Time intervals b/w conf
                  /* requests */
```

```
  6,              /* MaxConfigRequest-Max No. of Conf
                  /* requests */
  PPP_SERVER,     /*PPP mode-PPP_SERVER or PPP_CLIENT */
};

struct PppNetworkConf PppNwConf = {
   "192.168.2.1", /* My IP Address */
   "192.168.2.2", /* Peer IP Address */
   "192.168.2.10",/* Peer Primary DNS IP Address */
   0,             /* Peer Secondary DNS IP Address */
   0,             /* Peer Primary NBNS IP Address */
   0              /* Peer Secondary NBNS IP Address */
};
```

PPP Client Settings

```
struct PppConf PPP_CONF = {
  "ez80",         /* User ID */
  "Zilog123",     /* Password */
  ZTP_PPP_PAP,    /* ZTP_PPP_CHAP,ZTP_PPP_CHAP
                  /* Authentication protocol*/
  1400,           /* MRU */
  3,              /* ConfigTimer-Time intervals b/w conf
                  /* requests */
  6,              /* MaxConfigRequest-Max No. of Conf
                  /* requests */
  PPP_CLIENT,     /*PPP mode-PPP_SERVER or PPP_CLIEN */
};
struct PppNetworkConf PppNwConf = {
0,                /* My IP Address */
0,                /* Peer IP Address */
0,                /* Peer Primary DNS IP Address */
0,                /* Peer Secondary DNS IP Address */
0,                /* Peer Primary NBNS IP Address */
0                 /* Peer Secondary NBNS IP Address */
};
```

Sample PPP HDLC Server Settings Using An External Modem

```
chatscript_t DialServer[] = {
```

```
  {"ATE&F&K3&S1\r","OK",30},
  {NULL, "RING",0},
  {"ATA\r","CONNECT",60},
  };
HdlcConfig_t g_HdlcConf = {
   "SERIAL1",    /* Serial port on which modem */
   DialServer,
   3
};
```

Configuring PPPoE

The `PPPoE_conf.c` file contains the following variables which can be modified according to the requirements.

UINT8 g_PPPoE_PADI_RexmitCount. Maximum value of the retransmission count for PADI and PADO request, if no response is received.

UINT32 g_PPPoE_PADI_BlockTime. Maximum value of the block time in RZK ticks for a response from the server for PADI and PADO packets sent.

User Configuration Details

ZTP maintains a common list of the user name and password for FTP, SHELL, and Telnet. These details are stored in `ZTPUsrDtls.txt` file in Zilog File System if it is included in the project. If the project does not include ZFS then the user name and password are maintained as a linked list. `ZTPuserDetails.c` file has supporting routines to add/delete the user name and password pairs.

Table 3. ZTP Initialization Routines

File	Routine	Description
ZTPinit_Conf.c	RZK_KernelInit()	Initializes the RZK Kernel and all of the other objects used.
	Init_DataPersistence()	Initializes the data persistence structures and sets the values of the MAC addresses, IP addresses, gateway addresses, subnet mask, and DHCP enable/disable based on the values stored in Flash Information page.
	Init_Serial0_Device()	Adds the UART0 device to the RZK device driver frame work (DDF) and also execute the corresponding initialization routine.
	Init_Serial1_Device()	Adds the UART1 device to the RZK DDF and also execute the corresponding initialization routine.
	Init_RTC_Device()	Adds the RTC device to the RZK DDF and also execute the corresponding initialization routine.
	Init_EMAC_Device()	Adds the EMAC device to the RZK driver frame work and also execute the corresponding initialization routine.
	Init_TTY_Device()	Adds the TTY device to the RZK driver frame work and also execute the corresponding initialization routine.
	nifDriverInit()	Initializes network interfaces.
ZTPinit_Conf.c	DHCP_Init()	Initiates the DHCP processes.
	CreateZTPAppThread()	ZTPAppEntry() thread is created.
	RZK_KernelStart()	all of the threads that are created will start running.

Network Configuration

ZTP features a network stack configuration in which certain components can be included – or different stack parameters can be modified – based on system requirements. These variables and macros are located in the following filepath:

`..\ZTP\Conf\`

The variables and macros for the eZ80F91, eZ80F92, eZ80F93 and eZ80L92 devices are defined in the `ZTPConfig.c` configuration file; the configuration file for the eZ80F91 Mini configuration is named `ZTPConfig_mini.c`.

Table 4 lists the ZTP core configuration and the values for the variables or macros that are a part of the configuration.

Table 4. ZTP Core Configuration

File To Modify	Variable/Macro To Modify	Valid Configuration Values
ZTP\Conf\ ZTPConfig.c	MAX_IP_RX_BUFFH	Maximum number of IP receiver buffer size. Default value: 16.
	MAX_TCP_CONNECTIONS	Maximum number of TCP connections allowed at a specific point of time. Default value: 24.
	MAX_RX_BUFSIZE	Maximum TCP internal receiver buffer size. Default value: 4096.
	MAX_TX_BUFSIZE	Maximum TCP internal transmit buffer size. Default value: 4096.
	MAX_UDP_CONNECTIONS	Maximum number of UDP connections allowed at a specific point of time. Default value: 6.

Table 4. ZTP Core Configuration (Continued)

File To Modify	Variable/Macro To Modify	Valid Configuration Values
ZTP\Conf\ ZTPConfig.c (cont'd.)	MAX_NO_ETH_IF	Maximum number of Ethernet interfaces present in the system. Default value: 1.
	MAX_NO_SERIAL_IF	Maximum number of serial interfaces present in the system. Default value: 1.
	DEFAULT_IF	Index into ifTbl for the default interface. Default value: 0.
	DEFAULT_IF_TYPE	Default interface type. Default value: ETH. Valid values: ETH or PPP.
	MAX_NUM_USERS	Maximum number of users for FTP, Telnet and the shell. Default value: 3.
	IGMP_MAX_NO_GRP	Maximum number of IGMP groups. Default value: 10.
	REAS_MAXBUFS	Maximum number of IP reassembly buffers. Default value: 3.
	REAS_MAXBUF_SIZE	Maximum size of each IP reassembly buffer. Default value: 4507.
	NUMBER_OF_TTY_DEVICES	Number of TTY devices (Telnet and the shell).
	Reboot_if_diff_IP	Reboots the system if DHCP provides a different IP address during the renewal of the IP address. Default value: FALSE. Valid value: TRUE or FALSE.
	b_use_dhcp	Using DHCP to get the IP address for the system at startup. Default value: TRUE. Valid values: TRUE or FALSE.

Table 4. ZTP Core Configuration (Continued)

File To Modify	Variable/Macro To Modify	Valid Configuration Values
ZTP\Conf\ ZTPConfig.c (cont'd)	eZ80_name	Name of the system. Default value: TRUE.
	httppath	Directory in which the HTTP server searches for the requested web pages in this directory. Default value: "/".
	g_DefaultSearchFS	If TRUE, the HTTP server searches for requested web pages in Zilog File System. If not found, the HTTP server searches in the static web page array and vice-versa. Default value: FALSE. Valid values: TRUE or FALSE.
	ztpEnPPPAtBoot	Start PPP during system bootup. Default value: FALSE. Valid values: TRUE or FALSE.
	g_ShellPrompt	The g_ShellPrompt variable is used to change the ZTP shell prompt. If Zilog File System is enabled, the following prompt name is concatenated with Zilog File System volume name and displayed in the console. Default value: ZTP. Example: When Zilog File System is enabled and the current volume name is ZTP EXTF:/, the prompt is: [ZTP EXTF:/]>.
		If Zilog File System is not enabled, the prompt is displayed as: [g_ShellPrompt]> If the prompt name is ZTP, the display is: [ZTP]>.

Table 4. ZTP Core Configuration (Continued)

File To Modify	Variable/Macro To Modify	Valid Configuration Values
ZTP\Conf\ ZTPConfig.c (cont'd)	g_TelnetPrompt	The g_TelnetPrompt variable is used to change the ZTP Telnet server's prompt. Default value: "eZ80 Telnet".
	ztpDhcpRtrs	Max number of DHCP retries. Default value: 3.
	ztpForwardIP	IP Forwarding. Default value: 0. Valid values: 0: Disable IP forwarding; 1: Enable IP forwarding.
	ztpTftpTimeout	UDP receive time-out for TFTP. Default value: 5 (in seconds).
	g_ztpsntpTimeout	UDP receive time-out for SNTP. Default value: 5 (in seconds).
	ztpRarpMaxResend	Max number of RARP retries. Default value: 3.
	ztpTcpMaxRtrs	Max TCP transmission retires before getting disconnected. Default value: 10.
	g_ShellLoginReqd	Enable user login and password facility. Default login/password: anonymous.
	ztpDnsTimeout	UDP receive time-out for DNS. Default value: 5 seconds.
	g_console_dev_to_use	DEV_SERIAL0 indicates the device to print the messages. Valid values: DEV_SERIAL0 for UART0, DEV_SERIAL1 for UART1.

Table 4. ZTP Core Configuration (Continued)

File To Modify	Variable/Macro To Modify	Valid Configuration Values
ZTP\Conf\ ZTPConfig.c (cont'd)	defaultUsrName	The defaultUsrName variable is used as the user name to log in to the ZTP shell, Telnet server, and FTP server. If Zilog File System is enabled/used, the user name and password are stored in a file. If Zilog File System is not enabled/used, the user name and password are stored in a static array. You can add a new user name and password apart from the default user name and password using the addusr command. For more information about the addusr command, see the ZTP Shell Command Reference section on page 103. Default value: anonymous.
	password	Password used to log in to the ZTP shell, Telnet server, and FTP server. Default value: anonymous.
	g_FTPCmdPort	Port Number for the FTP Control connection. This is applicable for both FTP client and server.
	g_FTPDataPort	Port Number for the FTP Data Connection. This is applicable for FTP server.
	g_TELNETDPri	Indicates the priority of the Telnet server thread. Default value is 10.
	g_TELNETDShellPri	Indicates the priority of the Telnet server thread which is created for each client connected. Default value is 15.

Table 4. ZTP Core Configuration (Continued)

File To Modify	Variable/Macro To Modify	Valid Configuration Values
ZTP\Conf\ ZTPConfig.c (cont'd)	g_FTPDPri	Indicates the priority of the FTP server thread. Default value is 10.
	g_HTTPDPri	Indicates the priority of the HTTP server thread. Default value is 10.
	g_PPPDPri	Indicates the priority of the PPP thread. Default value is 7.
	g_SYSIPDPri	Indicates the priority of the system thread, i.e., IP. Default value is 28.
	g_BOOTPDPri	Indicates the priority of the BOOTP thread. Default value is 8.
	g_BOOTPDTimPri	Indicates the priority of the BOOTP Timer thread. Default value is 10.
	g_SHELLDPri	Indicates the priority of the shell thread. Default value is 15.
	g_SNMPDPri	Indicates the priority of the SNMP Agent thread. Default value is 20.
	g_AppEntryTaskPrio	Indicates the priority of the Application thread, ZTPAppEntry() present in main.c. Default value is 16.

Common Servers

Table 5 lists the default settings for commonly-used servers. If DHCP is enabled, these settings are not effective. If DHCP is disabled, these settings must be configured in accordance with the local network settings.

Table 5. Common Servers

File To Modify	Variable/Macro To Modify	IP Address
ZTP\Conf\ZTPConfig.c	csTbl	csTbl timeserver: Time server.
		csTbl NetworkTimeServer: Network Time server.
		csTbl rfserver: RF server.
		csTbl tftpserver: TFTP server.
		csTbl dnsserver: DNS server.

Network Interfaces

Table 6 lists the default network interface settings. If DHCP is enabled, these settings are not effective. If DHCP is disabled, these settings must be configured in accordance with local network settings.

Table 6. Network Interfaces

File To Modify	Variable/ Macro To Modify	Valid Configuration Values
ZTP\Conf\ZTPConfig.c	ifTbl	ifTbl pNetDev: Handle to Network device driver for this interface.
		ifTbl ifType: Interface type. Valid values: ETH or PPP.
		ifTbl mtu: Maximum transfer unit. Default value: 1480.
		ifTbl speed: Interface speed at which the interface must operate. Valid values for: Ethernet interfaces: ETH_100 or ETH_1. PPP interface: PPP_9K, PPP_56K, or PPP_115K.
		myipDefault: IP address of this system in dotted decimal notation.
		ifTbldefaultroute: Default gateway IP address of this system in dotted decimal notation.
		ifTblsubnetmask: Subnet mask(UINT32 value).

Configuring the SHELL

ZTP features a simple shell that contains commands for displaying different system characteristics, including commands for the kernel, the stack, the Zilog File System and other system components. By default, all of these commands are included in the system. If any particular shell command is to be excluded from the application, then the respective command must be commented in the `shell_conf.c` file. Table 7 lists these shell commands.

Table 7. Shell Commands

File To Modify	Commands That Can Be Commented To Reduce Footprint
ZTP\Conf\shell_conf.c	bpool, devs, echo, exit, hang, help, kill, mem, port, sem, sleep, gettime, settime, reboot, ps.
Zilog File System commands	cd, copy, create, cwd, del, deltree, deldir, dir, format, gettime, help, md, move, ren, rendir, settime, type, vol.

The `shell_conf.c` file contains a set of nonnetwork-related commands. These commands are made available by default; therefore, you are not required to call a `shell_add_commands()` statement to make them available. However, you can add new commands to the `struct cmdent defaultcmds[]` structure file, the contents of which are listed below.

```
struct cmdent defaultcmds[] = {
#ifndef EVB_F91_MINI
 {"bpool",
TRUE,(SHELL_CMD)x_bpool,NULL,g_ShellHelpStrings[0]},
 {"devs", TRUE, (SHELL_CMD)x_devs, NULL,
g_ShellHelpStrings[1]},
 {"echo", TRUE, (SHELL_CMD)x_echo, NULL,
g_ShellHelpStrings[2]},
 {"exit", TRUE, (SHELL_CMD)x_exit, NULL,
g_ShellHelpStrings[3]},
```

```
 {"hang", TRUE, (SHELL_CMD)x_hang, NULL,
g_ShellHelpStrings[4]},
 {"help", TRUE, (SHELL_CMD)x_help, NULL,
g_ShellHelpStrings[5]},
 {"kill", TRUE, (SHELL_CMD)x_kill, NULL,
g_ShellHelpStrings[6]},
 {"mem", TRUE, (SHELL_CMD)x_mem, NULL,
g_ShellHelpStrings[7]},
 {"port", TRUE, (SHELL_CMD)x_port, NULL,
g_ShellHelpStrings[8]},
{"sem", TRUE, (SHELL_CMD)x_sem, NULL,
g_ShellHelpStrings[9] },
{"sleep", TRUE, (SHELL_CMD)x_sleep, NULL,
g_ShellHelpStrings[10]},
{"gettime", TRUE, (SHELL_CMD)x_getdatetime,NULL,
g_ShellHelpStrings[11]},
  {"settime", TRUE, (SHELL_CMD)x_setdatetime,NULL,
g_ShellHelpStrings[12]},
  #endif
 {"reboot",TRUE,(SHELL_CMD)x_reboot,NULL,g_ShellHelpSt
rings[13]},
{"ps", TRUE, (SHELL_CMD)x_ps, NULL,
g_ShellHelpStrings[14]},
 {"?", TRUE, (SHELL_CMD)x_help, NULL,
g_ShellHelpStrings[15]},
 }; /* shell commands */

/**
 * @doc The number of entries in the defaultcmds[]
 * array. If the defaultcmds array is modified,
 * ndefaultcmds must be adjusted accordingly.
 */
UINT16 ndefaultcmds=sizeof(defaultcmds)/sizeof(struct
cmdent);
/*@}*/
```

The struct_cmdent_defaultcmds[] structure also contains the g_ShellHelpStrings variable, which contains the help strings provided by the shell.

If you require shell command help, you can define a macro called HELP_REQUIRED in shell_conf.c file. By default, help strings are disabled.

```
INT8 *g_ShellHelpStrings[] = {
"Displays information about system's buffer pools.\n",
"Prints device information in the device table.\n",
"Echos text entered after the echo command to the std
out.\n",
"Suspends the shell.\n",
"Displays the set of commands that can be executed
from the shell's command prompt. When help is followed
by a command name it prints help for that specific
command.\n",
"Kills a specified process.\n",
"Prints a summary of the state of system memory.\n",
"Formats and prints information about all message
ports currently in use.\n",
"Displays information about all active semaphores.\n",
"Places the shell process to sleep for a specified
number of seconds.\n",
"Prints the current date and time to a standard
output.\n",
"Sets the current date and time to a standard
output.\n",
"Causes the operating system to begin its
initialization sequence.\n",
"Displays information about all processes in the
system.\n",
"Same as Help.\n",
}
```

The struct_cmdent_defaultcmds[] structure also contains the g_ftpClientHelpStrings variable, which contains the help strings provided by the FTP client.

During the FTP session, you can obtain command functionality assistance by entering the `help FTP <CMD>` command in the console. Within the syntax of this command, `<CMD>` represents the command in question. The FTP `help` command responds by displaying the appropriate help string.

If you do not require FTP command help, a variable named `g_ftpClientHelpStrings` can be set to NULL to reduce system overhead. For more information about FTP commands, see the ftp section on page 123.

```
CHAR *g_ftpClientHelpStrings[] = {
"set ascii transfer type",
"set binary transfer type",
"terminate ftp session and exit",
"change remote working directory",
"terminate ftp session",
"delete remote file",
"list contents of remote directory",
"receive file",
"toggle printing `#' for each buffer transferred",
"print local help information",
"change working directory on local machine",
"list contents of remote directory",
"list contents of remote directory",
"make directory on remote machine",
"mode",
"list contents of remote directory",
"connect to remote ftp",
"send one file",
"print working directory on remote machine",
"terminate ftp session and exit",
"receive file",
"remove directory on remote machine",
"remove file on remote machine",
"rename the file on remote machine",
"show remote system type",
"send new user information"
};
```

Sample Usage

The following command displays the help string for the `mkdir` command.

```
[ZTP EXTF:/]> ftp
ftp> help mkdir
make directory on remote machine
```

Configuring Text Telephony

The `tty_conf.c` file contains the number of available Text Telephony (TTY) devices. The shell can be used via any device over which a TTY driver is layered. Similarly, the Telnet server layers a TTY device over the TCP connection that is created to service the Telnet session and allow another instance of the shell to execute. The `tty` structure is listed in the following code fragment.

```
struct tty ttytab[4];

/*
 * @doc The number of TTY devices is contained in Ntty.
 * This variable should be treated as Read-Only, and
 * its declaration in tty_conf.c should not be
 * changed.*
 */

/* Don't modify this declaration of Ntty */
UINT8 Ntty = sizeof(ttytab)/sizeof(struct tty);
```

Configuring the Management Information Base

The `snmib.c` file contains the management information base (MIB), which is controlled by the SNMP Agent. The MIB is implemented as an array of `SNMPMIBData` structures (refer to the `snmpMib.h` file in the `inc` directory). There is an entry in the `g_snmpMIBInfo []` array for each leaf object in the MIB. A leaf object contains no direct descendants. There are also entries in the `g_snmpMIBInfo []` array for tables of objects. Each `g_snmpMIBInfo[]` table entry contains the object identifier that

uniquely identifies the object within the MIB, the data type of the object, a pointer to the value of the object, and a flag that indicates if the object can be modified by the SNMP `Set` primitive.

You can add this file to a project and modify the `g_snmpMIBInfo []` as appropriate for the application. For every table that is added to the `g_snmpMIBInfo []`, a corresponding entry is added to the `sn_table[]` array located in the `snmib.c` file. This secondary table contains information required by the SNMP library to properly manipulate objects within a table. In particular, each entry in the `sn_table[]` array contains the address of a user-supplied routine to implement the SNMP functions `Get` and `Set` for all objects within the table. In addition, `sn_table[]` entries contain the address of a user-supplied function to find the `Next` object within the table that is accorded an arbitrary input object identifier. Finally, the `sn_table[]` entry describes the number of fields (i.e., columns) within the table and the number of subidentifiers comprising the table index. For more information about updating the SNMP `g_snmpMIBInfo []` and `sn_table[]` arrays, see the How to Use SNMP section on page 77.

Configuring the Simple Network Management Protocol

The `snmp_conf.c` file contains user-modifiable objects within the system group of the MIB and general SNMP configuration values. Objects within the system group that can be tailored to your application include:

g_sysObjectID. This object identifier uniquely identifies this product within your organization's enterprise code.

g_sysDescr. This displayable text string describes your product.

g_sysContact. This displayable string contains the email address of the contact person in your organization responsible for managing this device.

g_sysName. This displayable string contains the assigned name of this device. Typically, this name is the fully qualified domain name of the device, such as `blackbox238.company.com`.

g_sysLocation. This displayable string identifies the physical location of the device.

g_sysServices. This 7-bit quantity identifies the set of service layers offered by the device.

In addition to these objects from the SNMP system group, the `snmp_conf.c` file contains the following variables, which can be tailored for your application:

UINT16 g_snmpMaxObjectSize. This variable represents the number of bytes of the largest SNMP object value that the application must process. For example, if you define an object within the MIB that is a 2000-byte-long octet string, the value of `g_snmpMaxObjectSize` should be set to 2000. To ensure proper operation of the SNMP library routines, this value must always be at least as large as `sizeof(struct oid)`.

INT8 g_snmpTrapTargetIP[]/g_snmpTrapTargetPort. This string of characters identifies the name of the device to which all SNMP trap messages are sent; the name can be specified as a domain name or as an IP address. By default, SNMP trap messages are sent to UDP port 162 on the target device but can be configured by changing the `g_snmpTrapTargetPort` variable.

g_snmpMaxStrSize. This variable indicates the maximum number of bytes that a display string object type variable can contain.

g_snmpEnterpriseOid. This variable is used in the cases of SNMPV2 and SNMPV3, in which the enterprise-specific trap OID must be specified.

g_snmpEnterpriseOIDLen. You must set the length of the enterprise-specific trap OID in this variable.

Note: The Authentication failure trap generation is controlled by the value of `SnmpEnableAuthenTraps` variable. You can modify this variable in your application or it can be modified via an SNMP Set operation. By default, SNMP Authentication Traps are not generated.

The application can request the generation of an enterprise-specific trap by using the `snmpGenerateTrap` API. To send an enterprise-specific trap in SNMPV2 and SNMPV3, the enterprise-specific trap OID must be set in the `g_snmpEnterpriseOid` variable located in the `snmp_conf.c` file.

Build Options for the ZDS II Environment

This section discusses the options available for building ZTP in the ZDS II environment. To configure ZTP, you must modify the build configuration within ZDS II. This configuration includes the linking of different libraries and a modification of the project settings. A number of protocols can be omitted from ZTP by including the stub library instead of the protocol library, or by commenting out the protocol initialization.

Some of the features of the ZTP core are also made optional. Therefore, these features can be disabled by including the appropriate `.obj` file.

The following features are identified in ZTP to reduce the footprint:

IP Reassembly. If this feature is disabled, the IP layer will not perform any reassembly, and any fragmented packets will be discarded.

IP Routing. If this feature is disabled, the packet which is not destined to ZTP, is not forwarded, but discarded.

Optional ICMP Messages. If this feature is disabled, all of the ICMP messages received are discarded except the ICMP Echo message.

Table 8 lists the procedure to remove/omit a protocol from ZTP.

Table 8. Remove a Protocol or Feature in the ZDSII Environment

Protocol/Feature to be Removed/ Omitted	Procedure
Telnet server	Comment out the telnet_init() command in the main.c file.
Telnet client	Comment out the telnet command in the shell_conf.c file.
SNMP	1. Comment out the snmp_init() command in the main.c file. 2. Select **Project Settings**. 3. Select the **General** category in the **Linker** tab of the **Project Settings** dialog box. 4. Enter the nosnmp.obj object file in the **Object/Library Modules** text field.
SMTP	Comment out the mail command in the shell_conf.c file.
DHCP	1. Select **Project Settings**. 2. Select the **General** category in the **Linker** tab of the **Project Settings** dialog box.
DNS	1. Select **Project Settings**. 2. Select the **General** category in the **Linker** tab of the **Project Settings** dialog box.
FTP client	Comment out the ftp command in the shell_conf.c file.
FTP server	Comment out the ftdinit() command in the main.c file.
HTTP	Comment out the http_init() function call in the main.c file.
IGMP	Comment out the hginit() function call in the main.c file.
TFTP	Comment out the tftp_get and tftp_put commands from the shell_conf.c file.
TIMED	Comment out the time_rqest() command in the main.c file.
RARP	1. Select **Project Settings**. 2. Select the **General** category in the **Linker** tab of the **Project Settings** dialog box. 3. Enter the norarp.obj object file in the **Object/Library Modules** text field.

Table 8. Remove a Protocol or Feature in the ZDS II Environment (Continued)

Protocol/Feature to be Removed/ Omitted	Procedure
IP reassembly feature	To disable this feature, 1. Select **Project Settings**. 2. Select the **General** category in the **Linker** tab of the **Project Settings** dialog box. 3. Enter the noreassembly.obj object file in the **Object/Library Modules** text field.
IP routing	1. Select **Project Settings**. 2. Select the **General** category in the **Linker** tab of the **Project Settings dialog box**. 3. Enter the `noroutepacket.obj` object file in the **Object/Library Modules** text field.
Optional ICMP features	1. Select **Project Settings**. 2. Select the **General** category in the **Linker** tab of the **Project Settings** dialog box. 3. Enter the `noicmpoptionals.obj` object file in the **Object/Library Modules** text field.

Libraries

This section lists the names of the libraries and the corresponding complementary `.obj` stub that must be included in a library project to remove the component.

ZTP Libraries

The ZTP TCP/IP core library, `ZTPCore.lib`, is located in the following path:

```
<install directory>\ZTP\lib\ZTPCore.lib
```

Common Libraries

The ZTP common library, `CommonProtoLib.lib`, contains the following application protocols:

PPP	DHCP	DNS	SMTP
SNMP	TFTP client	FTP server	FTP client
TIMED	Telnet client	Telnet server	HTTP server
IGMP	RARP	SNTP	PPPoEClient

Apart from the common protocols listed above, the `commonProtoLib_SNMPV3.lib` library, which is part of the SSL package, contains SNMPV3 code.

Stub Library

Stub and shadow libraries must be included in a project if a particular component is not required to be included for the footprint. The following stub libraries are included:

`NoRarp.obj`	Stub for RARP.
`NoSnmp.obj`	Stub for SNMP.
`NoPpp.obj`	Stub for PPP.
`NoChap.obj`	Stub for PPP CHAP.
`Noroutepacket.obj`	Stub to disable ZTP's routing feature.
`Norassembly.obj`	Stub to disable ZTP's IP reassembly feature.
`Noicmpoptionals.obj`	Stub to discard the received ICMP error messages other than an ICMP echo message.

Note: The shell and TTY components do not feature stubs. Therefore, if you are required to remove these components, no relevant function calls must be a part of the application code. Because the Telnet protocol is highly dependent on the shell and TTY components, this protocol cannot be used without shell and TTY support.

Website Libraries

The `Acclaim_Website.lib` and `Mini_Website.lib` library files contain sample website libraries that must be included in your application. There are also the `AuthAcclaim_Website.lib` and `AuthMini_Website.lib` libraries, which contain sample website libraries for HTTP digest authentication schemes.

Using ZTP

This chapter describes how to use the protocols in the Zilog TCP/IP Software Suite.

How to Use HTTP

Using ZTP's HTTP user interface primarily involves writing the application code that calls an HTTP initialization function. It also involves building web pages into your webserver using ZDS II.

Initializing HTTP

To configure ZTP as a webserver to provide web pages with Digest-MD5 authentication to any web client (browser) connected to a network, a call must be made to the `httpDigestAuth_init` function. The following code fragment shows the syntax of this `httpDigestAuth_init` function.

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

The `httpDigestAuth_init` function initializes and runs the webserver. When called, `httpDigestAuth_init` sets the default webserver processes within the webserver and connects to the TCP/IP stack to allow communication over the web. After the setup of these webserver processes is complete and the webserver is running, `httpDigestAuth_init` either returns a `SYSERR` if the function fails, or returns the TCP port number if the function is successful.

A number of sample projects are included with ZTP. For an example of how to use the `httpDigestAuth_init` function, refer to the `main.c` file in these sample projects.

To configure the a webserver to provide web pages with authentication to any web client (browser) connected to a network, a call must be made to the `httpBasicAuth_init`, which initializes the HTTP function with Basic authentication. The following code fragment shows the syntax of this `httpBasicAuth_init` function.

```
INT16 httpBasicAuth_init (const Http_Method*
http_defmethods, const struct header_rec *
httpdefheaders, webpage*website, UNIT16 portnum);
```

The `httpAuth_init` function initializes and runs the webserver. When called, `httpAuth_init` sets the default webserver processes with authentication.The function returns a SYSERR if the function fails, or returns the TCP port number if the function is successful.

A description of the `httpDigestAuth_init` function parameters are provided in the following sections.

Note: In an eZ80 webserver, only one HTTP webserver can be initialized, with the `http_init()` function, the `httpDigestAuth_init()` function, or the `httpBasicAuth_init()` function.

Defining the HTTP Method

The first parameter of the `http_init` function is `http_defmethods`, which is externally defined as an array of `http_method` structures. The definition of the `http_method` structure is found in the `http.h` *include file*, and is shown as follows:

```
typedef struct http_method {
 INT key;
 INT8 *name;
```

```
 void (*method)(Http_Request *);
}Http_Method;
```

Each element in the `http_defmethods` array maps one of the methods supported by the `http_method` structure to the requested function that implements the `method` member of `http_method`. The `name` structure member is a string that identifies the method. Collectively, the elements of the `http_defmethods` array identify the set of HTTP methods (commands) to which the webserver responds.

ZTP's HTTP server implements the following methods:

POST	SUBSCRIBE
GET	UNSUBSCRIBE
HEAD	

These methods can be overridden, or extended, by replacing the `http_defmethods` array with a user-defined array of `http_method` structures.

The following code fragment lists a number of default HTTP methods.

```
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},
};
```

For example, you can replace the `http_get` function with `http_myget` function by changing the `HTTP_GET` entry as follows:

```
HTTP_GET, "GET", http_myget
```

`http_myget` must use a prototype similar to the prototype for `http_get`, as follows:

```
http_get(Http_Request *request)
```

The ZTP `http_post` function can also be replaced by a user-defined function in a similar way. Both of these functions use the `request` pointer, which is of the `http_request` structure type. Alternatively, you can choose a custom method by adding an entry to `http_defmethods`. For example, the custom request `NEW` can be declared as follows:

```
HTTP_NEW, "NEW", http_new
```

The resulting `http_new` custom function, as well as `HTTP_NEW`, must also be declared in the `http.h` file.

Defining the HTTP Header

The second parameter of the `http_init` function is `http_defheaders`, which is externally defined by the webserver software and contains the default table of recognized headers. If the header from the client request is recognized from this list, it is passed to the HTTP method.

Similar to the `http_defmethods` parameter, the `http_defheaders` parameter can accept new entries. The `http_defheaders` parameter is a structure of type `header_rec`, and is defined in the following code fragment.

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

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},
};
```

Defining HTTP Web Pages

The third parameter of the `http_init` function is `website`, which must be defined. This parameter defines the web pages to be included in the website. Because these web pages created by the webserver are accessed by the webserver software as embedded data elements (and not from a mass storage device such as a disk drive), the web pages are built into the webserver code using ZDS II.

The `website` parameter is an array of web page structures, and is defined in the `http.h` header file. There must be an element in this array for each web page in the website. The following code fragment lists the contents of the `http.h` header file.

```
struct webpage {
 UINT8 type;          /* Whether this is a static*/
                      /* HTTP_PAGE_STATIC) or dynamic*/
                      /*(HTTP_PAGE_DYNAMIC)*/
                      /* page. */

 INT8 *path;          /*The relative path to this page*/
 INT8 mimetype;       /*The mime type to be returned */
                      /* in the MIMETYPE header.*/

 union {              /*Either a structure defining*/
 const struct staticpage spage;     /* static page */
```

```
 INT (*cgi)(void *);/*or a 'cgi' function that */
 } content;         /* generates this page */
};
```

As the above definition reveals, the `webpage` structure contains four fields: `type`, a pointer to `path`, `mimetype`, and a pointer to either a page of type `staticpage` or to the `cgi` function. These four fields are described below.

The type parameter. The first parameter that defines a web page is the `type` parameter, which must be of type `http_page_static` or `HTTP_PAGE_DYNAMIC`, indicating whether the page is a static web page or a dynamic web page.

The path parameter. The second parameter `path` is a pointer to a character string containing the relative path (including filename) to the web page. Because the file structure for web pages in ZTP is flat, the `path` parameter is simply used by ZTP as a character string to match the path (character string) from a browser URL to a pointer for a web page, CGI function, image, applet, etc. The pointer is the fourth parameter in the `webpage` structure.

The mimetype parameter. The third parameter, `mimetype`, is a pointer to a character string that defines the mime type of the web page. Examples of mime types managed by a webserver are:

- text/html
- application/octet-stream
- image/gif
- image/jpg

The Fourth Parameter. The fourth parameter is dependent upon the definition of the first `website` parameter, `type`. This fourth parameter provides a definition for either static or dynamic web pages.

Static Web Pages. If the `type` parameter is defined to be a static web page (i.e., `HTTP_PAGE_STATIC`), then the fourth parameter must be a pointer to a `staticpage` structure, which is defined in `http.h` and shown below.

```
struct staticpage
{
 UINT8 *contents;
 UINT16 size;
};
```

When web pages are built into the code using ZDS II, each web page is provided a reference to a parameter declared with a `staticpage` structure. The name of this parameter is derived from the name of the web page file, as follows:

```
filename.htm filename_htm
```

You must therefore include these external declarations in the application code by using the `filename_htm` naming convention. For more information about this naming convention, see the Building Web Pages section on page 50.

As an example, `demo_htm` is defined in the final line of the `demo_htm.c` file. This file is located in the following path:

```
..\ZTP<version>\website
```

The following code fragment lists the contents of the `demo_htm.c` file.

```
const struct staticpage demo_htm =
{(UINT8 *)demo_htm_data, sizeof(demo_htm_data)};
```

The `filename_html` structure of the web page name must be used when editing or referencing the `filename.htm` static page. The following code fragment provides an example of this structure.

```
// HTML pages
extern struct staticpage demo_htm;
extern struct staticpage htmlpost_htm;
extern struct staticpage htmlget_htm;
```

```
extern struct staticpage javaapplet_htm;
extern struct staticpage javascript_htm;
extern struct staticpage products_htm;
extern struct staticpage siteinfo_htm;
```

The parameter names in the above code correspond to the `staticpage` parameter names for static pages in the `website` array.

Note: The static pages in the `website` array can be normal static web pages or static web pages containing links to applets. If there is a link to an applet, the website must also contain the applet function. All applet functions are downloaded to the browser with the static page containing the applet links.

The applet functions in the `website` array example provided above contain the *class* extension in the path and filename, and are of the application/octet-stream `mimetype`.

Applet functions can be created by writing java classes in `.java` files. These `.java` files are built using a Java IDS, such as Sun's JDK, to generate `.class` files. The output `.class` files are then placed into the website directory before a ZDS II build. When beginning a ZDS II build, ZDS II transforms these `.class` files into `.c` files that are then built with other source files to create a downloadable output file for the webserver. For additional information, see the Building Web Pages section on page 50.

Dynamic Web Pages. If the `type` parameter is defined as a dynamic web page (`HTTP_PAGE_DYNAMIC`), then the fourth parameter must be a pointer to a common gateway interface (CGI) function. This CGI function must observe the following structure.

```
INT function_name(struct http_request *request)
```

For example, the `website` array shows an entry in the dynamic web page definition with a reference to a function named `add_cgi`. In the `website`

directory, this function is found in the `add_cgi.c` file. When a request from a client web browser asks for the `../cgi-bin/add` directory, and this request is for a dynamic web page, the webserver invokes the `add_cgi` function.

ZTP provides functions that support the writing of CGI code. These functions return responses to the client browser and are described in the the CGI Functions section on page 46.

The code fragment that follows provides an example of both static and dynamic web page entries for the `website` array. The final entry in this array must always be a NULL entry.

```
webpage website[] = {
/* 3 different ways of specifying the default web page
*/
{HTTP_PAGE_DYNAMIC, "/","text/html", (struct
staticpage*)index_cgi },
{HTTP_PAGE_DYNAMIC, "/default.htm", "text/html",
(struct staticpage*)index_cgi},
{HTTP_PAGE_DYNAMIC, "/index.htm", "text/html", (struct
staticpage*)index_cgi},

/* Specifying a dynamic web page added by the user */

{HTTP_PAGE_DYNAMIC, "/cgi-bin/add", "text/html",
(struct staticpage*)add_cgi},
{HTTP_PAGE_STATIC, "/messagerA.class", "application/
octet-stream",
&messagerA_class},
{HTTP_PAGE_STATIC, "/JavaClock.class", "application/
octet-stream",
&JavaClock_class},
{HTTP_PAGE_STATIC, "/AnalogClock.class", "application/
octet-stream",
&AnalogClock_class},
{HTTP_PAGE_STATIC, "/CustomParser.class", "application/
octet-stream",
&CustomParser_class},
```

```
{HTTP_PAGE_STATIC, "/ParamParser.class", "application/
octet-stream",
&ParamParser_class},
{HTTP_PAGE_STATIC, "/demo.htm", "text/html", &demo_htm
},
{HTTP_PAGE_STATIC, "/htmlpost.htm", "text/html",
&htmlpost_htm},
{HTTP_PAGE_STATIC, "/htmlget.htm", "text/html",
&htmlget_htm},
{HTTP_PAGE_STATIC, "/javaapplet.htm", "text/html",
&javaapplet_htm},
{HTTP_PAGE_STATIC, "/javascript.htm", "text/html",
&javascript_htm},
{HTTP_PAGE_STATIC, "/products.htm", "text/html",
&products_htm},
{HTTP_PAGE_STATIC, "/siteinfo.htm", "text/html",
&siteinfo_htm},
{HTTP_PAGE_STATIC, "/webcam.htm", "text/html",
&webcam_htm},
{HTTP_PAGE_STATIC, "/zoffices.htm", "text/html",
&zoffices_htm},
{HTTP_PAGE_STATIC, "/aqua_bar1.gif", "image/gif",
&aqua_bar1_gif},
{HTTP_PAGE_STATIC, "/ez80banner.jpg", "image/jpg",
&ez80banner_jpg},
{HTTP_PAGE_STATIC, "/ez80chip.jpg", "image/jpg",
&ez80chip_jpg},
{HTTP_PAGE_STATIC, "/ez80logo.gif", "image/gif",
&ez80logo_gif},
{HTTP_PAGE_STATIC, "/pioneer_banner.jpg", "image/jpg",
&pioneer_banner_jpg},
{HTTP_PAGE_STATIC, "/metro.gif", "image/jpg",
&metro_gif},
{HTTP_PAGE_STATIC, "/zilog.jpg", "image/jpg",
&zilog_jpg},
{HTTP_PAGE_DYNAMIC, "/cgi-bin/ml_reflector","text/
plain", (struct staticpage*){reflect_cgi}},
{HTTP_PAGE_DYNAMIC, "/cgi-bin/ml_replacer","text/
plain", (struct staticpage*){replace_cgi}},
{0, NULL, NULL, NULL}
};
```

The port Parameter. Static and dynamic web pages call the `port` parameter, which is essentially a number that is used to differentiate applications, or instances of the same application, when using the TCP/IP stack protocol. For HTTP applications, Port 80 is used.

CGI Functions

CGI functions are invoked when the HTTP application in ZTP matches the `path` parameter in the array of `webpage` structures to a pointer of the CGI function. The `request` pointer of the `http_request` structure is passed to each CGI function for this client request. The `http_request` structure contains parameters from the client request. This structure is defined in the following code fragment from the `http.h` header file.

```
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;/*Used only in HTTP
                     CRAM-MD5 Authentication*/
  INT8               buffer[HTTP_REQUEST_BUF];
  INT8               keepalive;
  } Http_Request;
```

The `http_request` structure includes two other structures, `http_hdr` and `http_params`, as shown in the following segment from the `http.h` header file.

```
typedef struct http_hdr {
 UINT8 key;
 INT8* value;
} Http_Hdr;
/**
* A key/value pair of strings.
* @name http_params
* @type typedef struct http_params
*/
struct http_params {
 UINT8* key;           /** The key, typically an http
                       /* header. */
 INT8* value;          /** The value associated with that
                       /* key. */
};
```

ZTP provides the following CGI functions:

- `http_output_reply`
- `http_find_argument`
- `_http_write`
- `http_add_header`
- `http_output_headers`

In each CGI function, the pointer to the `request` structure is used to maintain separate requests from different clients. These five HTTP functions are described below.

The `http_output_reply` function is used to return an acknowledgement to the browser that made the request. This function is structured as follows:

```
INT16 http_output_reply
(Http_Request *request, UINT16 reply);
```

The `http_output_reply` function returns a reply that is defined in the `reply` parameter to the browser that sent the request. The `reply` parameter returns an appropriate reply code from a set of reply codes that is provided in `httpd.h` header file, as the following code listing shows:

```
HTTP_200_OK
HTTP_400_BAD_REQUEST
HTTP_401_AUTHORISATION_REQUIRED
HTTP_403_FORBIDDEN
HTTP_404_NOT_FOUND
HTTP_411_LENGTH_REQUIRED
HTTP_412_PRECONDITION_FAILED
HTTP_414_REQUEST_URI_TOO_LONG
HTTP_500_INTERNAL_ERROR
HTTP_501_NOT_IMPLEMENTED
HTTP_503_SERVICE_UNAVAILABLE
```

The structure of the `CGI` function `http_find_argument` is:

```
INT8 *http_find_argument
(Http_Request *request, UINT8 *arg);
```

The above CGI function receives data from the corresponding browser. This function is used to extract parameters from the received data in the parsed browser request. In the above function, the `request` parameter is the pointer to the request structure containing the parsed request from the browser. In the `http_find_argument` function the `arg` parameter is a character string that is used to identify the parameter to be extracted from the request structure. This function returns a character string containing the value of the extracted parameter.

The structure of the `_http_write, CGI` function is:

```
INT _http_write
(Http_Request *request, INT8 *buff, INT count);
```

The `_http_write` is a macro used to return data to the browser that sent the request that invoked the CGI function.

In the _http_write macro, the buff parameter is a pointer to a buffer that stores the character string that is to be sent to the browser. The count parameter is the length of this character string. This macro is defined in the httpd.h header file.

The http_add_header and http_output_headers functions can be used in CGI functions to dynamically add headers that function as responses to browser requests. The structure of the CGI function http_add_header is:

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

The http_add_header function is used to add a header to the http_request structure that prompts a response. The header parameter is the header type that must be added, and the value parameter is a character string containing the value of the header. The default header types recognized by ZTP are provided in the httpd.h header file, and are listed in the following code fragment.

```
HTTP_HDR_ACCEPT
HTTP_HDR_CACHE_CONTROL
HTTP_HDR_CONNECTION
HTTP_HDR_CONTENT_LENGTH
HTTP_HDR_CONTENT_TYPE
HTTP_HDR_TRANSFER_ENCODING
HTTP_HDR_DATE
HTTP_HDR_LOCATION
HTTP_HDR_HOST
HTTP_HDR_SERVER
HTTP_HDR_ASK_CLIENT_AUTH
HTTP_HDR_SEND_CLIENT_AUTH
HTTP_HDR_SEND_SERVER_AUTH
```

The structure of the CGI function http_output_header is:

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

The `http_output_headers` function is used to output a text representation of all of the `httpdefheaders` contained in the `respheaders` array. The corresponding value of `respheader` from the `http_request` structure is pointed to by the `request` parameter.

Example. Assume that the CGI routine calls the following function:

```
add_header(request, HTTP_HDR_LOCATION, "Jupiter");
```

Next, assume that this routine calls the subsequent function:

```
output_headers(request)
```

As a result of these calls, the following text is added to the HTTP response:

```
Location: Jupiter\r\n
```

Building Web Pages

In ZDS II, the Project Viewer contains the directories of all source files, dependencies, and web files in the project. The web files are added to the *web files* directory. When one of the webserver demonstration project files, such as `website.ZDSProj File`, is opened, the web files directory can appear empty (as seen in the Project Viewer). If the web files directory is empty, it is because the most recent build removed the files from this directory after building its contents and placed them into a library file called `Acclaim_Website.lib`. This library can be seen in the *source files* directory of the Project Viewer.

To include web page files in the project, or to change the set of web pages in the project, the current `Acclaim_Website.lib` file must first be removed from the project.

To remove the current `Acclaim_Website.lib` file, observe the following procedure:

1. In the Project Viewer, select the `Acclaim_Website.lib` file, then select **Project → Remove From Project**.

2. Add web page files to the project by choosing **Project → Add to Project file** from the ZDS II menu bar. When the **Add files into Project** dialog box appears, navigate to the directory containing the web pages (in the demonstration example, it is the `website` directory). For **files of Type**, choose **Web file**.
3. Select the web file to add from the **Web file** list, and click **Add** (or click **Add All** to add all of the `.htm` files in the directory). The selected file(s) in the web files directory are added to the web files directory in the Project Viewer.

> **Note:** Not all web files in the website directory can be `.htm`. All files linked to a web page, including java applets and CGI functions, are to be placed in the website directory. Table 1 lists these filename extensions.

Table 1. Web Page Filename Extensions

web pages	.htm
	.html
cgi functions	.c
applet classes	.class
image files	.jpg
	.gif

After a project is built, the resulting downloaded executable file contains web pages that are appropriate to the project. As discussed previously, the files in the web files directory are removed during the build, and a new `website.lib` file appears in the source files directory. This library includes structures of the type `staticpage` for each web page, and these structures are identified with a name that is derived from the name of each web file.

HTTP is interfaced with the Zilog File System. As a result, web pages can be uploaded to the eZ80 CPU at run time using either TFTP or FTP. All web files to be uploaded to the directory must be specified by the following statement in the `ZTPConfig.c` file:

```
INT8 httppath[] = "/"
```

HTTP first searches for the requested web page in the static `website` array. If it does not find the page in the static array, it searches in the directory specified by the `INT8 httppath[]` variable.

For web files that are added to the Zilog File System, the `Content type` field in the HTTP reply is populated based upon the file extension used. A standard mapping of a file extension to its content type is maintained in a structure that is defined in the `website.c` file, which is located in the following path:

```
\ZTP\SamplePrograms\website
```

The `website.c` file is listed in the following code fragment, which also includes a definition for the `mimetypes[]` structure.

```
struct mimetype
{
 /* file extensions */
 INT8 * fileExtns;
 /* Associated mime types for the file extensions */
 INT8 * type;
};

struct mimetype mimetypes[] =
{
 {".htm","text/html"},
 {".html","text/html"},
 {".jpg","image/jpg"},
 {".class","application/octet-stream"},
 {".gif","image/gif"},
// If the file extensions does not match any then
// unknown will be used.
// This should always be at the end.
```

```
 {"unknown","unknown"}
};
```

The final entry in the `website.c` file should always be `{"unknown","unknown"}`. The structure of this file can be updated with additional file extensions and their MIME types. If this structure is updated, then the website library must be rebuilt to reflect the updated elements.

Note: The default page displayed by the `website.c` file contains a link, `Flashfile`, which links to the `flashfile.htm` HTML file. This `flashfile.htm` file is present in the `website` directory, but is not part of the `Acclaim_Website.lib` library file. The `flashfile.htm` file must be uploaded to the default HTTP path set by `INT8 httppath[]` using either FTP or TFTP. `flashfile.htm` links to the `zilog.jpg` graphic element, which is present in the `website` directory. This `zilog.jpg` file must also be uploaded to the default HTTP path.

How to Use TFTP

To transfer files to and from another host using the Trivial File Transfer Protocol (TFTP), ZTP provides two TFTP functions:

1. `tftp_put`
2. `tftp_get`

The `tftp_get` function is used to download a file from a TFTP server, and the `tftp_put` function is used to upload a file to a TFTP server. Each function establishes a separate UDP connection for the file transfer.

The prototype of the `tftp_get` function is:

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

In this prototype, `Addr` is a pointer to a character string containing the name or IP address (in decimal/dotted notation) of the TFTP server, and `filename` is the name of the file to be downloaded. The format of the filename is server OS-dependent. Upon failure, the `tftp_get` function returns 0 (zero); otherwise, `tftp_get` returns the number of bytes that are loaded into the Zilog File System.

File that are downloaded from the server using `tftp_get` are stored in the thread's current working directory (CWD). If a file exists in the CWD that contains the same name as the file that is downloaded from the server, the original file is overwritten by the new file.

The prototype of the `tftp_put` function is:

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

In this prototype, `Addr` is a pointer to a character string containing the IP address (in decimal/dotted notation) of the TFTP server, and `filename` is the name of the file to be uploaded. This file should be present in the thread's CWD. The `tftp_put` function returns the number of bytes sent when successful, and 0 (zero) upon failure.

How to Use SMTP

ZTP provides a mail function to send email messages using the Simple Mail Transfer Protocol (SMTP). This mail function sends an SMTP mail message to a specified SMTP server/port; the SMTP function then establishes a TCP connection for the mail transfer. ZTP has the option of enabling the CRAM-MD5 algorithm authentication for SMTP. To enable the CRAM-MD5 algorithm authentication, add the `smtp_conf.c` file to the ZTP project.

The prototype of the `mail` function is:

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

In this prototype, the following elements can be defined:

- `Addr` is a pointer to a character string containing the name or IP address (in decimal/dotted notation) of the SMTP server.
- `port` is the SMTP port to use; this port is normally set to 25.
- The `subject` parameter is a character string containing the *Subject:* text in the mail message.
- The `to` and `from` parameters are character strings containing the email addresses of the recipient and sender, respectively.
- The `usrname` and `passwd` parameters are the user name and password for the SMTP server to authenticate the client. These parameters are used only if SMTP digest scheme authentication is enabled. If SMTP digest scheme authentication is not enabled, these two parameters are ignored (it is advisable to set this parameter to NULL if an SMTP digest scheme authentication is not enabled).
- The `data` parameter is also a character string containing the body of the email, plus any additional headers. The data buffer should contain a mime-content type header. An example of this type of header is provided here:

  ```
  MIME-Version: 1.0\r\nContent-Type: TEXT/PLAIN;
  charset=US-ASCII\r\n\r\n
  ```

- The `error` parameter is a pointer to a buffer-pointer in which ZTP can place a text string describing the reason why the `mail` function

failed to send the message. You are responsible for allocating and freeing this buffer.

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

The `mail` function automatically prepends the *Date:*, *Subject:*, *From:*, and *To:* lines in the body of the message.

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

An example of the `mail` function is shown below:

```
status = mail
(
"SmtpServer.mycompany.com", // Destination SMTP server
25,                          // Port number
"re Thermostat Control",    // Subject
"JohnDoe@mycompany.com",     // Recipient email address
"eZ80EvalBoard@zilog.com", // Sender's email address
"username",           // User name for server to
                      //authenticate
"password",           // Password for server to
                      //authenticate email body
"MIME-Version: 1.0\r\nContent-Type: TEXT/PLAIN;" \
"charset=US-ASCII\r\n\r\n" \
"My sensors indicate the temperature in Office 506" \
"is 20 degrees above normal room temperature.",

&p_buffer,            // Buffer to contain any
                      // returned error msgs
500                   // length of Buffer
);
```

The `mail` function in ZTP works with either of the Ethernet or PPP network interfaces.

Note: An SMTP server is required. Either the domain name or the IP address of the server must be specified. Email addresses with domain names or IP addresses can also be used for the sender's and recipient's email address.

How to Use the Telnet Server

The Telnet protocol is used to initiate a remote login session. To initialize a Telnet server on the eZ80 CPU, you must call `telnet_init()`, which creates a thread for each client. The Telnet server transfers control to the shell, which executes all of the commands entered by the remote client.

To use the Telnet server and to obtain a Telnet prompt, you must enter the login name and password. The default values for the login name and the password are provided in the `ZTPConfig.c` file. Refer to the `ZTPConfig.c` file to review the values that are set for the `defaultUsrName` and `password` variables. This file is located in the following path:

```
..\ZTP\Conf\ZTPConfig.c
```

Note: Comment out the line containing the `telnet_init()` statement if a Telnet server is not required.

How to Use the Telnet Client

ZTP provides a set of functions for remote login across a network using the Telnet protocol. The following three functions allow users to open and close a Telnet session with a remote host as well as send data to this remote host over the Telnet connection.

- `TelnetOpenConnection`
- `TelnetCloseConnection`
- `TelnetSendData`

Each of these Telnet functions are described in this section.

Connecting to a Remote Host Across a Network

To open a Telnet session with a server, ZTP provides the `TelnetOpenConnection` function, which establishes a TCP connection with a specified server and sends the `ECHO` and `SUPPRESS GO AHEAD` options. A prototype of the `TelnetOpenConnection` function is shown below.

```
TELNET_RET TelnetOpenConnection( IP_ADDRESS ipAddr,
TELNET_HANDLE *telnetAppHandle, TELNETREAD telnetRead-
Callback )
```

In this prototype, the following elements can be defined:

- `ipAddr` is the IP address or name of the Telnet server.
- `telnetAppHandle` is a handle furnished by the Telnet client to the application after successfully establishing a connection.
- `telnetReadCallback` is a function pointer furnished by the application that is used by the Telnet client to notify the application as to when the data is received from the remote system.

The `TelnetOpenConnection` function returns the `TELNET_SUCCESS` value if the connection is established successfully, and returns any one of the following values if the connection fails.

`TELNET_ALREADY_CONNECTED`	Indicates that a Telnet connection already exists.
`TELNET_INVALID_ARG`	Indicates that one or more arguments are invalid.
`TELNET_LOWER_LAYER_FAILURE`	Indicates that a TCP connection failure occurred.
`TELNET_CONNECT_FAILURE`	Indicates that an unknown error occurred.

> **Note:** Comment out the line containing the `telnet_init()` statement if a Telnet server is not required.

Closing a Connection to a Remote Host

To terminate a session with a Telnet server, ZTP provides the `TelnetCloseConnection` function. This function terminates the TCP connection with the specified server and clears the connection-related information for the specific application. A prototype of the `TelnetCloseConnection` function is provided below.

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

In this prototype, `telnetAppHandle` is a handle furnished by the Telnet client after successfully establishing a connection.

The `TelnetCloseConnection` function returns `TELNET_SUCCESS` after a connection has been successfully established, and returns any one of the following values if the connection fails.

`TELNET_NO_CONNECTION`	Indicates that a 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.

Sending Data to a Remote Host

To send data to a Telnet server (executing server-end commands), ZTP provides the `TelnetSendData` function. This function sends each entered character to the server; each character is then displayed on the console when the server echoes the character. A prototype of the `TelnetSendData` function is provided below.

```
TELNET_RET TelnetSendData ( TELNET_HANDLE telnetAp-
pHandle, TELNET_DATA *telnetData, TELNET_DATA_SIZE
telnetDataSize)
```

In this prototype, the following elements can be defined:

- `telnetAppHandle` is a handle furnished by the Telnet client to the application after successfully establishing a connection
- `telnetData` is the actual data that must be sent to the server
- `telnetDataSize` represents the amount of data to be sent

The `TelnetSendData` function returns a `TELNET_SUCCESS` value if the connection is established successfully, and returns any one of the following values if the connection fails.

`TELNET_NO_CONNECTION`	Indicates that a Telnet connection is not yet established.
`TELNET_INVALID_ARG`	Indicates that one or more arguments are invalid.

`TELNET_LOWER_LAYER_FAILURE`	Indicates failure at the lower layers of the stack.

How to Use the FTP Server

FTP is the user interface to the internet-standard File Transfer Protocol. The program allows you to transfer files to and from a remote network site.

To begin using ZTP's FTP service, call the `ftpdinit()` function from the application entry point, which is the `main` function. This function call initializes and sets the FTP server into a listening mode.

To use the FTP server, enter a login name and password. The default values for the login name and the password are provided in the `ZTPConfig.c` file. Refer to the `ZTPConfig.c` file to review the values that are set for the `defaultUsrName` and `password` variables. This file is located in the following path:

```
..\ZTP\Conf\ZTPConfig.c
```

You can add a new FTP user name and password using the `addusr` shell command and delete an existing FTP user name and password using the `deleteusr` shell command.

ZTP's FTP server listens at the standard FTP port 21, and supports the standard set of commands, which are listed in Table 2.

Table 2. Standard Set of Commands Supported by the FTP Server

ls	dir	mkdir	rmdir	chdir
put	get	delete	pwd	system
user	password	bin	ascii	bye

The prototype for the FTP daemon entry point is:

```
void ftpdinit (void)
```

How to Use the FTP Client

To log in to a remote site and conduct a file transfer using the File Transfer Protocol, ZTP provides the following set of functions, which are each described in this section.

- `ftp_connect`
- `do_programatic_login`
- `do_a_ftp_command`

Connecting to an FTP Server

Use the `ftp_connect` function to connect to an appropriate FTP server running on `FTP_PORT`. The prototype of the `ftp_connect` function is shown below.

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

In this prototype, the following elements can be defined:

- `server_name` is the IP address of the FTP server (in dotted notation)
- `server_port` is the port number of the FTP server
- `stdout` represents the console

This function returns a 0 if a connection has been established successfully, and returns a negative value if the connection fails.

Log In With a Username and Password

To log in to an FTP server with a username and a password, use the following function:

`do_programatic_login`

A prototype of the `do_programatic_login` function is shown below.

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

In this prototype, the following elements can be defined:

- `stdin` represents a console
- `stdout` represents a console
- `username` is the username
- `passwd` is the password

The `do_programatic_login` function returns a 1 if a connection has been successfully established, and returns 0 if the connection fails.

Issuing FTP Commands

FTP commands can be provided as arguments to the `do_a_ftp_command` function. These command names must be provided as an array of strings.

A prototype of the `do_a_ftp_command` function is provided below.

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

In this prototype, the following elements can be defined:

- `device` represents a console

- `nargs` is the number of arguments that the command expects
- `args` represents the arguments that support the command

The `do_a_ftp_command` function returns a 0 if a connection has been established successfully, and returns a negative value if the connection fails.

How to Use BOOTP

BOOTP is used to carry DHCP extensions that allow ZTP to obtain dynamic IP parameters from a DHCP server. If the network only contains a BOOTP server, then the only IP parameter that is configured is the IP address. For more information about using DHCP, see the How to Use DHCP section on page 64.

> **Note:** When BOOTP receives IP address information, it updates the IP addresses in the interface table.

How to Use DHCP

Zilog's TCP/IP protocol stack can be configured to use statically-assigned IP parameters, or to obtain these parameters dynamically from a DHCP server. The default IP parameters are contained in the `ifTbl` and `csTbl` structures that are contained in the `ZTPConfig.c` file, which is located in the following path:

```
..\ZTP\Conf\ZTPConfig.c
```

DHCP usage is controlled by the value of the `b_use_dhcp` flag in the `ZTPConfig.c` file. When this flag is set to `TRUE`, ZTP uses DHCP to

attempt to update the default IP parameters specified in the ifTbl and csTbl structures during system initialization. If a server cannot be found, ZTP uses the default values contained in the ifTbl and csTbl structures. When b_use_dhcp is set to FALSE, ZTP uses the values in the ifTbl and csTbl structures to determine its IP parameters.

When ZTP attempts to access a DHCP server, it starts a timer. If this timer times out, it either repeats the attempt or defaults to the static IP address. ZTP attempts to access the DHCP server a certain number of times, the frequency of which are specified by the #define DHCP_RETRIES parameter in the ZTPConfig.c file. The number of retries should be set according to the amount of congestion that is expected on the network.

There is also a UINT8 Reboot_if_diff_IP parameter in the ZTPConfig.c file. If IP parameters are obtained from the DHCP server, this IP address lease time parameter specifies that the assigned IP address is valid for a set amount of time. After a lapse of this lease time, the IP address must be renewed. During this renewal, if the IP address is different from the one initially obtained, then there are two options, which are listed below.

- The stack must be rebooted, because any applications running are still using the old IP address
- DHCP must inform any applications running that the IP address has changed

The second option is currently not implemented. With the first option, if the UINT8 Reboot_if_diff_IP configuration parameter in the ZTPConfig.c file is set to TRUE, then, under the conditions explained above, the stack is rebooted. If set to FALSE, a warning message is printed on the console.

The INT8 eZ80_name[] parameter in the ZTPConfig.c file is used to fill the host name option (option code 12) in the DHCP packets. The maximum size of the host name is 64 bytes.

How to Use DNS

To resolve a host name to an IP address using DNS, ZTP provides the following function:

```
UINT32 name2ip(INT8 *name);
```

In this function, `name` is a character string containing the host name or URL.

The `name2ip` function is defined in the `network.h` header file. This function accesses DNS directly using a DNS-formatted message in a UDP datagram with the DNS IP address acquired from the boot record. When `name2ip` receives the IP address from DNS, it is returned as an `UINT32` variable. If the name cannot be resolved, `name2ip` returns `SYSERR`. This error occurs in the following ways:

- The name server's IP address is unknown
- The NSame server is down
- The webserver is not attached to the network
- The gateway is down
- The user enters the name incorrectly

Therefore, if you resolve *www.zilog.com* into its associated IP address, `66.238.115.245`, the following code is added to the project:

```
UINT32 zilog_ip_address;
zilog_ip_address = name2ip( "www.zilog.com" );
```

You must verify that the returned address is not `SYSERR`.

How to Use IGMP

IGMP is initialized at startup by the following function:

```
INT16 hginit(UINT8 iface)
```

In this function, `iface` is the ethernet interface number.

Two functions are available that enable or disable specified group IP addresses at which the webserver responds. These functions are:

```
INT hgjoin(UINT8 ifnum, UINT32 ipa, UINT8 ttl)
INT16 hgleave(UINT8 ifnum, UINT32 ipa)
```

In these functions, `ifnum` is the interface index, which should always be set to the number of the primary Ethernet interface.

The `ipa` parameter is the multicast IP address to be added or removed from the host. It is of type `UINT32`. Each webserver host can provide as many multicast IP addresses as is specified in the following parameter:

```
#define IGMP_MAX_NO_GRP
```

This define parameter can be found in the `ZTPConfig.c` file. To add more multicast IP addresses, call `hgjoin` for each multicast address. To remove multicast IP addresses, call `hgleave` with the IP address specified in `ipa`.

Only multicast addresses in the range `224.0.0.2` to `239.255.255.255` must be used. Broadcast IP addresses cannot be used with this protocol.

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 the multicast packet can pass.

Note: Multicasting with a webserver on the network is limited to the local network if a multicast router is not present on the local network.

When multicasting is set up between hosts on a network, an application using UDP on a webserver host with ZTP must check the received messages from the UDP link to determine if the content is correct for a particular IP address, because other groups can be using the same multicast address.

All IP multicast addresses are Class D IPv4 addresses (i.e., the first four bits are `1110`). In dotted decimal notation, the range is `224.0.0.0` to `239.255.255.255`. Address `224.0.0.1` is reserved for the IGMP protocol. Address `224.0.0.2` is reserved for the multicast routers group. In general, addresses in the range `224.0.0.0` to `224.0.0.255` are reserved for routing protocols. IP multicast addresses are used only as destination addresses.

All IP multicast addresses map into the lower half of the following Ethernet address block:

```
01 00 5E xx xx xx
```

Therefore, only the last 23 bits of the 32-bit IP multicast address are mapped to an Ethernet multicast address. For example, both of the `224.0.10.10` and `230.128.10.10` addresses use the Ethernet multicast address `01 00 5E 00 0A 0A`.

How to Use TIMEP

The TIMEP protocol implemented in ZTP is compliant with RFC 738.

Requesting the Time

The `time_rqest()` function requests the current time from a TIMEP server. The default IP address for the TIMEP server is specified in the `csTbl` structure that is contained in the `ZTPConfig.c` file, which is located in the following path:

```
..\ZTP\Conf\ZTPConfig.c
```

When the time is received from the server, the time of day is updated in the eZ80 CPU's real-time clock.

How to Use PPP

The Point-to-Point Protocol (PPP) is designed primarily to provide a mechanism for connecting to a TCP/IP network via a serial line (HDLC) or an Ethernet link (PPPoE). ZTP provides PPP as a second network interface with a separate IP address. PPP HDLC supports both a client and a server, but PPPoE support is only client-based.

The following two demo commands are included by the `void ztpAddPPPShellCmds(void)` function call.

pppstart. Initializes and starts the PPP interface.

pppstop. Terminates the PPP connection.

The following two APIs are provided to start and terminate a PPP connection:

INT16 ztpPPPInit(void). This function initializes and starts the PPP interface.

INT16 ztpPPPStop(void). This function terminates the PPP connection.

After a connection is established, the negotiated network parameters are stored in a global variable named `PppStatus` of type `PppGlobal`. A definition of this `PppGlobal` structure is shown below.

```
struct PppGlobal {
  PppPhase          Phase;
  UINT32            MyIP;
  UINT32            PeerIP;
  UINT32            PppPriDns;
```

```
    UINT32              PppSecDns;
    UINT32              PppPriNbns;
    UINT32              PppSecNbns;
    struct              netif *PppNif;
    struct              If *PppIf;
    UINT16              IsPPPInitialized;
};
```

Table 3 describes each member of the `PppGlobal` structure.

Table 3. PppGlobal Structure Members

Member Name	Description
Phase	Indicates the current state of the PPP link. See Table 4 for a description of each identifier in the PppPhase enumeration.
MyIP	IP addresses of the local PPP interface after successful PPP connection.
PeerIP	IP addresses of the remote PPP interface after successful PPP connection.
PppPriDns	Primary DNS server IP address for the PPP interface.
PppSecDns	Secondary DNS server IP address for the PPP interface.
PppPriNbns	Primary NBNS server IP address for the PPP interface.
PppSecNbns	Secondary NBNS server IP address for the PPP interface.
PppNif	Pointer to the struct netif of PPP interface.
PppIf	Pointer to the struct If of PPP interface.
IsPPPInitialized	Flag which indicates if PPP is initialized.

Table 4 describes the `PppPhase` identifiers.

Table 4. PppPhase Identifiers

Identifier Name	Description
PPP_DEAD_PHASE	Indicates the dead phase, i.e., the default phase before ztpPPPInit() has been called.
PPP_LINK_PHASE	The lower layer (modem initialization in case of PPP HDLC and completion of Discovery process in case of PPPoE) has been successfully. LCP negotiations will start.
PPP_AUTHENTICATE_PHASE	If authentication has been enabled in this phase, then the user credentials are verified (PAC or CHAP).
PPP_IPCP_PHASE	IPCP negotiations begins in this phase.
PPP_NETWORK_PHASE	After all of the IPCP options are successfully negotiated PPP will be in this phase. This phase indicates successful PPP negotiations and that the PPP interface is up.
PPP_TERMINATE_PHASE	PPP is set to this phase if it receives a LCP terminate request from the peer or if ztpPPPStop( ) has been called by the application.

How to Use the HTTPS Server

The `SSL.lib` file contains an HTTPS server that can be used to serve encrypted web pages to client browsers. The server is initialized by calling the `https_init` API. This API accepts the same number and type of parameters as the standard HTTP server API. For example, to initialize the standard nonsecure HTTP server in ZTP, the following command is used:

```
http_init(http_defmethods,httpdefheaders,website,80);
```

To initialize the secure HTTPS server, the following command is used:

```
https_init(http_defmethods,httpdefheaders,web-
site,443);
```

It is possible to have both the secure and nonsecure webservers running at the same time; however, these two webservers must operate through different ports. The port number typically used for nonsecure HTTP servers is port 80; for secure HTTP servers (HTTP over SSL or HTTPS), the port number typically used is port 443.

A more thorough review of HTTP servers can provide a foundation. To understand more about HTTPS server functionality, see the How to Use HTTP section on page 36.

Limitations of the HTTPS Server

Three limitations of the ZTP HTTPS server may exist, as described below.

- It may become necessary to configure the client browser to support the SSL2 protocol. Refer to the documentation supporting your browser for details about how to perform this operation. For example, in Microsoft Internet Explorer, navigate to the **Tools** → **Internet Options** menu, click the **Advanced** tab, and ensure that the SSL v2.0 protocol is selected. Figure 1 displays a screen shot of the **Internet Options** dialog box.

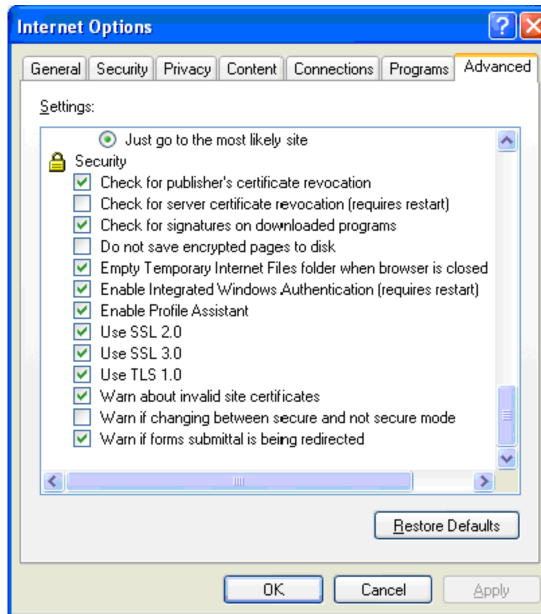

Figure 1. Internet Options Dialog Box

- When using the sample certificate with the HTTPS server in the SSLDemo project, be aware that client browsers such as Microsoft Internet Explorer may generate warning messages while processing the sample certificate (an example is shown in Figure 2). The first warning typically encountered occurs because the certificate was self-signed; therefore a trusted root certificate authority (CA) does not exist in the certificate chain. The second warning is generated because the certificate's subject (the distinguished name of identity of the SSL server) does not match the server's website or IP address. These warnings are not generated when the CA issues a valid certificate, in which the CN value matches the server's name or IP address.

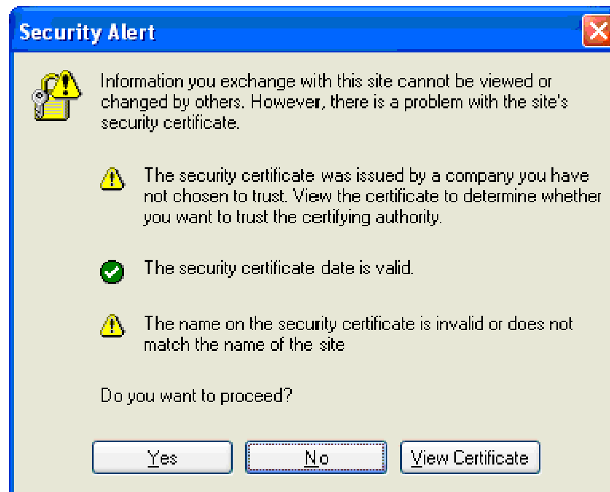

Figure 2. Security Alert Warning Message

- A single SSL certificate is supported in the current SSL layer implementation.

How to Use the Shell

ZTP provides a shell that allows you to interact with the system by using commands that are transferred via a remote terminal. This remote terminal can be a PC running a terminal program, such as HyperTerminal via a serial connection, or to a Telnet terminal via an Ethernet connection. For more information about Telnet, see the How to Use the Telnet Server section on page 57.

Shell code is included in the `CommoProtoLib.lib` library. To include the shell in your application, call the `shell_init` function in the code, as follows:

```
INT16 shell_init(RZK_DEVICE_CB_t * fd)
```

In the call statement above, `fd` is the device ID of the device over which the shell is to operate. The following code example shows how to enable the shell over the UART0 console.

```
RZK_DEVICE_CB_t *TTYDevID;
struct devCap *devSerial;

devSerial = (struct devCap *)malloc (sizeof(struct
devCap));
devSerial->devHdl = (VOID*)CONSOLE;
devSerial->devType = 0;
if
((TTYDevID=RZKDevOpen("TTYM",(RZK_DEV_MODE_t*)
devSerial))==(RZK_DEVICE_CB_t*)SYSERR)
{
 return SYSERR;
}
shell_init(TTYDevID);
```

The set of commands available through the shell is configurable by modifying the `defaultcmds` array in the `shell_conf.c` file, which is discussed in the the Configuring the SHELL section on page 24. These shell commands are described in the the ZTP Shell Command Reference section on page 103. Shell commands can also be added at run time by using the `shell_add_commands` function (from `shell.h`), as follows.

```
void shell_add_commands( struct cmdent *cmds, UINT16
ncmds, UINT8 bShType);
```

The `shell_add_commands` function contains three parameters:

- `cmds`
- `ncmds`
- `commandType`

cmds is a pointer to an array of cmdent structures containing information required to add a command. The cmdent structure is the same structure used to configure the set of shell commands in shell_conf.c. The information contained in the cmdent structures includes the following elements:

cmdnam. The character string representing the name of the command.

cbuiltin. This field should always be set to TRUE for forward compatibility.

cproc. The function to be called for the cmdnam command in the shell. The available shell functions that can be identified with the cproc parameter are shown in the shell.h file; they are identified by a name with a prefix of x_.

cnext. This field should always be set to NULL for forward compatibility.

helpDesc. The character string representing the help description for each command.

ncmds. Represents the number of commands to be added to the shell.

commandType. Represents the type of the command added to the shell. There are two values defined for this parameter: SHELL_ZTP identifies the command as a ZTP command, and SHELL_TELNET identifies the command as a Telnet command.

Both SHELL_ZTP and SHELL_TELNET are defined in shell.h.

The following code example shows how shell commands can be added using the shell_add_commands function.

```
struct cmdent *mycmds;
mycmds = getmem( sizeof(struct cmdent) * 1);

/* Set up mail command */
mycmds[0].cmdnam = "mail";
mycmds[0].cbuiltin = TRUE;
mycmds[0].cproc = (SHELL_CMD)x_mail;
```

```
mycmds[0].cnext=(struct cmdent *)NULL;
mycmds[0].helpDesc="Sends mail using SMTP\n";

/* Add SMTP demo command */
shell_add_commands(mycmds, 1, SHELL_ZTP);
mycmds = getmem( sizeof(struct cmdent) * 1);

/* Telnet quit command */
mycmds[0].cmdnam = "quit";
mycmds[0].cbuiltin = TRUE;
mycmds[0].cproc = (SHELL_CMD)Shell_TelnetQuit;
mycmds[0].helpDesc = "Quits Telnet\n";
mycmds[0].cnext=(struct cmdent *)NULL;

shell_add_commands(mycmds, 0, SHELL_TELNET);
```

A prepackaged number of network commands can be added as a set, as shown in the following code fragment.

```
/* Make the network-related shell commands available
/* to all shells */
shell_add_commands(netcmds, nnetcmds,SHELL_ZTP);
netcmds and nnetcmds are externals, as declared in
shell.h.
```

To use the shell, log in to the shell with the configured user name and password located in the `ZTPConfig.c` file. Refer to the `ZTPConfig.c` file to review the values that are set for the `defaultUsrName` and `password` variables. This file is located in the following path:

`..\ZTP\Conf\ZTPConfig.c`

How to Use SNMP

ZTP supports the SNMPv1, SNMPv2 and SNMPv3 versions of the Simple Network Management Protocol. To enable the SNMPv1 agent included in ZTP, call the `ztpSnmpV1Init()` API from the `main()` rou-

tine. This call activates the SNMPv1 agent that responds to SNMP requests for objects in the default MIB.

To enable SNMPv2, call the `ztpSnmpV2Init()` API from the `main()` routine. The SNMP v3 protocol is a separate package bundled along with SSL. To work with SNMP v3, install ZTP, then install the SSL package.

> **Note:** With an eZ80 CPU, you can run only one SNMP agent at a time. Therefore, only one of the APIs, `ztpSnmpV1Init()`, `ztpSnmpV2Init()` or `ztpSnmpV3Init()`, must be called.

The SNMP implementation in ZTP includes objects within the following MIB-II groups:

- System
- Interfaces
- Address translation
- IP
- ICMP
- TCP
- UDP
- SNMP

The following sections guide you through how to add objects unique to the application in the MIB.

Object Names

In the ZTP implementation, object identifiers are described by the following OID structure.

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

Therefore, the longest object identifier that ZTP can support must not be greater than 32 subidentifiers. The following code fragment provides an example.

```
1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.
22.23.24.25.26.27.28.29.30.31.32
```

Because each subidentifier is a 16-bit value, the largest value of any subidentifier is restricted to being `65535`.

For example, to declare an object identifier in your project that contains the value `1.2.3.4`, the following definition is used:

```
OID SampleOid = { {1,2,3,4}, 4 };
```

Note: Because of the popularity of SNMP, it is likely that the IANA's Private Enterprise codes will soon require at least 24 bits to contain new assignments. Therefore, a future version of ZTP will redefine object subidentifiers to be of the *unsigned int* (24-bit) type, or possibly the *UINT32* (32-bit) type.

Object Types

Objects within the SNMP MIB are restricted to use a subset of the primitive data types defined within the ASN.1 standards, such as *integers*, *octet strings*, and *object identifiers*. In addition, objects can be defined using SNMP-specific data types such as *IP address*, *counter*, *gauge*, and *timeticks* – each of which are defined using ASN.1 primitive data types. SNMP also allows these primitive data types to be aggregated to create lists or tables using the ASN.1 *sequence* constructor type. By using a

restricted set of data types, SNMP management tools from one vendor can interoperate with agents from a different vendor as they use ASN.1.

Table 5 lists the ASN.1 primitive data types supported by ZTP.

Table 5. ASN.1-Supported Primitive Data Types

ASN.1 Data Type	ZTP Data Type	Description
Integer	ASN1_INT	An arbitrarily long signed number.
Octet String	ASN1_OCTSTR	An arbitrarily long string of octets (bytes).
Object Identifier	ASN1_OBJID	An object Identifier used to name objects within the MIB.
NULL	ASN1_NULL	An object that does not contain a value is said to be of type NULL.

Note: As an implementation limit, the maximum size of an integer and octet string is constrained by the value that is assigned to the variable `g_snmpMaxObjectSize` in `snmp_conf.c`. The default value of this variable is `400`. Therefore, the default maximum length for any integer or octet string manipulated by ZTP is 400 bytes. If such large objects are not required, you can reduce the value of the `g_snmpMaxObjectSize` variable. Ensure that `g_snmpMaxObjectSize` does not exceed 2048 bytes, because the receive or transmit packet size is limited to 4000 bytes.

ZTP also supports the following SNMP-specific object types:

DisplayString. `SN_DISPLAY_STRING` is a 255-byte octet string containing text characters. ZTP allows you to change the maximum length of display strings.

Note: If you are using ZDS II, review the `snmp_conf.c` file, which is located in the following path:

```
<ZTP Install Directory>\ZTP\Conf
```

IpAddress. `ASN1_IPADDR` is a 4-byte octet string used to contain an IP address.

Counter. `ASN1_COUNTER` is a 32-bit monotonically-increasing unsigned integer that wraps from `FFFFFFFFh` to `00000000h`.

Gauge. `ASN1_GAUGE` is a 32-bit unsigned integer that latches when it reaches `FFFFFFFFh`.

PhysAddress. `SN_PHYS_ADDR` is a 6-byte octet string that contains a 48-bit MAC address.

TimeTicks. `ASN1_TIMETICKS` is a 32-bit unsigned integer that counts time in units of 10 ms since the beginning of a defined epoch.

SNMP Objects

Every SNMP object contains a name, a type, and a value. Object names are specified as Object Identifiers of type `SnmpObj` (as described in the Object Names section on page 78; the set of permissible object types is described in the the Object Types section on page 79). You can assign and/or update an object value in the user's SNMP application. Review the `snmpv1.h` file, which is located in the following path:

```
\ZTP\Inc\
```

Before discussing object updates, the definition of an SNMP object in ZTP must be examined, as the following code fragment shows.

```
typedef struct snmpObj
{
  UINT16            *objId ;
```

```
  UINT8                 objIdLen ;
  UINT8                 objValType;
  SNMPObjValue          objVal;
}SNMPObj ;
```

In this code fragment, an `SNMPObjValue` structure is defined as:

```
typedef union snmpObjVal
{
  void                  *pData;
  OID                   *pOid;
  SNMPDisplayStr        *pDescr;
  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;
```

The code above shows that an SNMP value is nothing more than a pointer to an arbitrary block of data, the meaning of which depends on the objValType member specified in the SNMPObj structure. For example, suppose your company produced two types of widgets: Type1 and Type2, and that you need to uniquely identify variables within the Type2 widget that contain the license key and serial number. Using named labels, the SNMP representation of these variables is:

```
private.enterprises.your_company.type2.license_key
private.enterprises.your_company.type2.serial_number
```

Before you can access these variables using SNMP, an administrator in your company must assign a (unique) numeric value to these Type1 and Type2 widgets, as well as (unique) values to the license key and serial number variables. In this example, Type1 widgets are assigned a numeric value of 1, Type2 widgets are assigned a value of 2, the license key a value of 10, and the serial number a value of 20. If your company's IANA-assigned enterprise ID is 22222, then the numeric representation of these variables is:

```
4.1.22222.2.10
4.1.22222.2.20
```

Suppose the program that maintains the serial number stores its value in a 32-bit variable named `SerialNumber`. The code fragment below, then, illustrates how you can construct an SNMP object to describe the serial number variable in the Type2 widget produced by your company:

```
UINT32  SerialNumber = 0x11223344;
UINT16  SerNumOid[]  = {4, 1, 22222, 2, 20, 0};
SNMPObj SerNumObj[]  = {&SerNumOid, 6, SN_UINT32,
                        &SerialNumber};
```

In the above code, notice that the standard `iso.org.dod.internet` prefix of `1.3.6.1` is missing from the beginning of the object identifier. It is not present because the SNMP library automatically prepends this prefix to all objects within the `g_snmpMIBInfo` that do not begin with a subidentifier of 1. This prepend restricts you from using objects within the `iso.org.dod.internet` branch of the set of all possible object identifiers. Additionally, to define objects within the `iso.org.dod.internet.directory` tree, you must fully specify all of the subidentifiers to the root.

To see how objects with predefined sizes are defined, simply use a `Type` field that matches the type of your variable, and set the object value to reference the variable. To define octet strings and integers of arbitrary length, wrap your variable in an `SNMPDisplayStr` structure. This `SNMPDisplayStr` structure is defined as:

```
typedef struct sn_descr_s
```

```
{
  void *              pData;
  UINT16              Length;
} SNMPDisplayStr;
```

In this structure, the `Length` member indicates the number of bytes of data currently required to contain the value of this object; the maximum length is limited to 255 bytes. For example, the code fragment below defines an integer – which can be up to 16 bytes long – as the eighth object within the `blackbox` group. The current value of this integer is `112233445566h`.

```
INT8 Data16[16] = {0x66,0x55,0x44,0x33,0x22,0x11};
SNMPDisplayStr Data16Descr =
{ Data16, 6 };

UINT8 snmpData[] = {{4,1,22222,115,8,0} ;

SNMPObj SNObject_for_Data16 =
{ & snmpData , 6 , SN_DISPLAY_STR, & Data16Descr };
```

Adding Objects to the MIB

After examining how to define SNMP objects, it is time to add an object to the MIB. In ZTP, the implementation of the MIB is contained within the `g_snmpMIBInfo []` array. Each entry in the `g_snmpMIBInfo []` array is of type `SNMPMIBData`, as shown in the following code fragment.

```
/** MIB structure **/
typedef struct snmpMIBs
{
 /* MIB object */
 SNMPObj mi_obj;
 /** Read/write status of the MIB **/
 BOOL mi_writable;
 /** Function to implement the get/set status of the
 /** MIB **/
 INT16 ( *mi_func )
 ( SNMPObjLs *,
```

```
   struct snmpMIBs *, UINT8
 ) ;
/** pointer to the next MIB **/
 struct snmpMIBs *mi_next;
}SNMPMIBData;
```

The different structure members of this MIB are described below.

mi_obj. An SNMP object, as described in the previous section.

mi_writable. A Boolean flag that, if set to `TRUE`, informs the SNMP library that the value of this object can be modified using the SNMP `Set` primitive.

> **Note:** Any object can be the target of an SNMP `Get Next` primitive; however, only leaf objects can be specified as the targets of a `Get` or `Set` request.

The `mi_func` structure member identifies the address of a routine that the SNMP library uses to perform `Get`, `Get Next` and `Set` requests on the object. The SNMP library contains a default routine called `snleaf` that manipulates all leaf variables in the MIB. Similarly, the library contains a routine called `sntable` that parses requests within tables. Unless supplying your own routine to parse tables and leaves is preferred, this structure member should always be specified as either `snleaf` or `sntable` (or `NULL` for aggregate objects).

The `mi_next` structure member should always be specified as `NULL` in the `mib[]` array. The system determines this value at run time and updates this field as required.

As an example of how to add entries to the MIB, consider the following declaration of the system group in the `g_snmpMIBInfo []` array in the `snmib.c` file:

```
const UINT16 g_snmpMgmt[] = {2};
const UINT16 g_snmpMib2[] = {2,1} ;
```

```
const UINT16 g_snmpSystem[] = {2,1,1} ;
const UINT16 g_snmpSystemDecsr[] = {2,1,1,1,0} ;
const UINT16 g_snmpSysObjectID[] = {2,1,1,2,0} ;
const UINT16 g_snmpSysUpTime[] = {2,1,1,3,0};
const UINT16 g_snmpSysContact[]= {2,1,1,4,0} ;
const UINT16 g_snmpSysName[] = {2,1,1,5,0} ;

SNMPMIBData g_snmpMIBInfo[] = {
{ g_snmpMgmt, 1, T_AGGREGATE, NULL, READ_ONLY, NULL,
NULL },
{ g_snmpMib2, 2, T_AGGREGATE, NULL, READ_ONLY, NULL,
NULL },

// System Group
//system.
{ g_snmpSystem, 3, T_AGGREGATE, NULL, READ_ONLY, NULL,
NULL },
//system.sysDescr,
{ g_snmpSystemDecsr, 5, SN_DISPLAY_STR, &g_sysDescr,
READ_ONLY, snleaf, NULL },
{ g_snmpSysObjectID, 5, ASN1_OBJID, &g_sysObjectID,
READ_ONLY, snleaf, NULL },
//system.sysUpTime
{ g_snmpSysUpTime, 5, ASN1_TIMETICKS, &SysUpTime,
READ_ONLY, snleaf, NULL },
//system.sysContact
{ g_snmpSysContact,5, SN_DISPLAY_STR,
&g_sysContact,READ_WRITE, snleaf, NULL },
//system.sysName
{ g_snmpSysName, 5,
SN_DISPLAY_STR,&g_sysName,READ_WRITE, snleaf, NULL }
};
```

The first element in the system group is an aggregate identifier for the group itself. Aggregate objects are not accessible using the `Get`, `Get Next` or `Set` SNMP primitives. However, their use in the `g_snmpMIBInfo []` array is required to allow the library's parsing routines to properly traverse the tree of objects in the MIB.

There are seven objects in the system group. Each of these objects is a leaf node, and the `mi_func` structure member for these entries is `snleaf`. Because each leaf entry is a child of the system aggregate object, the `mi_prefix` structure member names are all specified as *system*. Therefore, the text names and completely-specified corresponding object identifier for each of the entries in the system group are:

```
system.sysDescr     1.3.6.1.2.1.1.1.0
system.sysObjectID  1.3.6.1.2.1.1.2.0
system.sysUpTime    1.3.6.1.2.1.1.3.0
system.sysContact   1.3.6.1.2.1.1.4.0
system.sysName      1.3.6.1.2.1.1.5.0
system.sysLocation  1.3.6.1.2.1.1.6.0
system.sysServices  1.3.6.1.2.1.1.7.0
```

Observe that a zero is appended as the final subidentifier for all leaf objects. This zero is required by SNMP to uniquely identify the instance of the indicated object. Objects within tables are uniquely identified by an index that spans one or more subidentifiers in the object identifier (these tables are discussed in the next section).

Finally, notice that the `mi_writable` structure member is set to `TRUE` for `sysContact`, `sysName` and `sysLocation`. As a result, remote SNMP management entities are able to modify the values of these objects by using the `Set` SNMP primitive.

Using SNMP to Manipulate Leaf Objects in the MIB

After adding leaf objects to the MIB, ZTP's SNMP agent must manipulate these leaf objects by calling the `ztpSnmpV1Init()` function from within your `main()` routine. The `snleaf` function implemented in the library automatically processes all `Get`, `Get Next` and `Set` requests received from remote management entities for the objects added to the `g_snmpMIBInfo []` array.

Note: It is beyond the scope of this manual to describe how remote SNMP management application programs operate. Consult the technical documentation provided with your SNMP management application for information about how to perform `Get`, `Get Next` and `Set` SNMP requests.

Working with Tables

SNMP can also be used to manipulate tables of objects. The table itself is an aggregate object, and therefore not accessible; i.e., a table object identifier cannot be used as the target of a `Get` or `Set` operation. Only instances of objects created within the table are accessible when using `Get` and `Set`.

Before describing how tables are manipulated in ZTP, it is necessary to provide an understanding of the relationship between object identifiers and object instances within the table. A table is a list of rows that contain one or more columns. To access a particular item in the table, you must know which column and which row contain the item of interest. Within the SNMP protocol, tables are lists of objects that pertain to some entity. Each column in the table describes an attribute of the entity and each entity identifies the row of interest within the table.

Because all SNMP objects are named using object identifiers, an instance of an object in a table can only be accessed if its row and column information are included as part of the object identifier. Recall that for leaf nodes, accessing an instance of an object is quite simple: if a leaf object in the MIB has a name of *x*, then the object identifier of the single instance of that object is `x.0`. However, for tables, a slightly more complex naming convention is used. The generic form of an object identifier used to access an instance of an item in a table is:

```
TableID.TableEntry.Colum.Row or The TableID
```

This object identifier identifies the location of the root of the table in the hierarchy of objects. The `TableEntry` subidentifier is typically the only

child identifier of the TableID (i.e., {TableID 1}), and must be included in the name of every instance of an object located within the table. The Column subidentifier indicates the attribute of interest within the TableEntry, and the Row subidentifier is the instance of the object of interest.

Therefore, a simpler form of the object identifier for an instance of an object in the table is:

```
y.Row
```

In this simple-form object identifier, y, {TableID.TableEntry.Column} is the name of the attribute of interest within the table.

> **Note:** y.Row is a leaf node in the table, and is therefore a valid object identifier to use with the Get and Set SNMP primitives.

To provide an example, consider the ip.ipRouteTable defined in the standard MIB. The specification defines 13 attributes (columns) for each route (row) that appears in the table. The object identifier that corresponds to the next hop (attribute 7) of any route in the table contains the common prefix, 1.3.6.1.2.1.4.21.1.7, which corresponds to:

```
{ip.ipRouteTable.ipRouteEntry.ipRouteNextHop}
```

Therefore, the particular instance of the Next Hop attribute for a specific 1.2.3.4 route can be found by performing a Get request using an object identifier of 1.3.6.1.2.1.4.21.1.7.1.2.3.4.

How to Add a Table to the MIB

After developing an understanding of how tables are organized by SNMP, one can examine how tables are added to the g_snmpMIBInfo [] array in ZTP. Consider the declaration of the following If table in the g_snmpMIBInfo [] array in the snmib.c source file:

```
const UINT16 g_snmpIfTable[] = {2,1,2,2} ;
const UINT16 g_snmpIfEntry[] = {2,1,2,2,1} ;
{ g_snmpIfTable, 4, T_AGGREGATE, NULL, READ_WRITE,
NULL, NULL },
{ g_snmpIfEntry, 5, T_TABLE,
&sn_table[T_IFTABLE],READ_WRITE, sntable, NULL},
```

The object identifier for the root of the table is specified as `2.1.2.2`. As with leaf objects, the common prefix of `1.3.6.1` is omitted. The `If Table` is an aggregate object and contains a single child, `ifEntry` of type `T_TABLE`.

However, you might have the following questions:

- Where are the `ifEntry` child nodes that identify the columns of the table?
- Furthermore, why aren't any objects specified in the `g_snmpMIBInfo []` array for the `ifEntry` Table?

The reason these objects cannot be included in the `g_snmpMIBInfo []` array is because the leaf nodes in the table can only be determined at run time. Therefore, the SNMP library must call support routines at run time that help it to perform `Get`, `Get Next` and `Set` requests for objects located beneath the `ifEntry` node in the `g_snmpMIBInfo []` array.

These support routines are specified in the `sn_table[]` array. For every object in the `g_snmpMIBInfo []` array of type `T_TABLE`, its value member must reference an `SNMP_TABLE_S` structure in the `sn_table[]` array, as indicated in the following code fragment.

```
typedef struct SNMP_TABLE_S
{
  SNMP_GET_FUNC ti_get;     /* get operation */
  SNMP_NEXT_FUNC ti_next;   /* get next index */
  SNMP_SET_FUNC ti_set;     /* set operation */
  UINT16 max_fields;        /* number of 'rows'
                            /*in the table */
  UINT16 index_len;         /* number of */
```

```
                                        /*subidentifiers in*/
                                        /*index */
  } SNMP_TABLE_S;
```

The ti_get, ti_next and ti_set members specify the helper functions that the SNMP library uses when responding to Get, Get Next and Set requests for objects within the table. If you add your own tables to the MIB, you must implement these support routines.

The max_fields member indicates the number of columns in the table. For example, the specification of the IF table in the standard MIB identifies 22 attributes (child nodes) to the ifEntry. Therefore, the max_fields parameter for the IF table is specified as 22.

The index_len member defines the number of subidentifiers in the name of the TableEntry within the g_snmpMIBInfo [] hierarchy. For example, the complete TableEntry name of the ifEntry is:

1.3.6.1.2.1.2.2.1

However, in the ZTP implementation, the common root of 1.3.6.1 is not included for any entry in the g_snmpMIBInfo []. Therefore, the index_len of the ifEntry in the ZTP implementation is 5, which corresponds to an identifier of 2.1.2.2.1.

The ti_mip member cross-links the sn_table[] entry to the corresponding TableEntry in the g_snmpMIBInfo[] array, the object value of which references this sn_table[] entry. The SNMP library automatically determines the value of this pointer and updates the ti_mip value during SNMP initialization.

To add a table to the g_snmpMIBInfo [] array, you must also add an entry to the existing sn_table[] array to describe your table to the SNMP library. The code fragment below indicates the addition of a table to the g_snmpMIBInfo [] array.

```
{
 g_udpEntry,
 5,
```

```
 T_TABLE,
 &sn_table[T_SUETABLE],
 READ_ONLY,
 sntable,
 NULL
UINT16 g_snmpTableAgEg[] = {4,1,12897,2,19} ;
UINT16 g_snmpTableEg[] = {4,1,12897,2,19,1},

{ g_snmpTableAgEg , 5 , T_AGGREGATE, NULL,READ_ONLY,
NULL, NULL},
{ g_snmpTableEg , 6}, T_TABLE, &sn_table[7],
READ_ONLY, sntable, NULL }
```

The corresponding entry in the `sn_table[]` array is:

```
{sdt_get, sdt_next, sdt_set, SNUMF_DTTAB,
SDT_INDEX_LEN, NULL }
```

This table contains `SNUF_DTTAB` fields (currently defined to be 3). The rows in the table are indexed by a single subidentifier so that the value of `SDT_INDEX_LEN` is defined to be 1.

The SNMP_GET_FUNC Support Routine

The SNMP library calls the routine you specify in the `ti_get` field of the `SNMP_TABLE_S` structure when it requires the value of the named object in response to a `Get` SNMP request. A compatible function prototype for the `SNMP_GET_FUNC` function pointer is shown below.

```
INT Table_GET( SNMPObjLs * pObj );
```

In the `Table_GET` function, the `pObj` parameter references an incomplete SNMP object. Recall that SNMP objects contain a name, a type, and a value. When processing a `Get` SNMP request, the library is only able to determine the name (`pObjobjId`) of the requested object. It is up to your `Table_GET` routine to either supply the type and value of the object or to return integer error codes such as `SERR_NO_SUCH`. How objects are stored within your table is an implementation decision.

The pObj objId parameter contains the object identifier that corresponds to the instance of the object that is the target of a Get request. The SNMP library has no means by which to determine if a requested object is actually within your table. The only preprocessing that the library performs on the requested object identifier is to remove the common mib[] root prefix of 1.3.6.1, and to ensure that the requested object identifier begins with the same subidentifiers as the TableEntry corresponding to the ti_mip pointer in your table's sn_table[] array entry.

Therefore, the first task to be performed in the Table_GET routine is to ensure that the requested object is within your table. If the requested object cannot be located in the table, return the SERR_NO_SUCH error code to the SNMP library. Do not perform any further processing on the pObj parameter.

As a result of verifying that the object identifier is valid, the Table_GET routine should determine the row (table index) and column (field) of the applicable object. Converting the row and column identifiers into a meaningful index you can use at run time to access the value of the requested object is an implementation-specific design issue.

After your Table_GET routine locates the applicable object, the next step is to update the pObj objValType and pObj objVal fields, as appropriate. For more information about SNMP objects and data types in ZTP, see the SNMP Objects section on page 81.

As a simple example, if the Table_GET routine determines that the applicable object is a 32-bit ASN.1 counter, set pObjobjValType to ASN1_COUNTER, and set the *pObj objVal pCounter to the 32-bit unsigned value of the counter.

> **Note:** The code must not modify the value of pObjobjVal. You can only modify memory referenced by one of the members of the pObj Value union.

Before calling your `Table_GET` handler, the SNMP library allocates a buffer of size `g_snmpMaxObjectSize` and sets the `pData` and `MaxLen` members of `pObjobjVal.pDescr`.

The SNMP_SET_FUNC Support Routine

The SNMP library calls the routine specified in the `ti_set` field of the `SNMP_TABLE_S` structure when it requires you to update the value of the named object in response to an `Set` SNMP request. A compatible function prototype for the `SNMP_SET_FUNC` function pointer is:

```
INT Table_SET(SNMPObjLs *pObj);
```

The `pObjobjId` parameter contains the object identifier that corresponds to the instance of the object that is the target of a `Set` request. The SNMP library cannot determine if the requested object is actually within your table. The only preprocessing that the library performs on the requested object identifier is to remove the common `mib[]` root prefix of `1.3.6.1` and to ensure that the requested object identifier begins with the same subidentifiers as the `TableEntry` corresponding to the `ti_mip` pointer in the `sn_table[]` entry.

Therefore, the first task to be performed in the `Table_SET` routine is to ensure that the requested object is, in fact, within your table. If the requested object cannot be located in the table, return the `SERR_NO_SUCH` error code to the SNMP library. Do not perform any further processing on the `pObj` parameter.

As a result of verifying that the object identifier is valid, the `Table_SET` routine should have determined the row (table index) and column (field) of the applicable object. To convert the row and column identifiers into a meaningful index, you must use a design-specific implementation.

After the `Table_SET` routine locates the applicable object, the next step is to verify that the `pObjobjValType` field is compatible with the internal representation of the indicated object. Some SNMP object types are syntactically specified as one object type, but are encoded using a compatible ASN.1 primitive data type. For example, SNMP displays strings

(represented in ZTP as type SN_DISPLAY_STR) that are encoded as ASN.1 octet strings. Therefore, even though you can create an object of type SN_DISPLAY_STR, the SNMP library sets the pObjobjValType parameters to ASN1_OCTSTR before calling your Table_SET routine. Consequently, your Table_SET routine should accept an octet string as a valid data type for the SN_DISPLAY_STR object. However, if the remote SNMP management entity specified the object type as an ASN.1 integer in the Set request, then your Table_SET routine should not accept the value of the object, because an integer data type is not compatible with either an octet string or a display string. In this case, your Table_SET routine should return SERR_BAD_VALUE to the SNMP library and perform no further processing.

After the Table_SET routine verifies that the applicable object exists within the table and that the pObjobjValType field of the object is appropriate, the next step is to determine if the object size is valid. If the value of g_snmpMaxObjectSize is defined appropriately, then the SNMP library ensures that pObjobjVal does not exceed g_snmpMaxObjectSize bytes of data. Otherwise, for object values that use an SNMPDisplayStr (arbitrary length integers and octet strings), ensure that the size of the object value specified in the Set operation does not exceed the size of the local buffer you are using to contain the value of the target object. If the value of the object specified in the Set operation exceeds the storage capacity of the local buffer, your Table_SET routine should return SERR_TOO_BIG and not perform any more processing of the pObj parameter.

It can also be appropriate to verify the correctness of the object value specified in the Set operation. For example, if you define an object within your table of type Integer and specify that the permissible range of values for that integer is between 10 and 20, then you can either allow the remote management entity to assign an invalid value to your object (such as +1729, or –15) or you can return SERR_BAD_VALUE to the SNMP library and perform no further processing on the pObj parameter.

If the object name, type and value specified in the `Set` operation are all appropriate, then the final step to perform in the `Table_SET` routine is to copy the value of the `pObj` input parameter into the memory location in which you are storing the value of the target object. It is important to copy the contents of the value into your local buffer and not to retain the value of the pointer that the library provides to the object data. This task is important because the object value supplied by the SNMP library is located in a dynamically allocated buffer that is released after your `Table_SET` routine returns control to the SNMP library.

As a simple example, if the `Table_SET` routine determines that the target object is a 32-bit ASN.1 counter, you would set the value of the counter to the value `* pObj` `objVal` `pCounter`.

For more information about objects, see the SNMP Objects section on page 81.

The SNMP_NEXT_FUNC Support Routine

The SNMP library calls the routine specified in the `ti_next` field of the `SNMP_TABLE_S` structure when it requires you to determine the name of the object that immediately follows (in lexicographical order) the specified object identifier. A compatible function prototype for the `SNMP_SET_FUNC` function pointer is:

```
INT Table_NEXT(UINT16 * pSubID);
```

The `pSubID` pointer references an array of subidentifiers that begins with the table index (i.e., row) of interest. The `Table_NEXT` routine must modify this list of subidentifiers to match the index of the next row in the table. If there is no element in the table that follows the specified index, the `Table_NEXT` routine should return `SERR_NO_SUCH` and not modify the specified list of subidentifiers.

The key to implementing the `Table_NEXT` function is to understand what is meant by *lexicographical order*. Lexicographical order is sometimes referred to as *dictionary order*. In a dictionary, the definition of the word *the* appears before the definition of the word *then* but after the definition

of the word *tap*. Therefore, the lexicographical ordering of these words is *tap the then*.

For example, if your table index is defined as an IP address containing three rows, then the index of each row is:

```
192.168.1.50
192.168.1.200
192.168.4.75
```

Therefore, it is easy to see that if the `pSubID` array contains the value `192.168.1.50`, then the next index (in lexicographical order) that appears in your table is `192.168.1.200`. If the `pSubID` array contains the values `192.168.1.60`, there is no row in your table with this index. However, the next row in the table that follows in lexicographical order is still `192.168.1.200`, because the value 60 is between 50 and 200. Similarly, the next entry in the table after `192.168.2.1` is `192.168.4.75`.

Determining lexicographical order is slightly more complicated if the `pSubID` contains more or less subidentifiers than the amount you expect as a valid index. For example, given an input subidentifier string of `192.168.2`, the next index is `192.168.4.75`. Similarly, the next index after `192.168.1.3.4.5.6.7.8.9`. is also `192.168.4.75`. Two other special cases to consider are the case in which the `pSubID` input array references an item before the first object in your table (for example, for an input of `100.1`, the next row in the table is `192.168.1.50`) and the case in which there is no element in the table that follows the input list of subidentifiers. For example, given an input of `192.168.4.75`, there is an element in the table that follows this subidentifier so that the `Table_NEXT` routine should return `SERR_NO_SUCH`.

The SNMP library automatically pads subidentifiers shorter than the `index_len` specified in the `sn_table[]` array entry for your table with a zero to simplify lexicographical processing.

Updating SNMP Values

The SNMP library automatically updates the values of SNMP objects defined in the standard MIB. However, you are free to update the values of SNMP objects specific to your application, if appropriate. For example, if you define an SNMP object of type `Counter` to count some event unique to your application and add it to the `g_snmpMIBInfo []` array, the SNMP library's `Get` and `Set` functions obtain and set the value of the object in response to requests from a remote SNMP management entity. However, it is up to your application to increase the value of the counter when the trigger event occurs. Conversely, if you define an SNMP object of type `Octet String` to contain the serial number of your embedded device, it is likely that you do not require an update of this value during run-time.

Additionally, the SNMP library does not use critical sections (i.e., does not disable interrupts) while manipulating objects within the `g_snmpMIBInfo []` array. If this issue causes problems for your application, then Zilog recommends that you update SNMP objects in a process that runs at a lower priority than the SNMP agent (which currently executes at priority 20) and that your application only updates SNMP variables from within a critical section. The first measure ensures that the process in your application which updates SNMP variables can never preempt the SNMP agent while it is manipulating an object within the `g_snmpMIBInfo []` array. The second measure ensures that the SNMP agent (nor any other process in the system) is not able to preempt the process in your application that updates SNMP variables.

Note: The `g_snmpMIBInfo []` array is contained in RAM. Any changes made to the `g_snmpMIBInfo []` array are lost when power is removed from the system.

Trap Generation

The SNMP library in ZTP is capable of generating the following SNMP v1 traps:

- Cold Start trap
- Link Up trap
- Link Down trap
- Authentication Failure trap
- Enterprise-specific trap

A Cold Start trap is generated when the system boots up, 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). This situation occurs because the `g_snmpMIBInfo[]` is stored in RAM, and any changes made to the MIB – which could affect the operation of this device – are lost when the system is reinitialized. Therefore, from an SNMP perspective, every initialization is a Cold Start.

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.

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 a PPP link is disconnected.

If the `SnmpEnableAuthenTraps` variable is set to `SNMP_AUTH_TRAPS_DISABLED`, the system generates an Authentication Failure trap whenever a request (`Get`, `Get Next` or `Set`) is received containing a community name that does not match the community name specified in the `g_snmpCommunityName[]` string. The `SnmpEnableAuthenTraps` variable can be modified by your application or by a remote management entity.

The above-described traps are sent by the SNMP system automatically, but an enterprise-specific trap must be sent by the application itself. The code fragment that follows provides an example of how an enterprise-specific trap is sent using the `snmpGenerateTrap()` API.

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

In this `snmpGenerateTrap()` code, the following parameters can be defined:

pMgrAddress. The IP Address of the target device. If the parameter passed is NULL, then the traps are directed to the device identified by the `g_snmpTrapTargetIP[]` variable in the `snmp_conf.c` file.

Type. The `Type` parameter should always be specified as `SN_TRAP_ENTERPRISE_SPECIFIC`.

Code. The `Code` parameter is a 32-bit value unique to the application that identifies the particular trap message that is generated. For SNMPV2 and SNMPV3, this parameter is not valid.

NumObjects. The `NumObjects` parameter specifies the number of `SNMPObj` structures that must be included in the body of the trap message. If the application-specific trap does not require any of the objects to be included in the trap message, then set this parameter to 0.

pObjList. The `pObjList` parameter is 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 of the objects to be included in the trap message, set this parameter to NULL.

To send an SNMPV2/SNMPV3 enterprise-specific trap, you must provide the trap OID in the `g_snmpEnterpriseOid[]` variable located in the `snmp_conf.c` file.

Working with SNMPv3

ZTP SNMPv3 supports the HMAC MD5 protocol for authentication and the CBC-DES protocol for privacy.

To add new MIBs to SNMPv3, you must modify the `snmpv3_mib.c` file located in the `..\ZTP\Conf` folder. For more details about how to add MIBs to SNMP, see the Adding Objects to the MIB section on page 84.

SNMPV3 is released along with SSL as a separate package; the SNMPV3 code is located in `CommoProtoLib_SNMPV3.lib` file. To work with SNMPV3, define a SNMPV3 macro and add the `CommoProtoLib_SNMPV3.lib` library to the workspace.

SNMPV3 Demo project workspaces are located in the following path:

```
<ZTPInstalledDirectory>\ZTP_X.Y.Z_Lib_ZDS\ZTP\
SamplePrograms\SNMPV3Demo
```

In this path, `X.Y.Z` represents the ZTP version number.

You must configure the eZ80 CPU for authentication and privacy features. The structure to provide the required information is:

```
typedef struct snmpV3_usrs
{
 UINT8 username [SNMP_USER_NAME_LEN];
 UINT8 snmpAuthKeyChange [SNMP_AUTH_PASSWORD_LEN];
 UINT8 authReqd;
 UINT8 snmpPrivKeyChange [SNMP_PRIV_PASSWORD_LEN];
 UINT8 privReqd;
} SNMPV3_USERS;
```

In the above structure, the following parameters can be defined:

username. A parameter is a string that contains a valid user name.

snmpAuthKeyChange. This parameter specifies a `authKeyChange` value required for authentication.

authReqd. This parameter is used to enable or disable the authentication feature.

snmpPrivKeyChange. This parameter is a `privacyKeyChange` value required for privacy.

The `Snmp_Users` global variable that is defined in the `snmpV3_config.c` file contains the default values. You can change these values or add new values, if required.

How to Use the SNTP Client

To get the current time from the Time server using the Simple Network Time Protocol (SNTP), ZTP provides the `ztpSNTPClient` function, which is used to update the real-time clock within the Zilog Real-Time Kernel (RZK). This function establishes a separate UDP connection to communicate with the server, and sends the message format to the specified `targetIPAddress` through `portNum` 123. The function receives the time (in seconds) from `targetIPAddress`, converts this time into the RZK Device format, and updates the RZK real-time clock.

The prototype of the `ztpSNTPClient` function is:

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

In this prototype, `targetIPAddress` is a pointer to the IP address of the time server, and `portNum` is the port number through which the client communicates with the Time server.

ZTP Shell Command Reference

ZTP includes a shell program that allows you to interactively enter commands and query status information. The shell can be used via any device over which a TTY driver is layered. By default, the ZTP system configures Serial Port0 for use as a console to the shell. Similarly, the Telnet server layers a TTY device over the TCP connection created to service the Telnet session and allow another instance of the shell to run. You can modify the list of default commands that can be executed from within the shell. For details, see the description of `shell_conf.c` file in the the Configuring the SHELL section on page 24. Additionally, see the descriptions of `shell_add_commands` and `shell_init` functions in the the How to Use the Shell section on page 74.

Table 1 provides a brief description of the shell commands.

Table 1. Shell Commands

Shell Command	Description
addusr	Adds a user to the FTP server database
arp	Display the ARP table
bpool	Display buffer pool information
cd	Changes the current working directory
close	Used to terminate a Telnet session but remain in Telnet mode
copy	Copies a file
cwd	Displays the current working directory
del	Deletes a file
deldir	Deletes a directory
deleteusr	Deletes a user from the FTP server database
deltree	Deletes a directory, including the subdirectories and the files within the directory

Table 1. Shell Commands (Continued)

devs	Print device table
dir	Displays the list of files and directories present in a directory
echo	Echo arguments
exit	Exit shell
ftp	Starts an FTP session
gettime	Returns the date and time
hang	Puts system into a tight loop (system hang) for test
help	Print help information
ifstat	Print interface status
igmp	Subscribe and unsubscribe to multicast groups
kill	Kill a process
mail	Interactively compose an email message
md	Creates a directory
mem	Print memory usage information
move	Moves a file from source to destination
open	Used to initiate a Telnet session
ping	Send ICMP echo (ping) packets
port	Display port information
pppstart	Start PPP connection
pppstop	Force the PPP layer to disconnect from the remote peer
ps	Display process information
quit	Used to terminate a Telnet session but exit Telnet mode
reboot	Reboot system
ren	Renames a file
rendir	Renames a directory
sem	Display semaphore information
settime	Sets the time and date

Table 1. Shell Commands (Continued)

sleep	Places the shell process to sleep for a specified number of seconds
telnet	Starts a telnet session
tftp_get	Download a file from a TFTP server
tftp_put	Upload a file to the TFTP server
type	Displays the contents of a file
vol	Displays the disk details
SNTPClient	Retrieves the current time from SNTP server
setipparams	Stores network related parameters such as IP address, subnet Mask in the nonvolatile memory.
configwlan	Stores the SSID and encryption key in nonvolatile memory. It requires a pass phrase to generate the key.
keyIndex	Changes the WEP encryption key at run time
pass-phrase	Generates PSK for WPA and WPA2

Table 2 provides brief description of the shell commands that are applicable only to Zdots SBC WLAN solution.

Table 2. Shell Commands (Zdots® SBC WLAN Solution)

Shell Command	Description
scan	Searches and displays the available wireless Access Points in the vicinity.
join	Allows the Zdots SBC to connect to a particular Wireless Access Point that is scanned previously by the `scan` command.
configwlan	Adds WLAN configuration parameters to Data persistence.

ADDUSR

Syntax

`addusr username password`

Description

The `addusr` command adds a user to the FTP server database. You can log in from an FTP client only if the login name and password has already been added by a system administrator with the `addusr` shell command.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>addusr john john123
```

ARP

Syntax

`arp`

Description

The `arp` shell command prints resolved Internet addresses to physical address maps. There is one row in the table for each of the Datalink/Physical Layer mappings in effect.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>arp
------------------------------------
172.16.6.1      0:E0:7B:F3:96:B2
172.16.6.2      0:D0:B7:8F:5:92
172.16.6.10     0:80:C8:1:96:9
172.16.6.4      0:D:56:29:FE:E1
```

BPOOL

Syntax

`bpool`

Description

The `bpool` shell command displays information about each of the system's buffer pools. Each line in the display corresponds to one of the system's fixed buffer pools. The `state` field indicates the state of the buffer pool. The `count` field indicates the number of buffers initially created within each pool. Each buffer is `size` bytes long.

Because these pools are a shared resource, each use a semaphore to synchronize requests for buffers. `UsedBlocks` indicates the number of available blocks in the semaphore.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>bpool
pool     state    count    size     UsedBlocks
------------------------------------------------------
B8FC53    -
B8FC78    -
B8FC9D    -
B8FCC2    -
B8FE7E    -
B8FEA3    -
B8FEC8    -
B8FEED    -
B8FF12    -
- means unallocated
[ZTP EXTF:/]>
```

CD

Syntax

```
cd <directory Name>
```

Description

The `cd` command changes the current working directory. The user of this command can specify the absolute path or relative path in the argument. If the directory specified is not present in the Zilog File System an error is returned.

Argument(s)

`directory name`	Name of the directory.

Sample Usage

Example 1

```
[ZTP EXTF:/]> md zilog
[ZTP EXTF:/]> cd zilog
```

The above command changes the current working directory to `zilog`.

Example 2

```
[ZTP EXTF:/zilog]>
[ZTP EXTF:/zilog]>cd /
```

The above command changes the current working directory to the root.

Example 3

```
[ZTP EXTF:/zilog/ZTP]>cd ..
```

The above command changes the current working directory to the parent directory (one level up). The current working directory becomes:

```
[ZTP EXTF:/zilog ]>
```

CLOSE

Syntax

`close`

Description

The `close` command is used to terminate a Telnet session. Control is still in Telnet mode; the connection can be reestablished by calling the `open` command.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]> telnet
eZ80 Telnet% open xxx.xx.x.xx
Welcome to xxx server.
Login:
Password:

Server prompt>>

eZ80 Telnet %close

eZ80 Telnet %
```

COPY

Syntax

```
copy <srcfileName> <destnDirName>
```

Description

The `copy` command copies a file from one location to another. The file is copied to the destination directory with the same name.

Argument(s)

`srcfileName`	File that must be copied.
`destnDirName`	Destination directory name.

Sample Usage

```
[ZTP EXTF:/ZILOG]> copy one.txt [ZTP EXTF:/ZILOG/ZTP]>
```

The above command copies the `one.txt` file from `[ZTP EXTF:/ZILOG]>` to `[ZTP EXTF:/ZILOG/ZTP]>`.

CWD

Syntax

`cwd`

Description

The `cwd` command displays the current working directory.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/ZILOG]> cwd
```

The current working directory is `[ZTP EXTF:/ZILOG]>`

DEL

Syntax

```
del <file Name>
```

Description

The `del` command deletes a file if exists.

Argument(s)

`file name`	File that must be deleted.
`destnDirName`	Destination directory name.

Sample Usage

Example 1

```
[ZTP EXTF:/]> del one.txt
```

The above command deletes the `one.txt` file present in the `[ZTP EXTF:/]>` directory.

Example 2

```
[ZTP EXTF:/]> del ZTP EXTF:/ZILOG/ZTP/one.txt
```

The above command deletes the `one.txt` file present in the `[ZTP EXTF:/ZILOG/ZTP]>` path.

Example 3

```
[ZTP EXTF:/ZILOG/ZTP]> del ../../one.txt
```

The above command deletes the `one.txt` file present in the `[ZTP EXTF:/]>` directory.

DELDIR

Syntax

```
deldir <directory Name>
```

Description

The `deldir` command deletes a directory if it exists. If the directory is empty, only then the specified directory is deleted; otherwise an error is returned.

Argument(s)

`directory name`	Name of the directory to be deleted.

Sample Usage

Example 1

```
[ZTP EXTF:/ZILOG]> deldir ZTP
```

The above command deletes the ZTP directory, which is present in the `[ZTP EXTF:/ZILOG]>` directory.

Example 2

```
[ZTP EXTF:/ZILOG]> deldir /ONE
```

The above command deletes the `ONE` directory, which is present in the `[ZTP EXTF:/]>` (root directory).

Example 3

```
[ZTP EXTF:/ZILOG/ZTP]> deldir ../ONE
```

The above command deletes the `ONE` directory, which is located in the `ZILOG` directory.

DELETEUSR

Syntax

```
deleteusr username
```

Description

The `deleteusr` command deletes an existing FTP user from the FTP server database.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>deleteusr john
```

The above command deletes the user `john`.

DELTREE

Syntax

```
deltree <directory Name>
```

Description

The `deltree` command deletes a directory tree. All of the subdirectories and files, if present, are also deleted.

Argument(s)

`directory name`	Name of the directory to be deleted.

Sample Usage

```
[ZTP EXTF:/ZILOG]> md ONE
[ZTP EXTF:/ZILOG]> md TWO
```

There are two subdirectories present in the `ZILOG` directory.

```
[ZTP EXTF:/]> deltree ZILOG
```

The above command removes directories `ONE` and `TWO` in addition to the `ZILOG` directory.

DEVS

Syntax

`devs`

Description

The `devs` shell command prints device information in the device table. The size of this table is fixed and cannot be adjusted. However, you can alter the number of devices that the system adds to the table.

Argument(s)

None.

Sample Usage

```
-----------------------------------
  [ZTP EXTF:/]>devs
-----------------------------------
Device Name iVec    Minor   CtlBlk
TCP         0000    0000    000000
UDP         0000    0000    000000
CONSOLE     0000    0000    000000
SERIAL0     2E0B    0000    00B114
SERIAL1     354C    0000    00B141
TTY         0000    0000    000000
TTY         0000    0000    006F4C
TTY         0000    0001    000000
TTY         0000    0002    000000
TTY         0000    0003    000000
EMAC        385A    0000    00B2C4
```

```
RTC          7B35     0000     000000
HDLC         0000     0000     000000
mSSLm        0000     0000     000000
SSL          0000     0000     003BF1
SSL          0000     0000     004098
SSL          0000     0000     005334
SSL          0000     0000     0057DB
SSL          0000     0000     005C82
FREE
FREE
FREE
```

DIR

Syntax

```
dir < directory Name>
```

Note: The `directory Name` argument is optional.

Description

The `dir` command displays a list of subfolders and files present in a directory. If the argument is not present, the files and subfolders present in the current working directory are returned; otherwise the list of files and subfolders present in the specified directory is returned.

Argument(s)

`directory name`	Name of the directory (optional).

Sample Usage

```
[ZTP EXTF:/]>dir
******************************************************
DATE         TIME       TYPE     SIZE (bytes)   NAME
******************************************************

08/23/2004   14:41:02            958            C2.txt
08/23/2004   18:47:08            1001           C6.txt
09/02/2004   16:24:48   <DIR>                   ZILOG
08/23/2004   18:47:08   <DIR>                   ONE

 Number of file(s) 2
 Number of Dir(s) 2
******************************************************
[ZTP EXTF:/]>
```

ECHO

Syntax

```
echo [text]
```

Description

The `echo` shell command is used to echo the text entered after the echo command to the standard output device associated with the shell processing the command. If the echo command is issued on the console device, the input string is echoed to the console. If the `echo` command is issued in a Telnet session, the input string is echoed to the Telnet session.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]> echo Zilog
Zilog
[ZTP EXTF:/]>
text Test string to be echoed to the shell's standard
output device.
```

EXIT

Syntax

`exit`

Description

The `exit` shell command terminates the shell process. If this command is issued to the shell associated with a Telnet session, the Telnet session is effectively terminated. If this command is executed on the console, it is the last input or output operation that is processed by console. After the console shell is terminated, reboot the system to restart.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]> exit
```

FTP

Syntax

`ftp [hostname]`

Description

The `ftp` shell command is the user interface to the Internet-standard File Transfer Protocol (FTP). The program allows you to transfer files to and from a remote network site.

The server host with which FTP is to communicate can be specified on the command line. FTP immediately attempts to establish a connection to an FTP server on that host; otherwise, FTP enters its command interpreter and awaits instructions from the user. When FTP is awaiting commands, the `'ftp>'` prompt appears in the console. FTP provides a list of supported commands if the `Help` command is entered without any arguments. If the `Help` command is entered with an FTP command name, it displays a help topic specific to that command.

Table 3 lists the commands recognized by `ftp`.

Table 3. FTP Commands

FTP Command	Description
ascii	Set the file transfer type to the network ASCII (default).
bin	Set the file transfer type to support binary image transfer.
bye	Terminate the FTP session with the remote server and exit FTP. An end-of-file command also terminates and exits the session.
cd remote directory	Change the working directory on the remote machine to the remote directory.
close	Terminate the FTP session with the remote server and return to the command interpreter. Any defined macros are erased.

Table 3. FTP Commands (Continued)

FTP Command	Description
delete remote file	Delete the file on the remote machine.
dir [remote directory]	Print a listing of the directory contents in the 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 filename is not specified, it retrieves a file containing the same name as the file on the remote machine. The current settings for type, form, mode, and structure are observed while transferring the file.
hash	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.
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 produce output from the ls -l command. (Also see nlist. If the 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.
nlist [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.

Table 3. FTP Commands (Continued)

FTP Command	Description
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	Print the name of the current working directory on the remote machine.
quit	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] file.
rmdir directory name	Delete a directory on the remote machine.
system	Show the type of operating system running on the remote machine.

GETTIME

Syntax

```
gettime
```

Description

The `gettime` command retrieves the current time from the Real-Time Clock using the RTC driver.

Argument(s)

None.

Sample Usage

If the `gettime` command is entered in the ZTP shell, the current time is displayed in the following format:

```
[ZTP EXTF:/]> gettime
Sun , 25 Mar 2007 10:25:20
```

HANG

Syntax

`hang`

Description

The `hang` shell command intentionally hangs the system. As a result of executing this command, maskable interrupts are disabled and the processor spins in a tight loop. Unless the Watchdog Timer is activated to process a nonmaskable interrupt (NMI), the only way to recover is to reboot the system.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>hang
```

HELP

Syntax

`help`

Description

The `help` shell command displays the set of commands that can be executed from the shell's command prompt. Multiple instances of the shell can be active at the same time, but each instance shares the same command set.

Argument(s)

None.

Sample Usage

Commands are:

```
?             SNTPClient   addusr      arp
bpool         cd           copy        cwd
del           deldir       deleteusr   deltree
devs          dir          echo        exit
ftp           gettime      hang        help
ifstat        igmp         kill        mail
md            mem          move        netstat
ping          port         pppmode     pppopt
pppstart      pppstat      pppstop     ps
reboot        ren          rendir      sem
setipparams   settime      sleep       telnet
tftp_get      tftp_put     type        vol
[ZTP EXTF:/]>
```

IFSTAT

Syntax

`ifstat`

Description

The `ifstat` shell command prints network interface information for all of the available interfaces. The following display provides an example of the `Ifstat` command.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>ifstat
------------------------------------------------------
IP address     Def Gtway   State Type      H/W address
172.16.6.212   172.16.6.1  UP    Ethernet  0:90:23:0:3:3
192.168.2.12   192.168.2.1 DOWN  PPP        -
------------------------------------------------------
```

IGMP

Syntax

`igmp interface {join | leave} group`

Description

The `igmp` shell command adds or removes a specified IP multicast address from the list of addresses a host is using. This information can also be conveyed to the IGMP layer using the `hgjoin` and `hgleave` APIs. The IGMP protocol ensures that IP multicast routers in the same subnet as the host forward IP multicast frames for all group addresses to any node that the router domain is using. Therefore, when group membership is no longer required, the IGMP `hgleave` API should be issued to avoid unnecessary multicasting.

Argument(s)

`interface`	The interface number of the primary Ethernet interface (for single Ethernet interface it should be 0).
`join | leave`	If the string join is provided as the first argument, membership is added. If the string leave is specified, group membership is terminated.
`group`	The IP multicast address of the group.

Sample Usage

```
[ZTP EXTF:/]>igmp 0 join 227.21.4.3
[ZTP EXTF:/]>
[ZTP EXTF:/]>igmp 0 leave 227.21.4.3
[ZTP EXTF:/]>
```

KILL

Syntax

```
kill process
```

Description

The `kill` shell command kills a specified process. This shell command performs the same function as the kill process manipulation API.

Argument(s)

`process`	The ASCII decimal integer representation of the process ID to be killed.

Sample Usage

```
[ZTP EXTF:/]>kill B8F036
```

MAIL

Syntax

`mail`

Description

The `mail` shell command is used to interactively compose an email message that is sent to an SMTP server for delivery to a specified recipient. This shell command performs the same operation as the `mail` API.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>mail
Press <ESC> then <Enter> to exit early
Enter the name or IP of the SMTP Server: 172.16.6.99
Enter the port number to connect to (normally 25): 25
Enter the user name (only if using CRAM-MD5
authentication else press ESC):satish
Enter the password (only if using CRAM-MD5
authentication else press ESC):satish
Enter the email Subject: Test mail
Enter the recipient's email address: test@zilog.com
Enter the sender's email address: eZ80@zilog.com
Enter the body of the email (Press ESC/Enter to
complete):
This is a test mail
Please wait while the message is processed.
Mail message was successfully sent.
```

MD

Syntax

`md <directory Name>`

Description

The `md` command creates a directory. If the argument contains only the name of the directory, then it is created in the current working directory. The directory name along with the path can also be specified if the directory must be created in a different location.

Argument(s)

`directory name`	Name of the directory to be created.

Sample Usage

Example 1

```
[ZTP EXTF:/]> md zilog
```

The above command creates a `zilog` directory in the `[ZTP EXTF:/]>` directory.

Example 2

```
[ZTP EXTF:/]> md EXTF:/zilog/ZTP
```

The above command creates a `ZTP` directory in the `[ZTP EXTF:/zilog]>` directory.

MEM

Syntax

`mem`

Description

The `mem` shell command prints a summary of the state of system memory. It prints an address of the region, the state, and the minimum allocatable size for the specified region.

Argument(s)

None.

Sample Usage

```
  [ZTP EXTF:/]>mem
Region       State   Unit Size
-------------------------------
B9099B       PRIO    16
B909C6       -
B909F1       -
B90A1C       -
B90A47       -
B90A72       -
B90A9D       -
B90AC8       -
B90AF3       -
B909B1E      -
B90B49       -
B90B74       -
```

```
B90B9F      -
B90BCA      -
B90BF5      -
B90C20      -
B90C4B      -
B90C76      -
B90CA1      -
B90CCC      -
  [ZTP EXTF:/]>
-------------------------------
```

MOVE

Syntax

```
move <srcfileName> <destnDirName>
```

Description

The `move` command moves a file from one location to another. The file is moved to the destination directory with the same name.

Argument(s)

`srcfileName`	File that must be moved.
`destnDirName`	Name of the destination directory.

Sample Usage

```
[ZTP EXTF:/ZILOG]> move one.txt EXTF:/ZILOG/ZTP
```

The above command moves the `one.txt` file from the `[ZTP EXTF:/ZILOG]>` directory to the `[ZTP EXTF:/ZILOG/ZTP]>` directory.

OPEN

Syntax

```
open <name or ipaddress of the remote system>
```

Description

The `open` command is used to start a Telnet session with a specified IP address. This command establishes a TCP connection with a server on port 23.

When a connection is established with the server, and after login, the server returns a prompt to the client. The client then begins executing commands on the server as if it were using the server's terminal.

Note: This command can be executed only after executing the `telnet` command with no arguments.

Argument(s)

`name or ipaddress of the remote system`	Name or IP address of the remote system to be opened.

Sample Usage

```
[ZTP EXTF:/]> telnet
eZ80 Telnet% open xxx.xx.x.xx
```

PING

Syntax

```
ping host [count [size [delay]]]
```

Description

The `ping` shell command sends ICMP echo request packets to a specified host and reports statistics upon successful replies.

Argument(s)

`host`	The IP address of the target system.
`count`	An optional number of packets to send. If this parameter is not specified, 10 packets are sent.
`size`	If the count parameter is specified, then the size in bytes of each ping packet can also be specified. If this parameter is not specified, each ping packet contains 56 bytes of data.
`delay`	If the size parameter is specified, then the delay (in seconds) between each ping request can also be specified.

Sample Usage

```
ZTP EXTF:/]>
[ZTP EXTF:/]>ping 172.16.6.1
ZTP2.3.0(C) Zilog Inc. [ping utility]
PING#1, Reply from 172.16.6.1 :rtt < 50 ms
PING#2, Reply from 172.16.6.1 :rtt < 50 ms
PING#3, Reply from 172.16.6.1 :rtt < 50 ms
Ping Stats:
Sent:3
Recvd:3
Success:100%
Avg RTT(Approx):0.00 sec
[ZTP EXTF:/]>
```

PORT

Syntax

```
port
```

Description

The `port` shell command formats and prints information about all message ports currently in use. There are twenty message ports available for use in the system (port IDs between 0 and 19). The state of active ports is 3 (see `ports.h` in the `includes` directory).

- `state` indicates the state of the message queue.
- `length` indicates the length of the message queue.
- `MaxMsgSize` indicates the maximum size of the message queue.
- `MsgSpaceLeft` indicates the amount of messages that the queue can accept.
- `Blocked Threads` indicates the number of threads blocked on the message queue.
- `start` indicates the start location of the message queue.

Argument(s)

None.

Sample Usage

```
 [ZTP EXTF:/]>port
port     state    length   MaxMsgSizeMsgSpaceLeftBlockedThreadsstart
------------------------------------------------------------------
B8FF37   PRIO     4        3          4            1              CAF2
B8FF68   -
B8FF99   -
B8FFCA   -
```

```
B8FFFB  -
B9002C  -
B9005D  -
B9008E  -
B900BF  -
B900F0  -
B90121  -
B90152  -
B90183  -
B901B4  -
B901E5  -
B90216  -
B90247  -
B90278  -
B902A9  -
B902DA  -
  [ZTP EXTF:/]>
---------------------------------------------------------------
```

PPPSTART

Syntax

`pppstart`

Description

Starts the PPP connection. Depending on the configuration of the `ppp_conf.c` file, PPP starts as a DCC client, a DCC server, a dial-up client, or a dial-up server. For more information about PPP configuration, see the description of the `ppp_conf.c` file in the Configuring PPP section on page 7.

Argument(s)

None.

Sample Usage

```
Sample Usage
[ZTP EXTF:/]> pppstart
Configuring PPP ...
PPP Started
[ZTP EXTF:/]>
```

PPPSTOP

Syntax

`pppstop`

Description

The `pppstop` shell command forces the PPP layer to disconnect from the remote peer. If the PPP layer is not connected when this command is issued, there is no effect. If the `do_auto_reconnect` flag is set to `TRUE` in the PPP structure, the PPP layer automatically attempts to reestablish the disconnected link. Therefore, if it is required that the PPP connection not be immediately reestablished after disconnecting, the `do_auto_reconnect` flag should be set to `FALSE` before calling the `pppstop` command. This command performs the same function as the `ppp_stop` API.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>pppstop
PPP Stop
Sending LCP_Terminate_Request...
[ZTP EXTF:/]>
PPP DEAD
```

PS

Syntax

`ps`

Description

The `ps` shell command displays information about all processes in the system that are created but not yet killed. The PID identifies each process and is used as an input parameter on the various process manipulation functions.

`Index`–It displays an index of the `ps` table.

`Name`–It displays the name provided to the process when it is created.

`pid`–It indicates the process ID/handle of the process in the system.

`State`–It indicates the scheduling state of each process. All processes not marked as `ready` or `curr` are blocked on a given resource or are suspended.

`Mode–It` indicates the mode of the created process.

`quantum`–It indicates the round-robin timeslice value of the thread.

`Priority`–It indicates the scheduling priority assigned when the process is created.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>ps
Index     Name        pid       state     quantum   priority
0         IDLE        B84ACF    ready     1         31
1         SYSIT       B84B4E    InfSus    20        0
```

```
2          SER0IT      B84BCD    InfSus    2          6
3          EMACIT      B84C4C    InfSus    2          6
4          http_rqst   B84CCB    InfSus    1          9
5          SYSD        B84D4A    FinBlock  1          20
6          DHCPtmr     B84DC9    Infblock  1          10
7          -           B84E48
8          FTPD        B84EC7    InfSus    1          28
9          TLNTD       B84F46    InfSus    1          10
10         SNMPD       B84FC5    InfSus    1          20
11         SHL         B85044    Running   1          24
12         -           B850C3
13         -           B85142
14         -           B851C1
15         -           B85240
16         -           B852BF
17         -           B8533E
[ZTP EXTF:/]>
```

QUIT

Syntax

`quit`

Description

The `quit` command is used to terminate the Telnet session. Control is completely relinquished from Telnet. The `telnet` command must be used to reestablish the connection.

> **Note:** The `quit` command can be executed only in the Telnet command mode.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]> telnet
eZ80 Telnet% open xxx.xx.x.xx
Welcome to xxx server.
Login:
Password:

Server prompt>>

eZ80 Telnet %quit

[ZTP EXTF:/]>
```

REBOOT

Syntax

`reboot`

Description

The `reboot` shell command causes the operating system to begin its initialization sequence. This command is not the same as the reboot command used as a hardware reset. For details, refer to the eZ80 product specification appropriate to your target processor.

Argument(s)

None.

REN

Syntax

```
ren <fileName1> <fileName2>
```

Description

The `ren` command renames a file.

Argument(s)

`fileName1`	Original filename
`fileName2`	New name

Sample Usage

```
[ZTP EXTF:/ZILOG]> ren one.txt newfile.txt
```

RENDIR

Syntax

```
rendir <dirName1> <dirName2>
```

Description

The `rendir` command renames a directory.

Argument(s)

`dirName1`	Original directory name.
`dirName2`	New directory name.

Sample Usage

```
[ZTP EXTF:/ZILOG]> rendir TCPIP ZTP
```

SEM

Syntax

`sem`

Description

The `sem` shell command displays information about all active semaphores. There is a finite number of semaphores in the system defined by `MAX_SEMAPHORESH` in the `ZSysgen.h` header file. Each row in the display shows the semaphore control block address, its state, its current semaphore count, its dynamic count, and the number of threads blocked on this semaphore.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>sem
sem         state    InitCount    DynamicCount   NumThreads
B864F5      PRIO     1            1              0
B8651B      PRIO     1            1              0
B86541      PRIO     1            1              0
B86567      PRIO     1            1              0
B8658D      PRIO     1            1              0
B865B3      PRIO     1            1              0
B865D9      PRIO     1            1              0
B865FF      PRIO     1            1              0
B86625      PRIO     1            1              0
B8664B      PRIO     1            1              0
B86671      PRIO     1            1              0
B86697      PRIO     1            1              0
B86755      -
B8677B      -
B867A1      -
[ZTP EXTF:/]>
```

SETTIME

Syntax

```
settime <year> <month> <dayofmonth> <dayofweek> <hrs>
<mins> <secs>
```

Description

The `settime` command writes the current time into the RTC registers using the RTC driver. After the time is set, RTC increments the time every second. After it is set initially, the correct system time is displayed, even if the eZ80 hardware is rebooted.

Argument(s)

`year`	Value of year.
`month`	Value of month (starting with Jan as 01 to Dec as 12).
`dayofmonth`	Value of the day of month.
`dayofweek`	Value of the day of the week (starting with Mon as 01 to Sun as 07).
`hrs`	Value of the hours in 24 hrs standard.
`mins`	Value of minutes.
`secs`	Value of seconds.

Sample Usage

If the `gettime` command is entered in the ZTP shell, the current time is displayed in the following format:

```
[ZTP EXTF:/]> settime 2007 03 25 07 10 25 20
[ZTP EXTF:/]> gettime
Sun , 25 Mar 2007 10:25:20
```

SLEEP

Syntax

```
sleep <seconds>
```

Description

The `sleep` command causes the shell to sleep for a specified number of seconds.

Argument(s)

`seconds`	The amount of time to sleep, in seconds.

Sample Usage

```
sleep <number of seconds to sleep>
```

Example

```
sleep 20
```

TELNET

Syntax

```
telnet <name or ipaddress of the remote system>
```

Description

The `telnet` command is used to start a Telnet session with a specified name or IP address of the remote system. This command establishes a TCP connection with a server on port 23.

When a connection is established with the server, and after login, the server returns a prompt to the client. The client then begins executing commands on the server as if it were using the server's terminal.

> **Note:** If an argument is not present, the shell reverts to Telnet mode and you must send the `open` command to establish the connection.

Argument(s)

```
name or ipaddress of the remote system
```

Sample Usage

Example 1

```
[ZTP EXTF:/]> telnet 172.16.6.xx
```

Example 2

```
[ZTP EXTF:/]> telnet
eZ80 Telnet% open xxx.xx.x.xx
```

TFTP_GET

Syntax

`tftp_get host filename`

Description

The `tftp_get` shell command is used to download a file from a TFTP server in which `host` is the IP address (dotted notation) of the TFTP server and `filename` is the name of the file to be downloaded from the server. The file is downloaded to the shell's current working directory (CWD). If the file name in the shell's CWD is same as the one that is downloaded from server, then the original file is overwritten with the new file name.

Argument(s)

`host`	IP address of the TFTP server.
`filename`	Name of the file to be retrieved from the TFTP server.

Sample Usage

```
[ZTP EXTF:/]>tftp_get 172.16.6.42 testfile.txt
Getting file : testfile.txt
......................
file transfer Successful
[ZTP EXTF:/]>
```

TFTP_PUT

Syntax

`tftp_put host filename`

Description

The `tftp_put` shell command is used to upload a file to the TFTP server in which `host` is the IP address (dotted notation) of the TFTP server and `filename` is the name of the file to be uploaded to the server. The file to be uploaded should be present in the shell's current working directory (CWD).

Argument(s)

`host`	IP address of the TFTP server.
`filename`	Name of the file to be uploaded to the TFTP server.

Sample Usage

```
[ZTP EXTF:/]>
[ZTP EXTF:/]>tftp_put 172.16.6.42 testfile.txt
Uploading file : testfile.txt

......................
file transfer Successful
[ZTP EXTF:/]>
```

TYPE

Syntax

```
type <filename>
```

Description

The `type` command displays the contents of the specified file.

Argument(s)

`filename`	Name of the file.

Sample Usage

```
[ZTP EXTF:/]>type RM.txt

Zilog's TCP/IP solution, ZTP, is an integrated,
preemptive multitasking OS and TCP/IP protocol
software suite that has been optimized for embedded
systems.
ZTP works in conjunction with the award-winning
eZ80Acclaim! family of Flash microcontrollers to
provide standard network connectivity in a wide range
of applications, such as industrial control,
automation, facility management, IP appliances, and
remote systems communication.
```

VOL

Syntax

`vol`

Description

The `vol` command displays details about the volume like the space left in the volume, the space already used, total space available in the volume and the dirty space. This command returns the details about all of the volumes present in the system.

Argument(s)

None.

Sample Usage

```
[ZTP EXTF:/]>vol

******************************************************
Volume Name Total Space Free Space  Used Space  Dirty Space
            (bytes)     (bytes)     (bytes)     (bytes)
******************************************************
EXTF        917504      219648      174080      523776


******************************************************
[ZTP EXTF:/]>
```

SNTPCLIENT

Syntax

`SNTPClient`

Description

The `SNTPClient` command retrieves the current time from SNTP server by sending a Time request using the SNTPClient protocol.

Argument(s)

`ServerIPAddress`	IP Address of the SNTP Time server.

Sample Usage

```
[ZTP EXTF:/]> SNTPClient 193.1.250.3

Fri, 12th Jan 2006, 14:03:57
```

NETSTAT

Syntax

`netstat`

Description

`netstat` is a diagnostic command that displays protocol statistics and TCP/UDP network connections.

Argument(s)

None.

Sample Usage

Each column heading in the following code is described below.

```
[ZTP EXTF:/]>netstat
FD  Type  Oport State   SrcPort DstPort SrcIP         DstIP
0   TCP   80    LISTEN  0       0       0.0.0.0       0.0.0.0
1   TCP   0     LISTEN  80      0       0.0.0.0       0.0.0.0
2   TCP   23    LISTEN  0       0       0.0.0.0       0.0.0.0
3   TCP   21    LISTEN  0       0       0.0.0.0       0.0.0.0
4   TCP   0     ESTB    21      4794    172.16.6.184  172.16.6.177
5   TCP   0     ESTB    23      4796    172.16.6.184  172.16.6.177
6   UDP   161
7   TCP   0     LISTEN  21      0       0.0.0.0       0.0.0.0
8   TCP   0     LISTEN  23      0       0.0.0.0       0.0.0.0
9   NA
10  NA
11  NA
12  NA
```

FD. Socket Descriptor, returned by the `socket()` API.

Type. Indicates whether the socket is a TCP or an UDP socket.

Oport. This is the port Number on which the Master socket is listening.

State. This is applicable to TCP Sockets, which indicates the TCP Connection states. The possible states are:

`LISTEN`	Socket in Listen state.
`SYNSENT`	SYN is sent.
`SYNREC`	Received SYN.
`ESTB`	Established connection with the remote system.
`FINWT1`	Sent FIN.
`FINWT2`	Sent FIN, received FINACK.
`CLOSEWT`	Received FIN, send ACK.
`CLOSING`	Sent FIN, received FIN (waiting for FINACK).
`LASTACK`	FIN received, sent FINACK and also FIN.
`TIMEWT`	Wait state after sending final FINACK.
`CLOSE`	Connection is closed.

SrcPort. Port Number of the local system.

DstPort. Indicates the Port Number of the remote System. In cases where the connection is not yet established, the port number is shown as zero.

ScrIP. IP Address of the local system.

DstIP. IP Address of the remote system.

SETIPPARAMS

Syntax

```
setipparams <DHCP [e]nable/[d]isable> <emac addr> <IP
addr> <GateWay> <Net Mask>
```

Description

The `setipparams` command stores DHCP state, EMAC address, IP address, Gateway address, and Network mask in the nonvolatile memory. This command stores the data permanently. Therefore, the network parameters can be retained across the reboots. The `setipparams` works only if `Init_DataPersistence()` is called in `ZTPInit_Conf.c` file `main()` function.

Argument(s)

`<DHCP [e]nable/[d]isable>`	Specifies whether DHCP should be enabled or disabled. e – enables DHCP. d – disables DHCP.
`<emac addr>`	The MAC of the module.

If DHCP fails, the following parameters are considered to be the default by the module.

`<IP addr>`	IP address.
`<GateWay>`	Gateway address.
`<Net Mask>`	Network mask.

Sample Usage

```
[ZTP:/]> setipparams d 00:90:23:00:01:01 192.168.1.50
         192.168.1.1 255.255.255.0
```

SCAN

Syntax

```
scan <[SSID]>
```

Description

The `scan` command searches the BSS (Access Points) available in the range.

Argument(s)

`SSID`	Specific Service Set Identifier (SSID) of the Access Point (AP); it is an optional parameter. If the SSID is not determined in scan, it searches all of the available APs; otherwise it searches only the specified SSID.

Sample Usage

```
[ZTP:/]> scan
```

or

```
scan XYZ
```

JOIN

Syntax

```
join <SSID> <[WEP Key]>
```

Description

The `join` command connects to the BSS (AP) of the specified SSID.

Argument(s)

`SSID`	The Service Set Identifier (SSID) of the Access Point (AP).
`WEP Key`	It is 10 or 26 hexadecimal characters. The WEP Key lengths for different encryption modes are:

Encryption	WEP Key length
None	Not applicable.
WEP40	10 hexadecimal characters.
WEP104	26 hexadecimal characters.

Sample Usage

`[ZTP:/]>` `join XXX` – No Encryption.

`join YYY 6B2DECA79E`: For WEP40, 10 characters WEP Key.

`join ZZZ 6B2DECA79E0D85E5DDBC9F4ECF`: For WEP104, 26-character WEP Key.

CONFIGWLAN

Syntax

`configwlan <SSID> <pass-phrase> <encType>`

Description

The `configwlan` command stores the specified Service Set Identifier (SSID) and WEP Key in the internal Flash information page. It uses a pass phrase to generate the key. This command works only for WLAN demo FLASH and COPY_TO_RAM configurations. This command stores the data permanently. Therefore, the WLAN configuration parameters can be retained across the reboots.

> **Note:** This command will take a long time to complete. While the PSK (Pre-Shared Key) is generated, a period will be displayed on the console approximately every 2-3 seconds.

Argument(s)

`SSID`	The Service Set Identifier (SSID) of the Access Point (AP).
`pass-phrase`	The pass-phrase to generate the key. For WEP-40, four keys are generated with the key index 0 to 3. For WEP-104 and WPA/WPA2, one key is generated with a key index 0.
`encType`	Indicates the type of encryption. 0: No encryption. 1: 64-bit WEP encryption.

2: 128-bit WEP encryption.

3: WPA/WPA2.

Sample Usage

`[ZTP:/]> configwlan AAA 00 0`	For no encryption.
`configwlan BBB abcde 1`	For WEP40.
`configwlan CCC zilog 2`	For WEP104.
`configwlan DDD zilog 3`	For WPA/WPA2.

KEYINDEX

Syntax

`keyIndex <Key ID> <[WEP Key]>`

Description

The `keyIndex` command is used to change the key index. Some APs can support more than one WEP key at a time, and this security feature can be used by the station if it allows changing the key at run time.This command is not available in the RAM configuration.

> **Note:** If the WEP key is provided as the second argument, the key is added/modified in the data persistence; else, the key set using the `configwlan` command will be considered for further data transfers.

Argument(s)

`Key ID`	Key index 0–3.
`WEP Key`	The WEP Key lengths for different encryption modes are 10 or 26 hexadecimal characters.

Sample Usage

`[ZTP:/]> keyIndex 0`	A WEP key of 0 is set for further communication.

PASS-PHRASE

Syntax

`pass-phrase <SSID> <pass-phrase>`

Description

The `pass-phrase` command is used to change or generate the PSK (Pre-Shared Key) for WPA and WPA2 in the RAM configuration. For COPY_TO_RAM and FLASH configurations, see the configwlan API definition on page 163.

Note: This command will take a long time to complete. While the PSK is generated, a period will be displayed on the console approximately every 2–3 seconds.

Argument(s)

`SSID`	The Service Set Identifier (SSID) of the Access Point (AP).
`pass-phrase`	A set of ASCII characters used to generate the PSK.

Sample Usage

`[ZTP:/]> pass-phrase XYZ abcdefghijkl`	PSK is generated for SSID XYZ and pass phrase 'abcdefghijkl'.

Appendix A. Creating ZTP Shell Commands

This appendix provides shows how to create your own shell command, using the PING command as an example.

ping Command Example

To ping a system from the eZ80 Development Platform, ZTP provides an example x_ping() function. This function sends a specified number of ping requests to a specified host, and displays the resulting statistics.

A prototype of the x_ping() function is:

```
INT16 x_ping ( struct shvars * Shl,
 RZK_DEVICE_CB_t * stdin,
 RZK_DEVICE_CB_t * stdout,
 RZK_DEVICE_CB_t * stderr,
 UINT16 nargs,
 INT8 * args[] )
```

In the above code, Shl is a pointer to the shvars structure defined for the shell, and therefore represents the shell. You must declare a variable for the shvars structure and pass the address of this variable as a parameter to the x_ping() function.

stdin, stdout and stderr are integer values that specify the device to which the data is to be written. nargs is the number of arguments to the command, and args is an array string containing the command and its arguments.

Sample Usage

```
INT16 MyFunction(void)
{
 struct shvars Shl;
 RZK_DEVICE_CB_t * TTYDevID; // Keep command and its
 //argument in an array of strings
 INT8* pingargs[] = {"ping", "172.16.6.48", "5"};

 devSerial = (struct devCap *) malloc (sizeof(struct
  devCap));
 devSerial->devHdl = (VOID *)CONSOLE;
 devSerial->devType = 0;

 if( (TTYDevID = RZKDevOpen("TTYM",(RZK_DEV_MODE_t*)
  devSerial)) == (RZK_DEVICE_CB_t *)SYSERR )
 {
  printf("Can't open tty for SERIAL0\n");
  return SYSERR;
 }
 // TTYDevID is an integer value specifying the
 // device.
 x_ping(&Shl, TTYDevID, TTYDevID, TTYDevID,3,
pingargs);
}
```

Appendix B. Guidelines to Porting SNMP and PPP Applications

SNMP and PPP protocols are modified in ZTP v2.2.0 in terms of configuration and APIs. This section explains the differences between ZTP v2.1.0 (and earlier) and ZTP v2.2.0 and later releases. This section provides details about porting the ZTP v2.1.0 applications to ZTP v2.2.0 and later releases.

For more information about the SNMP APIs, SNMP functions, PPP APIs, and PPP functions, refer to the Zilog TCP/IP Stack API Reference Manual (RM0040).

PPP and SNMP protocols are part of `CommoprotoLib.lib`, therefore the project settings need not be changed.

API Changes

Table 4 lists the API changes between ZTP versions.

Table 4. API Changes

	APIs	
Module	**ZTP v2.1.0**	**ZTP v2.2.0 and Later Versions**
SNMP	`void snmp_init(SN_TRAP_NOTIFY snTrapNotifyFunc)`	`INT16 ztpSnmpV1Init( ZTPSNMP_TRAP_NOTIFY snTrapNotifyFunc )`
	`void snmpv2_init(SN_TRAP_NOTIFY snTrapNotifyFunc)`	`INT16 ztpSnmpV2Init( ZTPSNMP_TRAP_NOTIFY snTrapNotifyFunc )`

Table 4. API Changes (Continued)

Module	APIs	
	ZTP v2.1.0	**ZTP v2.2.0 and Later Versions**
SNMP (cont'd.)	`void snmpv3_init(SN_TRAP_NOTIFY snTrapNotifyFunc)`	`INT16 ztpSnmpV3Init( ZTPSNMP_TRAP_NOTIFY snTrapNotifyFunc )`
	`INT16 TrapGen ( UINT8 Type, UINT32 Code, UINT16 NumObjects, SN_Object_s *pObjList )`	`INT16 snmpGenerateTrap ( INT8 *userName, UINT8 *pMgrAddress, UINT8 Type, UINT32 Code, UINT16 NumObjects, SNMPObj *pObjList )`
PPP	`void ppp_init(INT8 * serdev)`	`INT16 ztpPPPInit(void)`
	`void ppp_stop( void )`	`INT16 ztpPPPStop(void)`

Table 5 lists the changes to the configuration files in terms of data structures and variable names.

Table 5. Data Structure Changes

Module	Data Structures		Header File(.h) References (if any)	Exposed .c File	Remarks
	ZTP v2.1.0	ZTP v2.2.0			
SNMP	struct mib_info	SNMPMIBData	Located in snmpmib.h file at ..\ZTP\Inc	snmib.c	Functionality wise, SNMPMIBData remains the same as struct mib_info, but SNMPMIBData is an optimized version of mib_info.
	struct SN_Descr_s	SNMPDisplayStr	Located in snmpv1.h file at ..\ZTP\Inc	snmp_conf.c	
	struct SNMP_TABLE_S	SNMP_TABLE_S	Located in snmpmib.h at ..\ZTP\Inc	snmib.c	
	struct SN_Object_s	SNMPObjLs	snmpv1.h	snmib.c	
	snmp_conf.c	snmp_conf.c		snmp_conf.c	all of the variable names are changed

Index

A
About This Manual viii
add_cgi function 44
Adding Objects to the MIB 84
Address Resolution Protocol 4
Address Resolution Protocol, Reverse 4
ARP 4

B
BOOTP, How to Use 64
Build Options 31
Building Web Pages 50

C
CD 109
CGI code 44
CGI function 41, 43
CGI Functions 46
CGI functions 51
CGI routine 50
Closing a Connection to a Remote Host 59
configure the client browser, limitation of ZTP HTTPS 72
Configuring Text Telephony 28
Configuring the Network 28
configwlan 163
Connecting to a Remote Host Across a Network 58
Connecting to an FTP Server 62
copy 112
Counter 81
Crystal CS8900A 1

D
Defining the HTTP Header 39
Defining the HTTP Method 37
Defining the Website Structure 40
del 114
deldir 115
deltree 117
DHCP 1, 4
DHCP, How to Use 64
dir 120
DisplayString 80
DNS 1, 4, 66
DNS IP address 66
DNS, How to Use 66
DNS-formatted message 66
Domain Name Server 4
Dynamic Host Configuration Protocol 4
Dynamic web pages 43

F
File Transfer Protocol 4
FTP 1, 4
FTP Client, How to Use 62
FTP Server, How to Use 61

G
Gauge 81
gettime 126

H
How to Add a Table to the MIB 89
How to Use BOOTP 64
How to Use DHCP 64
How to Use DNS 66
How to Use HTTP 36
How to Use IGMP 67
How to Use SMTP 54
How to Use SNMP 77
How to Use TELNET 57
How to Use TFTP 53
How to Use the HTTPS Server 71
How to Use the Serial Ports 74
How to Use the Shell 74
How to Use the Telnet Client 57
How to Use TIMEP 68
HTTP 1, 4
HTTP applications 46
HTTP method 39
HTTP response 50
HTTP, How to Use 36
http_defheaders 39
http_defmethods 37, 38, 39
http_init function 37, 39, 40
http_request 50
http_request structure 39
https_init API 71
HyperText Transfer Protocol 4

I
ICMP 1, 4
IGMP 1, 4
Initializing HTTP 36
Intended Audience viii
Internet Control Message Protocol 4
Internet Group Management Protocol 4
Internet Protocol 4
IP 1, 4
IP address 54, 55, 66
IP address, DNS 66
IpAddress 81
Issuing FTP Commands 63

K
keyIndex 165

L
Limitations, HTTPS server 72
Login With A Username and Password 63

M
Manual Objectives viii
md 133
mimetype 41, 43
move 136

O
Object Names 78
Object Types 79
Online Information x
Organization ix
OS Plane 2

P
pass-phrase 166
PhysAddress 81
Point-to-Point Protocol 4
port 443, secure HTTPS server 72

port 80, non-secure HTTP server 72
PPP 1, 4
pppstart 141

R
RARP 1, 4
Related Documents ix
ren 147
rendir 148
Requesting the Time 68
Reverse Address Resolution Protocol 4
RZK Zilog Real Time operating system viii

S
Safeguards x
Secure Socket Layer Protocol 4
Sending Data to a Remote Host 60
settime 150
Simple Mail Transfer Protocol 4
Simple Network Management Protocol 4
sleep 151
SMTP 1, 4
SMTP port 55
SMTP server 55
SMTP, How to Use 54
SNMP Objects 81
SSL 1, 4
Stack Plane 2
Static web pages 42
SysContact 29
SysDescr 29
SysLocation 30
SysName 29
SysObjectID 29
SysServices 30
System Features 1

T
TCP/IP protocol stack 64
TCP/IP software libraries viii
TCP/IP stack 36
TCP/IP stack protocol 46
TELNET 1, 4
TFTP 1, 4
TFTP server 53
TFTP, How to Use 53
The Fourth parameter 41
The path parameter 41
The port Parameter 46
The SNMP_GET_FUNC Support Routine 92
The SNMP_NEXT_FUNC Support Routine 96
The SNMP_SET_FUNC Support Routine 94
The type parameter 41
Time Protocol 4
TIMEP 1, 4
TIMEP, How to Use 68
TimeTicks 81
Trademarks x
Transmission Control Protocol 4
Trap Generation 99
type 155

U
UDP 1, 4

UDP connection 53
UDP datagram 66
Updating SNMP Values 98
User Datagram Protocol 4
u_short snmp_max_object_size 30
Using SNMP to Manipulate Leaf Objects
in the MIB 87
Using ZTP 36

V

VOL 156

W

warning messages, limitations of ZTP
HTTPS 73
web client 36
Working with Tables 88

Z

Zilog TCP/IP Software Suite viii, 1
ZTP Software 2

zilog
Embedded in Life
An IXYS Company

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 at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/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.